University of Brasília

Exact Sciences Institute
Computer Science Department

A Socio-Technical Grounded Theory about Source Code Rejuvenation

Walter Lucas Monteiro de Mendonça

Thesis submitted in partial requirement for the
Doctoral Degree in Informatics

Advisor
Prof. Dr. Rodrigo Bonifácio de Almeida

Coorientador
Prof. Dr. João Alexandre Baptista Vieira Saraiva

Brasília
2026

University of Brasília

Exact Sciences Institute
Computer Science Department

A Socio-Technical Grounded Theory about Source Code Rejuvenation

Walter Lucas Monteiro de Mendonça

Thesis submitted in partial requirement for the
Doctoral Degree in Informatics

Prof. Dr. Rodrigo Bonifácio de Almeida (Advisor)
Universidade Federal de Pernambuco

Prof. Dr. Kiev Santos da Gama
Universidade Federal de Pernambuco

Prof. Dr. Eduardo Henrique Da Silva Aranha
Universidade Federal do Rio Grande do Norte

Prof. Dr. André Cavalcante Hora
Universidade Federal de Minas Gerais

Prof.a Dr.a Genaina Nunes Rodrigues
CIC/UnB

Prof.a Dr.a Cláudia Nalon
Coordenador of Postgraduate Program in Informatics

Brasília, April 23 2026

Dedicated to

I dedicate this thesis to the memory of my father, *Walter Pereira de Mendonça*, and my mother, *Jaidete Monteiro de Mendonça*. Thank you for your example of faith, honesty, and perseverance, for your unwavering integrity, and for believing in education as a force for transformation. Your legacy lives on in me.

Acknowledgements

First and foremost, I express my gratitude to God for sustaining me throughout this long journey, and to my family for their unwavering support.

In a very special way, I thank my partner and wife, Ana Carolinne, for her patience and understanding during the many difficult moments, and for her constant support. Always by my side, she helped me overcome challenges and remain focused on our goals. Her presence was essential in enabling me to move forward, even in the most sensitive and painful situations.

I am also deeply grateful to my niece, Elisa Martins, for her love, affection, and contagious joy, which continuously inspired me to keep going.

I sincerely thank my advisor, Prof. Dr. Rodrigo Bonifácio, for his patient guidance, valuable insights, and thoughtful advice throughout this research. I am also grateful for his friendship and for his support both within and beyond the academic environment. I also extend my thanks to Prof. Dr. João Saraiva for welcoming me into his research group at the University of Minho, and for his teachings and friendship.

I would also like to express my gratitude to the Graduate Program in Informatics at the University of Brasília and to the Department of Computer Science for their support and assistance. I thank all faculty and staff for their kindness, guidance, and support, as well as the members of the examination committee for their time and for their valuable contributions to improving this work.

Finally, I would like to acknowledge my friends from the laboratory, work, and life: Fausto Carvalho, Luís Amaral, Prof. Dr. Edna Canedo, Ranyelson Neres, Pedro Costa, Handrick Costa, Leomar Camargo, Roberto Roselló, Prof. Dr. Thiago Paulo, Nilson Guerin, Guilherme Branco, Elton Sarmanho, Daniel Souza, Rui Rua, Jerome Kempf, and José Clavo. Their companionship made this journey lighter, filled with moments of joy and meaningful exchanges of knowledge.

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Uma Teoria Fundamentada Sociotécnica sobre o Rejuvenescimento do Código-fonte

Resumo

A evolução contínua das linguagens de programação trouxe benefícios e novos desafios para os desenvolvedores de software. Nos últimos anos, observou-se um ritmo acelerado no lançamento de novas versões de linguagens de programação. Contribuintes e mantenedores dessas linguagens sugerem que tais avanços melhoram a produtividade e a qualidade do software. No entanto, esse cenário impôs um desafio recorrente aos desenvolvedores: como manter sistemas legados atualizados frente à constante evolução das linguagens? Esta tese investigou esse problema ao aprofundar o entendimento sobre o fenômeno do rejuvenescimento de código-fonte, entendido como um tipo particular de migração de software cujo objetivo é substituir construções e padrões legados por alternativas modernas, evitando a obsolescência dos sistemas. O objetivo central foi compreender as motivações, desafios e práticas adotadas por desenvolvedores ao conduzir esforços de rejuvenescimento em sistemas em constante evolução. Para alcançar esse objetivo, foi conduzido um estudo de métodos mistos, fundamentado na abordagem de Socio-Technical Grounded Theory (STGT). A tese foi composta por quatro estudos complementares. Realizaram-se, inicialmente, entrevistas semiestruturadas com 23 desenvolvedores profissionais, com o objetivo de compreender percepções, desafios e práticas relacionadas ao rejuvenescimento de código. Esses dados qualitativos motivaram e orientaram a realização de três estudos de mineração de repositórios de software em larga escala, abrangendo diferentes linguagens de programação (C++, JavaScript e Python). Tais estudos combinaram análise de séries temporais para caracterizar tendências de adoção de funcionalidades modernas, evidências qualitativas oriundas de discussões em pull requests e um inquérito com desenvolvedores que realizaram esforços de rejuvenescimento. A partir da integração dos estudos de mineração com os dados das entrevistas, foram conduzidas 6 novas entrevistas em um estudo final, em que as informações foram analisadas de forma iterativa e comparativa, resultando na construção de uma teoria sociotécnica emergente provinda desses dados. Como resultado, esta tese propôs uma teoria sociotécnica que explicou como, quando e por que o rejuvenescimento de código ocorreu em sistemas legados em constante evolução. Os resultados mostraram que o rejuvenescimento não ocorre de forma uniforme, sendo influenciado por fatores técnicos, organizacionais e humanos, bem como por estratégias adotadas pe-

los desenvolvedores para gerenciar riscos, custos e esforços. Por fim, esta tese contribuiu tanto para a academia quanto para a prática, oferecendo uma explicação abrangente do fenômeno e fornecendo subsídios para apoiar desenvolvedores e organizações na condução de esforços de rejuvenescimento de forma segura e eficaz.

Palavras-chave: Teoria Fundamentada Sociotécnica, Rejuvenescimento de código-fonte, Engenharia de Software, Evolução de Software, Manutenção de software

Abstract

The continuous evolution of programming languages has brought benefits and new challenges for software developers. Recently, there has been an accelerated pace in the release of new versions of programming languages. Contributors and maintainers of these languages suggest that such advancements improve productivity and software quality. However, this scenario has imposed a recurring challenge on developers: how to keep legacy systems updated in the face of the constant evolution of languages? This thesis investigated this problem by deepening the understanding of the phenomenon of code rejuvenation, understood as a particular type of software migration whose objective is to replace legacy constructs and patterns with modern alternatives, avoiding system obsolescence. The central objective was to understand the motivations, challenges, and practices adopted by developers when conducting rejuvenation efforts in constantly evolving systems. To achieve this objective, a mixed-methods study was conducted based on the Socio-Technical Grounded Theory (STGT) approach. The thesis was composed of four complementary studies. Initially, semi-structured interviews were conducted with 23 professional developers, with the aim of understanding perceptions, challenges, and practices related to code rejuvenation. These qualitative data motivated and guided the conduct of three large-scale software repository mining studies, covering different programming languages (C++, JavaScript, and Python). Such studies combined time series analysis to characterize trends in the adoption of modern features, qualitative evidence from discussions in pull requests, and a survey with developers who undertook rejuvenation efforts. Through the integration of mining studies with interview data, six new interviews were conducted in a final study, where the information was analyzed iteratively and comparatively, resulting in the construction of an emerging sociotechnical theory derived from these data. As a result, this thesis proposed a sociotechnical theory that explained how, when, and why code rejuvenation occurred in constantly evolving legacy systems. The results indicated that rejuvenation does not occur uniformly, being influenced by technical, organizational, and human factors, as well as by strategies adopted by developers to manage risks, costs, and efforts. Finally, this thesis contributed both to academia and practice, offering a comprehensive explanation of the phenomenon and providing support

for developers and organizations in conducting rejuvenation efforts safely and effectively.

Keywords: Socio-Technical Grounded Theory, Source code rejuvenation, Software Engineering, Software Evolution, Software Maintenance

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Programming languages have been evolving at an increasingly rapid pace, with new versions, specifications, and feature proposals released far more frequently than in previous decades [8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18]. This acceleration has driven the introduction of a wide range of modern features in mature languages with more than 30 years of evolution (e.g., Java, Python, and C++), including new syntactic abstractions, richer type systems, more declarative module mechanisms, and functional control-flow constructs. These features are often intended to improve code clarity, safety, and maintainability, and—by reducing verbosity—in some contexts make programs easier to read and reason about [15, 13, 16, 17, 19, 20].

However, despite the benefits of new language constructs, their coexistence with older constructs—retained for backward compatibility—often leads to legacy codebases in which multiple "temporal dialects" of the same language coexist. As a result, code repositories tend to evolve unevenly: newly written modules increasingly adopt recent idioms, while substantial portions of long-lived systems continue to rely on outdated patterns or deprecated styles accumulated over years of evolution [21, 9, 10, 11].

As a result, developers responsible for the long-term evolution of systems must understand multiple idioms and constructs of the same language, substantially increasing cognitive load [9]. Empirical observations indicate that real-world projects maintain mixtures of legacy and modern ways for solving a problem using a language for long periods [14, 22, 23, 9, 10, 11], reinforcing that language evolution often outpaces code evolution. At the same time, manually migrating large amounts of legacy code to modern constructs is labor-intensive, error-prone, and requires large maintenance efforts.

To mitigate these burdens, prior research argues that automated refactoring and source code rejuvenation tools are essential mechanisms for keeping code aligned with language evolution [8, 9, 24, 10, 11]. Overbey and Johnson, for instance, conceptualize refactorings as integral to language implementations, enabling cost-effective removal of obsolete

idioms [8]. Pirkelbauer et al. propose automated rejuvenation to ease upgrades to new language versions and support IDEs in providing live, educational transformations [9]. Tools such as LambdaFicator [10], RJTL [11], and DemacroFication [24] demonstrate how automated transformations can systematically retrofit modern constructs into legacy code.

Prior studies have examined transitions between language versions, most notably the migration from Python 2 to Python 3, characterizing adoption as a long-term, incremental process shaped by compatibility constraints, dependencies, and migration costs [14, 22]. Other work has focused on the evolution of individual language features, such as Python type annotations, analyzing when they first appear, how their usage grows, and how developers progressively adapt to increasingly expressive type systems [23]. Feature-specific adoption has also been studied in the context of Java, including the uptake of generics [25] and Java 8 constructs such as lambda expressions and streams [26], revealing gradual and selective adoption driven by cognitive overhead, refactoring effort, and shifts in programming style.

Complementary studies have analyzed how developers use core libraries and abstractions, showing that advanced constructs are often underutilized despite being widely available [27]. Finally, large-scale mining studies in Python have examined the introduction and diffusion of new language features over time, identifying substantial delays between feature release and widespread use across projects [28].

Our previous empirical studies on language feature evolution, such as the adoption of Java lambda expressions, reveal contradictory and context-dependent effects on readability and highlight the complexity of developers' perceptions [3, 19]. Finally, studies on software modernization and migration outlines high-level organizational benefits, such as increased agility, flexibility, and cost reduction [29, 30]. ***Despite these contributions, no prior study provides an in-depth empirical examination of the motivations, challenges, and situated practices through which professional developers perform source code rejuvenation***. This gap suggests the need for a grounded theory explaining the socio-technical dynamics of rejuvenation in legacy systems.

1.1 Goals and Methods

The main goal of this thesis is to develop a theory that explains the phenomenon of source code rejuvenation—i.e., the process by which developers decide whether, when, and how to replace legacy language constructs with modern ones in their legacy systems. In this thesis, we use the term *modern language features* to refer to language constructs intro-

duced in recent versions of programming languages (e.g., ES6+ for JavaScript, Python 3.x, C++11 and later).

From a methodological perspective, we adopted Socio-Technical Grounded Theory (STGT) to develop an explanatory theory of source code rejuvenation. Following STGT principles, we applied iterative cycles of open coding, focused coding, constant comparison, theoretical sampling, memoing, and theory structuring. This methodological framework enables us to capture how developers perceive, negotiate, and perform rejuvenation efforts within specific socio-technical contexts, balancing constraints, improvement expectations, team dynamics, and the evolving design of programming languages.

Within this broader STGT process, we conducted three large-scale empirical studies to form a single socio-technical theoretical sampling round based on existing data. Each study combines mining of software repositories with qualitative analysis of socio-technical artefacts such as pull-request discussions, pull-request comments, and survey responses (Study A (C++), Study B (JavaScript), and Study C (Python)). We integrate these data into the STGT analysis by re-coding selected excerpts using open and focused coding, comparing them constantly with interview data, and developing memos that connect our observations to the emerging categories of the theory.

1.2 Research Questions

The study is guided by the following research questions, which structure the theoretical sampling process and the analytical focus:

RQ1: What are the motivations that lead software developers to rejuvenate their software?

RQ2: What are the challenges that hinder software developers from rejuvenating their legacy code?

RQ3: What practices do developers follow or propose to deal with the challenges of source code rejuvenation?

These research questions are not treated as hypotheses to be tested but as sensitizing guides that orient theoretical sampling and analytic focus throughout the study. RQ1 directs early data collection and open coding toward understanding the motivating conditions underlying source code rejuvenation, while RQ2 guides the analysis of constraining and contextual factors that shape developers' decisions and actions. Finally, RQ3 focuses attention on action and interaction strategies, capturing how developers respond

to challenges in practice. Together, these questions evolve through constant comparison and memoing, supporting an iterative, inductive theory-building process grounded in empirical data rather than predefined assumptions.

1.3 Contributions

This thesis comprises four studies. Two studies were published [3, 4], and two others are being prepared for submission in the near future. Each study in this thesis addresses a specific aspect of the source code rejuvenation phenomenon, contributing complementary evidence toward the development of the emergent theory.

Study A investigated the adoption of modern C++ features in a large-scale empirical study of open-source systems. Through the analysis of 272 KDE projects, this study identified how developers incorporated modern constructs into legacy systems and the practical challenges involved in large-scale rejuvenation efforts. In addition to mining software repositories, this study included a survey with developers who performed identified rejuvenation efforts, providing direct insights into their motivations and challenges. Furthermore, it analyzed adoption trends through time-series analysis to characterize how feature usage evolved over time. Its contribution was providing evidence of real-world rejuvenation practices and highlighting the socio-technical trade-offs involved in source code rejuvenation legacy C++ systems.

Study B analyzed the adoption of modern JavaScript features following the same methodological approach as Study A. By examining 158 open-source projects, this study combined repository mining, time-series analysis of feature adoption, and qualitative analysis of pull-request discussions to understand how and when modern features were adopted. The results revealed rapid and delayed adoption cycles and large-scale rejuvenation efforts. Its contribution lay in identifying temporal adoption patterns and reinforcing that rejuvenation was a recurring and systematic phenomenon in evolving environments, while also capturing developer perceptions through collaborative discussions.

Study C followed the same methodological structure, focusing on the adoption of modern Python features. It combined large-scale mining of 424 repositories, time-series analysis to identify adoption trends, and qualitative analysis of pull-request discussions. This study identified distinct adoption patterns (fast vs. slow), quantified adoption delays, and uncovered key motivations and challenges driving rejuvenation decisions. Its contribution was providing a deeper understanding of the "how, when, and why" of rejuvenation, linking longitudinal quantitative trends with developer rationale observed in practice.

The **STGT Core Study** represents the central contribution of this thesis. It integrates evidence from semi-structured interviews, mining studies, surveys, developer discussions, and iterative analysis to construct a socio-technical grounded theory that explains how, when, and why source code rejuvenation occurs in evolving software systems under continuous language evolution. This study synthesizes motivations, challenges, enabling and constraining conditions, strategies, covariances, and consequences into a cohesive explanatory framework.

Overall, the C++, JavaScript, and Python mining studies, together with the interviews, played a fundamental role in the construction of the emergent theory by providing complementary empirical evidence across different programming languages, contexts, ecosystems, and methodological approaches. These studies supported the processes of data collection, analysis, refinement, expansion, and theoretical sampling, enabling the theory to be strengthened and extended beyond the initial interview findings. As a result, this thesis advances a theory that explains source code rejuvenation as a socio-technical maintenance process through which developers and organizations negotiate source code rejuvenation, risk, quality, and long-term system viability in response to the rapid evolution of programming languages and software ecosystems.

These contributions are further reported through theoretical hypotheses, implications, and practical directions derived from the emergent theory. The hypotheses synthesize the relationships among the main theoretical categories and indicate how source code rejuvenation tends to emerge, vary, and produce consequences in practice. The implications and directions, in turn, translate the theory into guidance for developers and organizations, supporting decisions about when, why, and how to prioritize and operationalize source code rejuvenation in evolving software systems.

Data Availability: the datasets generated and analyzed across the studies of this thesis are publicly available in a dedicated replication repository. This repository contains the collected data, processed datasets, and supporting materials used in the empirical analyses, enabling transparency and reproducibility of the results. The data can be accessed at: `https://github.com/waltim/socio-technical-grounded-theory-scr`.

1.4 Thesis Organization

This thesis is organized to progressively introduce the phenomenon of source code rejuvenation, establish the theoretical and methodological foundations of the research, present the grounded theory emerging from the empirical investigation, and articulate its implications for research and practice.

Chapter 1 introduced the research problem, the gap in the current understanding of source code rejuvenation, the overarching goals of this thesis, and the research questions guiding the study.

Chapter 2 presents the conceptual and technical background necessary to contextualize this work. It first revisits the foundations of software aging and maintenance, including maintenance types, maintenance processes, and their role in long-lived systems. Then, it introduces program transformation, refactoring, and source code rejuvenation, like the phenomenon investigated.

Chapter 3 describes the methodological approach adopted in this thesis. It details the Socio-Technical Grounded Theory (STGT) procedures used for data collection, coding, comparison, theoretical sampling, saturation, and theory construction. It also explains the multi-study research design, clarifying how the qualitative investigation is complemented by evidence from three large-scale mining studies.

Chapter 4 presents the main theoretical result of this thesis: a socio-technical grounded theory of source code rejuvenation. The theory is presented through Glaser's Six C's template, which is used to organize and explain the phenomenon by means of *Context*, *Conditions*, *Causes*, *Consequences*, *Contingencies*, and *Covariances*.

Chapter 5 discusses the emergent theory and is organized into five main parts. First, Section 5.1 answers the research questions through the emergent theory. Second, Section 5.2 presents the theoretical hypotheses derived from the relationships among the Six C's categories. Third, Section 5.3 presents the implications of the findings for developers and organizations. Fourth, Section 5.4 positions the emergent theory in relation to the existing literature. Finally, Section 5.5 discusses the limitations of the study.

Chapter 6 presents the final conclusions of the thesis and outlines directions for future work.

Finally, the appendices provide supplementary material related to the empirical studies and the structure of the emergent theory. Appendix A presents the C++ study, Appendix B presents the JavaScript study, and Appendix C presents the Python study. In addition, Appendix D provides information about the participants involved in Rounds 1 and 3, while Appendix E presents the hierarchical summary of the Six C's structure.

Chapter 2

Background

2.1 Software Aging

Software aging, as described by Parnas [31], refers to the inevitable phenomenon where software products, similar to human aging, become legacies that require increasing support efforts. Software ages and it behaves similarly to products that deteriorate over time: it becomes difficult to maintain and loses the ability to meet the needs of users and the market. Aging occurs mainly for two reasons: the absence of necessary modifications to keep up with external changes and the negative effects of modifications made over time.

The effects of this aging appear in the increasing difficulty of adding new functionalities, the degradation of the program's internal structure, the reduction in performance in terms of space and time, and the increased propensity to errors. Moreover, repeated modifications, made without a proper understanding of the original design concept, worsen the program's structure, increase the cost of future changes, and raise the likelihood of introducing new bugs. As a consequence, the software owners find it difficult to keep up with the market, while the product becomes less reliable, less efficient, and more expensive to maintain [31, 32].

To mitigate this aging, organizations and development teams can benefit from preventive measures and treatment measures for already aged software. Among the preventive measures, the principle of designing for change stands out, using abstraction and information hiding to isolate parts that are more likely to be altered. The idea is to estimate possible future changes and organize the design in such a way that these changes are confined to small sections of the code. Documentation is also presented as essential, as it must record principles and design decisions; when it is neglected, future changes become more difficult and costly [31, 32]. Moreover, rigorous design reviews performed during software maintenance tasks are pointed out as a way to preserve the structural quality of software over time [31].

2.2 Software Maintenance

Software maintenance can be understood as a set of individual changes made to a software system throughout its lifecycle [33]. According to the ISO/IEC/IEEE 14764:2022 standard [34], software maintenance is categorized into six main types: corrective, adaptive, perfective, preventive, additive, and emergency. Corrective maintenance involves fixing faults discovered post-delivery. Adaptive maintenance ensures the software continues to work in evolving environments, such as after an OS upgrade. Perfective maintenance enhances performance or maintainability by improving code, documentation, or user interfaces. Preventive maintenance addresses latent faults before they cause issues in live systems. Additive maintenance, which some organizations consider separately, involves adding new features or functionality beyond corrections or improvements. Lastly, emergency maintenance is a reactive, unscheduled intervention to keep systems operational until proper fixes can be applied. These types are often initiated by modification requests (MRs) or problem reports (PRs) and may be further classified as reactive (corrective and adaptive) or proactive (preventive, perfective, and additive) forms of maintenance.

According to Bennett and Rajlich [35], software maintenance is a process used in almost all useful and successful software due to the recurring need to incorporate improvements, such as the inclusion of new functionalities, and changes, such as the correction of program problems. Software maintenance is an inevitable and strategic phase in the software life cycle, driven largely by user demands for enhancements rather than mere bug fixes. Studies [36, 37, 38] show that 50–70% of maintenance efforts focus on perfective and adaptive changes, not corrective actions. Lehman's laws of software evolution reinforce that software must continually adapt to remain useful in a dynamic environment, leading to persistent growth in complexity and maintenance effort [39]. Key challenges include poor documentation, staff turnover, and inconsistent user expectations—all of which exacerbate costs and delay implementation.

To mitigate these challenges, researchers point out that there are well-defined and systematic maintenance processes supported by clear documentation, robust architectural design, and continuous regression testing. The ISO/IEC/IEEE 14764:2022 standard defines the maintenance process as a structured set of coordinated activities that support the continued operability, adaptability, and effectiveness of software systems throughout their life cycle [34]. The process includes six fundamental activities, which are:

- Process Implementation: Involves setting up the maintenance capability, including agreements, tools, roles, and procedures needed to execute the maintenance process. It ensures readiness before any technical work begins.

- Problem and Modification Analysis: Focuses on analyzing modification requests (MRs) and problem reports (PRs), performing impact assessments, feasibility studies, and prioritizing changes.
- Modification Implementation: Entails the design, coding, testing, and integration of the approved changes into the software system. It includes validation that the change fulfills its intended purpose.
- Maintenance Review/Acceptance: Reviews the implemented changes with stakeholders, verifies quality, and obtains formal acceptance before deployment into the operational environment.
- Migration: Handles the movement of modified or upgraded software to the target environment, ensuring compatibility and minimal disruption during transition.
- Software Retirement: the orderly removal of software or its components from active service, including archiving, data migration, and decommissioning tasks.

These activities provide a framework for systematically managing changes—from identifying a problem to retiring obsolete components—and they are essential for maintaining software quality, minimizing disruption, and aligning with stakeholder needs. Despite the importance of software maintenance activities, some studies [40, 41, 42] show that developers have not always perceived the importance of these activities, which, in turn, can lead to neglect of maintenance. However, later studies [43, 37, 44] point to a positive change in developers' perception of software maintenance, which has come to be considered an evolutionary development activity.

2.3 Software Evolution and Migration

Software evolution refers to the continuous transformation of a software system from a lesser or incomplete state toward a better, more capable one, driven by emerging user needs, environmental changes, and expanding stakeholder knowledge [45, 46]. Unlike hardware, which is repaired by replacing worn physical parts, software evolves conceptually: it must be repeatedly modified so that it continues to deliver value and does not degrade in usefulness over time. As Lehman's early principles emphasized, real-world software must continually change to remain relevant, because systems that remain static become progressively less aligned with their operational environment and user expectations [46]. This ongoing adaptation is fundamental to sustaining software reliability, functionality, and maintainability throughout its lifespan.

The activities that shape software evolution can be understood through Lehman's evolving set of laws, which provide an explanatory framework for why and how long-lived software must change. These laws highlight several recurring phenomena: systems require continuing change to satisfy emerging needs [47, 46]; unchecked modifications lead to increasing complexity unless explicit effort is invested in controlling it; evolutionary processes tend to be self-regulating as organizational activity patterns stabilize; and both system growth and stakeholder familiarity must progress incrementally to avoid overwhelming developers and users [45]. Further laws emphasize that the functional content of systems undergoes continuing growth, system qualities will decline unless design corrections and environmental adaptations occur, and evolution operates as a feedback-driven process involving multi-level interactions between stakeholders, processes, and system artifacts. Collectively, these observations show that evolution is not a random accumulation of changes but a structured, intrinsically constrained socio-technical process [46, 30].

Different types of evolution arise depending on how a system interacts with its environment and how its underlying problem domain changes over time. Lehman's SPE classification distinguishes S-type programs with fixed specifications, P-type programs that solve well-defined but complex problems, and E-type programs whose problem contexts change continuously and thus require persistent evolution to remain viable [46]. As E-type systems age, organizations often confront the need for more substantial transformations, leading to the domain of software migration—the process of moving operational systems to new platforms, architectures, or technologies while retaining essential functionality. Migration is therefore not separate from software evolution but a natural manifestation of it: as systems continue to grow, accumulate complexity, and encounter shifting technological environments, migration becomes a strategic evolutionary response that restructures or replatforms software to ensure its long-term sustainability, usability, and organizational value. Bragagnolo et al. [30] identify, through a bottom-up literature-based taxonomy, several migration objectives that can be applied during software modernization. These objectives represent concrete forms of software migration found in industrial and academic studies, including:

- **Service migration**, in which existing application functionalities are exposed or restructured as services [48].
- **Library migration**, which consists of replacing or delegating responsibilities to a different library or framework [49].
- **Language migration**, involving the translation of source code from one programming language to another (e.g., C to Java [50]).

- **Paradigm migration**, which entails reorganizing program structure and semantics, such as moving from procedural to object-oriented code [51].

Beyond these forms of migration, Pirkelbauer et al. [9] also discuss migrations that occur within a programming language, particularly across versions. This type of evolution involves transforming source code to adopt new language constructs. They classify such transformations as a form of source code rejuvenation, since the goal is to update the codebase to a more modern, maintainable, and expressive version of the same language. In this work, we focus specifically on this kind of migration — that is, migrations whose main goal is to replace legacy language constructs by the new ones available in more recent versions of a programming language.

2.4 Program Transformation and Source Code Rejuvenation

Program transformation refers to systematic modifications of programs while preserving or intentionally altering specific semantic aspects to achieve a particular goal [52]. As Visser notes, programs are structured artefacts with well-defined semantics, which enable their transformation through formal or semi-automated means. Transformations may vary widely in purpose: they can preserve the observable behavior of a program, improve its design, increase its performance, modernize its constructs, or extract new representations at different abstraction levels. From this perspective, program transformation constitutes a unifying framework that encompasses activities such as compilation, optimization, refactoring, renovation, program synthesis, and reverse engineering.

In his taxonomy, Visser classifies program transformation into two major categories: Translation and Rephrasing. Translation refers to transformations where the source and target languages differ, such as compilation, migration, synthesis, and reverse engineering. Rephrasing, by contrast, involves rewriting a program into another form within the same language. This includes normalization, optimization, refactoring, and renovation. These categories capture a wide continuum of transformations that range from lowering abstraction levels (e.g., synthesis), lateral restructuring at equivalent abstraction (e.g., migration), and upward movement to higher abstraction (e.g., reverse engineering). Within rephrasing, we find transformations that maintain functionality, such as refactoring [53], those that improve performance, such as optimization, and those that intentionally alter extensional behavior to repair or update legacy systems, such as source code rejuvenation [9].

2.4.1 Refactoring

Refactoring is defined by Fowler as a disciplined process of improving the internal structure of existing code while strictly preserving its external behavior [53]. Its purpose is to improve readability, reduce complexity, eliminate duplication, and enable future modifications [11, 53]. Refactorings are systematic transformations—such as Extract Method, Inline Variable, or Replace Conditional with Polymorphism—that target structural deficiencies in the code.

A representative example is Fowler's Substitute Algorithm[1] refactoring, which replaces a complicated or inefficient algorithm with a clearer and more direct one while ensuring that the resulting behavior remains unchanged. For example, an imperative loop computing the average of a collection may be replaced by a more concise formulation leveraging built-in abstractions, using built-in language features or a clearer control structure. The essence of the refactoring lies not in altering the semantics of the program but in improving the clarity and maintainability of the algorithmic expression. In this sense, refactoring is tightly concerned with behavioral preservation and structural improvement, enabling developers to work effectively with evolving codebases [53].

2.4.2 Source Code Rejuvenation

Source code rejuvenation, as introduced by Pirkelbauer, is a source-to-source transformation that replaces outdated idioms and deprecated constructs with modern language features and higher-level abstractions [9]. Unlike refactoring—which focuses primarily on structural improvements—source code rejuvenation is fundamentally tied to the evolution of programming languages and aims to rejuvenate code so that it aligns with contemporary language capabilities. Rejuvenation is a unidirectional process: it transforms code written in older idioms into equivalent or improved representations that leverage the abstractions of the newer language version.

For example, Figure 2.1 illustrates a common rejuvenation step discussed in *Exploring ES6*: replacing ES5-style string concatenation with ES6 template literals. Legacy code often builds formatted strings by combining fragments and values manually, which tends to be verbose and error-prone (e.g., misplaced delimiters). In contrast, template literals provide direct interpolation, making formatting intent explicit and improving readability and maintainability [54].

The transformation reduces boilerplate and clarifies formatting intent, aligning legacy code with modern JavaScript idioms. Additionally, Figure 2.2 highlights another fre-

[1] Martin Fowler, "Substitute Algorithm," *Refactoring Catalog*, available at: `https://refactoring.com/catalog/substituteAlgorithm.html`, accessed on February 25, 2026.

```
function printCoord(x, y) {
    console.log('(' + x + ', ' + y +
    ')');
}
```

Listing 2.1: Before (ES5): String concatenation

```
function printCoord(x, y) {
    console.log(`(${x}, ${y})`);
}
```

Listing 2.2: After (ES6): Template literal interpolation

Figure 2.1: Source code rejuvenation in JavaScript: replacing ES5 string concatenation with ES6 template literals.

quent rejuvenation: replacing ES5 function expressions used as callbacks with ES6 arrow functions. In legacy code, developers often introduced auxiliary variables (e.g., `_this`) to preserve the surrounding object context inside event handlers. Arrow functions avoid this pattern by capturing `this` lexically, resulting in more concise and less error-prone callbacks [54].

```
function UiComponent() {
    var _this = this;
    var button =
    document.getElementById('buttonX');
    button.addEventListener('click',
    function () {
        console.log('CLICK');
        _this.handleClick();
    });
}

UiComponent.prototype.handleClick =
    function () {
    // ...
};
```

Listing 2.3: Before (ES5): Callback with `_this` workaround

```
class UiComponent {
    constructor() {
        let button =
    document.getElementById('buttonX');

    button.addEventListener('click',
    () => {
            console.log('CLICK');
            this.handleClick();
        });
    }

    handleClick() {
        // ...
    }
}
```

Listing 2.4: After (ES6): Arrow function with lexical `this`

Figure 2.2: Source code rejuvenation in JavaScript: replacing ES5 callback patterns with ES6 arrow functions to preserve lexical `this`.

As a contribution of our thesis, in a previous empirical study [4], we found that transformations such as template literal adoption and arrow function migration are among the most common rejuvenation practices in modern JavaScript systems. Our mining results indicate that developers adopt these features primarily to improve readability,

reduce syntactic verbosity, and express intent more clearly—benefits that directly align with the motivations described in [54].

Similarly, in the C++ progamming language, legacy idioms often required verbose iterator declarations, manual incrementing, and explicit dereferencing—patterns that are error-prone and reduce readability [3]. Modern constructs, such as auto-typed variables, lambda expressions, and range-based for, simplify these patterns, reinforcing the broader empirical observation that rejuvenation efforts tend to prioritize improvements in code clarity, maintainability, and cognitive simplicity [1, 16].

Figure 2.3 shows an example of a transformation that replaces an explicit iterator-based loop with the Modern C++ `range-based for` construct introduced in C++11. The rejuvenated version also leverages the `auto` type deduction mechanism to eliminate redundant type declarations for the iteration variable. Together, these constructs reduce syntactic verbosity, prevent common iterator misuse errors (such as incorrect incrementing or dereferencing), and improve the semantic clarity of the iteration intent. By adopting range-based iteration and type inference, the code becomes more readable, less error-prone, and better aligned with contemporary C++ idioms, ultimately enhancing maintainability and long-term evolvability.

```
QList<QString>::iterator it;
for (it = names.begin(); it !=
   names.end(); ++it) {
    process(*it);
}
```

Listing 2.5: Before: Legacy iterator-based loop

```
for (const auto& name : names) {
    process(name);
}
```

Listing 2.6: After: Modern C++ with range-based for and ‘auto‘

Figure 2.3: Source code rejuvenation in C++: migrating from iterator-based looping to range-based for with type inference (based on Effective Modern C++ [1]).

Another example, in the C++ programming language, Figure 2.4 illustrates a transformation that replaces a `std::bind`-based callback with a Modern C++ lambda expression. While `std::bind` introduces an additional layer of indirection and placeholder semantics, lambda expressions provide explicit parameter lists and capture clauses, making the control flow and data dependencies more transparent. The rejuvenated version improves expressiveness by clearly specifying captured variables and argument types, thereby reducing cognitive load and simplifying reasoning about behavior. This transformation enhances readability, minimizes subtle binding-related errors, and aligns the code with modern C++ best practices that favor lambdas for local function objects.

```
using namespace std::placeholders;
auto handler =
    std::bind(&Widget::onEvent, &w,
    _1);

subscribe(source, handler);
```

Listing 2.7: Before: Indirect call via ‘std::bind‘

```
auto handler = [&w](const Event&
    e) {
     w.onEvent(e);
};

subscribe(source, handler);
```

Listing 2.8: After: Modern C++ lambda with explicit capture

Figure 2.4: Source code rejuvenation in C++: replacing ‘std::bind‘ indirection with a lambda and explicit capture (based on Effective Modern C++ [1]).

As an additional contribution of our thesis, in a previous study [3], we found that transformations such as the adoption of `auto`, lambda expressions, and range-based for loops are common in rejuvenation efforts and are frequently automated due to their syntactic regularity and semantic preservation. The study further indicates that rejuvenation is often performed through commits explicitly aimed at source code rejuvenation, sometimes supported by automated refactoring tools (e.g., `clang-tidy`).

Finally, regarding Python, the evolution of Python 3 introduced improvements that reinforce code clarity, readability, and robustness. Among the main advances are the gradual typing system with type hints, the asynchronous model (async/await), and improvements in expressiveness such as f-strings. These features make the developer's intent more explicit, facilitate maintenance, and allow the use of effective static analysis tools [55].

For instance, 2.5 show an example of rejuvenation that replaces an untyped function with explicit type annotations, as recommended in modern Python practices. By introducing parameter and return type hints, the transformation makes implicit interface contracts explicit, enabling static analysis tools and improving code comprehension. Although runtime behavior remains unchanged, the annotated version reduces ambiguity about expected input and output types, facilitates early detection of inconsistencies, and strengthens long-term maintainability. This transformation exemplifies rejuvenation that enhances semantic clarity without modifying functional semantics.

In another example, Figure 2.6, the rejuvenation transforms a synchronous I/O-bound function into an asynchronous implementation using `async/await`. The legacy version blocks execution while waiting for external resources, limiting scalability and responsiveness. By adopting native asynchronous constructs introduced in Python 3, the rejuvenated version enables non-blocking execution, improves concurrency handling, and aligns the code with modern standards. The transformation preserves logical behavior

```python
def calculate_total(price, quantity):
    return price * quantity
```

Listing 2.9: Before: Legacy function without type annotations

```python
def calculate_total(price: float,
    quantity: int) -> float:
    return price * quantity
```

Listing 2.10: After: Function rejuvenated with type hints (PEP 3107, PEP 526)

Figure 2.5: Source code rejuvenation in Python: introducing type hints to make interface contracts explicit and enable static analysis support, while preserving runtime behavior.

while enhancing performance characteristics and architectural flexibility in I/O-intensive contexts.

```python
import requests

def fetch_data(url):
    response = requests.get(url)
    return response.text
```

Listing 2.11: Before: Blocking I/O implementation

```python
import aiohttp
import asyncio

async def fetch_data(url: str) -> str:
    async with
    aiohttp.ClientSession() as
    session:
        async with session.get(url)
    as response:
            return await
    response.text()
```

Listing 2.12: After: Asynchronous implementation using async/await (PEP 492)

Figure 2.6: Source code rejuvenation in Python: transforming blocking I/O into non-blocking asynchronous execution using native async/await constructs.

Building upon the empirical findings of our Python study [5], this thesis contributes by clarifying how modern Python features like type hints and async/await are perceived within the rejuvenation process. The large-scale analysis shows that features associated with immediate improvements in readability and structural clarity tend to achieve broader and faster adoption across projects, while qualitative evidence indicates that developers explicitly associate source code rejuvenation efforts with gains in comprehension, safety, and performance.

Refactoring and source code rejuvenation share the common goal of improving code quality and promoting maintainable, comprehensible software. Both rely on systematic rule-based transformations and frequently operate on similar syntactic structures. How-

ever, they differ fundamentally in purpose and scope. Refactoring is primarily concerned with improving the design of code while strictly preserving its behavior, often targeting structural issues internal to the program. In contrast, source code rejuvenation is explicitly driven by language evolution: it replaces outdated constructs with modern abstractions, allowing improvements in expressiveness, safety, and performance. Whereas Fowler's Substitute Algorithm [53] refactoring improves a local algorithmic expression for clarity, rejuvenation substitutes entire obsolete idioms or patterns to bring legacy code into alignment with modern language standards. Together, these transformations provide complementary mechanisms through which software systems adapt, survive, and thrive amid the continuous evolution of programming languages and development practices.

Chapter 3

Research Methodology

This chapter presents the methodological procedures adopted in this research. Section 3.1 introduces the goals and questions that structured the overall research objective, define guiding research questions, and clarify the analytical focus of the study. Section 3.2 describes the Socio-Technical Grounded Theory (STGT) approach that underpins the qualitative and iterative nature of the investigation. Section 3.3 details the overall research design, outlining how the study was structured across multiple rounds. Section 3.4 presents the data collection and analysis procedures conducted in Round 1, including participant recruitment, coding processes, and early theoretical structuring through memoing. Finally, Section 3.5 describes Rounds 2 and 3 of theoretical sampling, covering the theoretically grounded cross-language sampling and triangulation strategy, the cross-language repository mining studies, the semi-structured interviews, and the procedures through which the emerging theory was refined, expanded, saturated, and finally structured.

3.1 Goal and Questions

The aim of this thesis is to investigate how professional software developers perceive, evaluate, and carry out source code rejuvenation in legacy systems. Source code rejuvenation is characterized as the transition from outdated idioms to contemporary language constructs, a process that frequently occurs within legacy systems where older and newer constructs of the same programming language version coexist. This phenomenon emerges at the intersection of technological evolution and maintenance efforts. The research questions operationalize this goal:

RQ1: What are the motivations that lead software developers to rejuvenate their software?

RQ2: What are the challenges that hinder software developers from rejuvenating their legacy code?

RQ3: What practices do developers follow or propose to deal with the challenges of source code rejuvenation?

These RQs served as sensitizing devices rather than fixed hypotheses. They guided theoretical sampling and data collection while allowing categories and relationships to emerge inductively during the steps of data collection and analysis. To investigate this phenomenon, we adopted a Socio-Technical Grounded Theory (STGT) approach [2]. For data collection, we use two different approaches. First, we conducted semi-structured interviews with developers who have direct experience with legacy systems, modern language features, large-scale refactoring efforts, or automated transformation tools.

Participants were recruited from diverse professional sectors, including private software companies, enterprise corporate environments, banking and financial institutions, public government agencies, and academia, to capture a broad range of rejuvenation experiences. Given their central role in software maintenance, these practitioners provided detailed and practice-oriented insights into the motivations, challenges, and strategies that shape rejuvenation efforts.

Their firsthand reports offered rich descriptions of the factors that influence rejuvenation decisions, including the benefits of modern features (e.g., improve code expressiveness, improve performance, reduce technical debt), organizational constraints (e.g., company restrictions, management approval required, and keep system running and evolutive maintenance are more important than source code rejuvenation), perceived risks (e.g., rejuvenation efforts may introduce bugs, language evolution triggering hidden regressions), and tool support (e.g., automated rejuvenation tools, IDE/compiler assistance).

Second, we conducted mining studies that combine quantitative analyses of the adoption of modern language features in open-source projects with qualitative evidence extracted from survey responses and pull-request discussions. This integrated design allowed us to examine observable usage patterns alongside the perceptions of practitioners in development contexts. Together, the interview data and mining studies formed the empirical foundation for the grounded theory developed in this thesis.

3.2 Socio-Technical Grounded Theory Approach

We selected Socio-Technical Grounded Theory (STGT) as our empirical research method because the phenomenon of source code rejuvenation aligns closely with STGT's socio-technical research framework [2] across its four dimensions:

- **ST phenomenon**: Source code rejuvenation is inherently socio-technical. It involves not only technical transformations—such as replacing deprecated constructs with modern abstractions—but also social dynamics around team practices, developer expertise, and organizational constraints. These intertwined dimensions cannot be fully understood through purely technical or purely social lenses.

- **ST domain and actors**: The domain of this research is software engineering, an intrinsically socio-technical discipline. The actors we studied are professional software developers who routinely make decisions about maintaining, rejuvenating, and evolving code. Their perspectives reveal how technical and organizational factors shape rejuvenation behavior.

- **ST researchers**: The research team includes experienced socio-technical researchers with extensive backgrounds in software development, software evolution, qualitative research methods, and empirical studies of programming language features adoption. This expertise ensured rigorous application of STGT practices and sensitivity to socio-technical interactions.

- **ST data tools and techniques**: We used socio-technical data and analysis methods aligned with STGT principles across two iterative rounds of data collection and analysis. Semi-structured interviews were conducted in the first and third rounds (Round 1 and 3) using Google Meet[1] and Microsoft Teams[2]. All sessions were recorded using OBS Studio[3]. Transcripts, analytical memos, and other qualitative materials were systematically coded using ATLAS.ti[4], which supported structured, iterative refinement of emerging concepts and categories throughout the basic and advanced stages of STGT analysis.

  The analysis of the initial interviews revealed gaps, tensions, and emerging categories that required further empirical exploration beyond the first interview dataset. In particular, the emerging theory required additional evidence to examine how source code rejuvenation manifests across different programming languages, project histories, and socio-technical contexts. Therefore, subsequent rounds of theoretical sampling were conducted to refine, contrast, and expand the emerging theory by integrating complementary forms of empirical evidence.

  In the second round, three empirical software repository mining studies constituted a theoretical sampling round (Round 2) within the broader STGT process.

[1] `https://meet.google.com`
[2] `https://www.microsoft.com/microsoft-teams`
[3] `https://obsproject.com`
[4] `https://atlasti.com`

These studies generated complementary data, including quantitative measurements of the adoption and usage of modern C++, JavaScript, and Python features across projects, as well as qualitatively analyzed survey responses and pull-request discussions. As detailed in the appendices (Study A (C++ MSR) [3], Study B (JavaScript MSR) [4], and Study C (Python MSR) [5]), these mining studies enriched the grounded theory by situating developers' reports from the interviews within observable and measurable patterns of system evolution, enabling the integration of experiential evidence with longitudinal usage trends.

In the third round, an additional set of interviews was conducted as interview-based theoretical sampling (Round 3). This round focused on theoretical gaps and open questions that remained insufficiently covered after the initial interviews and the mining studies. These interviews helped strengthen and expand the emerging theory by providing further practitioner perspectives, supporting the refinement of categories and relationships, and enabling additional comparisons across the evidence collected in Rounds 1 and 2.

In line with STGT guidelines, our formulated research questions (Section 3.1) guide data collection and analysis. Although STGT does not require predefined RQs, such questions help clarify the study's scope, ensure alignment with the socio-technical nature of the phenomenon, and maintain openness to emergent concepts [2, 56]. These broad questions supported the flexible sequencing of the three rounds of data collection: semi-structured interviews for initial theory development (Round 1), theoretical sampling through repository mining studies (Round 2), and interview-based theoretical sampling for further theory refinement and expansion (Round 3).

STGT requires researchers to articulate the underlying research paradigm, as it influences interviewing, interpretation, and theory construction. While traditional GT variants align with specific paradigms (Glaserian with positivism [57], Strauss-Corbinian with symbolic interactionism [58], and Charmaz's with constructivism [59, 60, 61]), STGT offers flexibility in selecting a paradigm that suits the study context. Given the extensive experience of the research group, my professional background in industry, and our sustained engagement with developers and programming language ecosystems undergoing continuous evolution, we adopted a constructivist paradigm to guide this study. This paradigm acknowledges the researcher's active role in interpreting participant experiences, guiding theoretical sensitivity, and collaboratively constructing meaning throughout the analysis process [2, 56].

3.3 Research Design

Figure 3.1 presents a detailed overview of the methodological process adopted in this thesis, depicting the complete socio-technical grounded theory (STGT) workflow from the pilot phase to final theory construction. As illustrated in the figure, the study followed a structured yet iterative trajectory composed of three major rounds, organized across the Basic and Advanced stages of STGT, culminating in theoretical integration and structuring.

The process began in the Basic Stage with a Pilot Round, which included a lean literature review, a pilot study, and the design and refinement of a Pre-Interview Questionnaire (PIQ). This preparatory phase supported the transition to Round 1, also conducted within the Basic Stage, where the refined PIQ guided the conducting of semi-structured interviews using convenience sampling. These interviews constituted the primary data source of the STGT. Data analysis in Round 1 involved open coding, constant comparison, and intensive conceptual memoing. This process led to the emergence of an initial conceptual structure grounded in participants' reports. The findings from Round 1 were consolidated into coherent emerging categories, providing an initial explanatory foundation that continued to evolve throughout the subsequent rounds.

Building on the Round 1 findings, the study advanced to the Advanced Stage with Round 2, which comprised three theoretically motivated large-scale mining studies: Study A (C++ MSR) [3], Study B (JavaScript MSR) [4], and Study C (Python MSR) [5]. These studies were motivated by conceptual gaps, tensions, and explanatory needs identified during Round 1. Each study generated its own set of findings, which were analyzed before initiating the subsequent study.

Within Round 2, data analysis involved targeted coding, constant comparison, conceptual memoing, and explicit checks for theoretical saturation. The mining investigations integrated quantitative evidence (e.g., temporal adoption patterns and large-scale repository indicators) with qualitative evidence (e.g., survey responses and pull-request discussions), enabling theoretical sampling at the level of repositories and feature adoption contexts.

Still within the Advanced Stage, the study proceeded to Round 3, which consisted of a new round of semi-structured interviews conducted as interview-based theoretical sampling. This round was designed to address theoretical gaps and open questions that remained insufficiently covered after the initial interviews and the cross-language mining studies. The Round 3 interviews provided additional practitioner perspectives, supported further comparison with the evidence collected in Rounds 1 and 2, and helped refine, expand, and strengthen the emerging theory. In particular, this final round contributed to clarifying relationships among categories, examining the applicability of the emerging

explanations across different socio-technical contexts, and supporting the assessment of theoretical saturation.

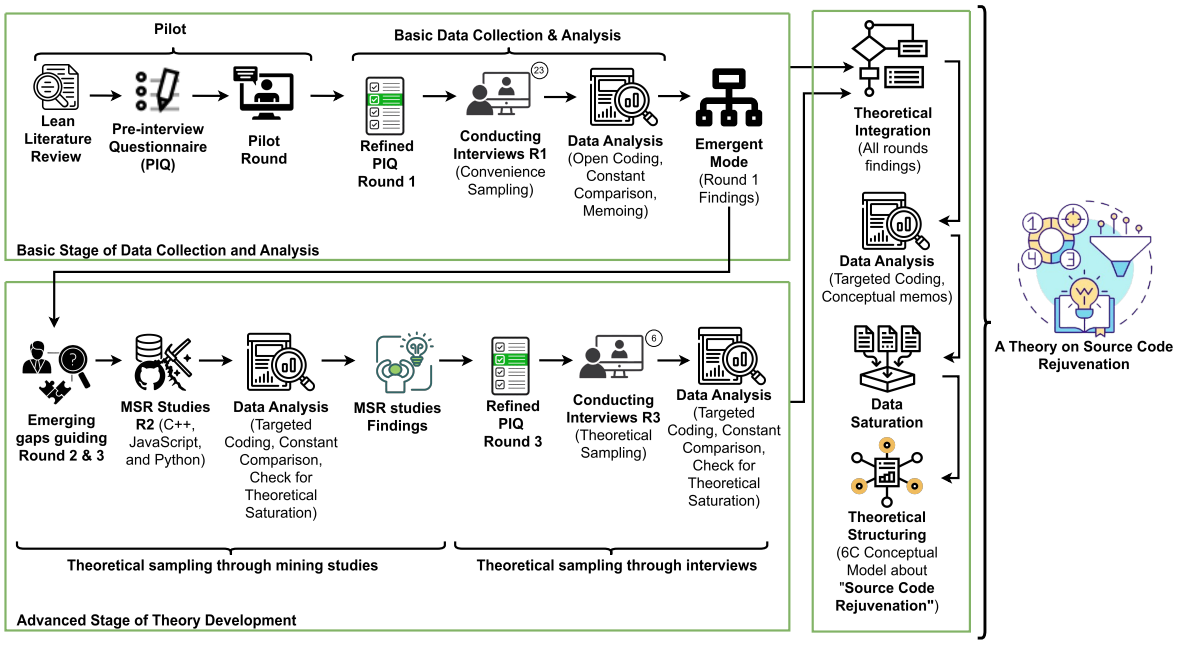


Figure 3.1: Research methodology: Applying mixed methods research using socio-technical grounded theory for data analysis [2] to develop the theory on Source Code Rejuvenation. *MSR: Mining Software Repositories, C++ Study A [3], JavaScript Study B [4], Python Study C [5].

New categories and properties emerging from the theoretical sampling conducted through the mining studies and follow-up semi-structured interviews were incorporated into the evolving conceptual structure and systematically compared with the concepts derived from Round 1 interviews. Saturation was assessed at the conceptual level (data saturation) when additional evidence no longer introduced substantively new properties or dimensions to the categories but instead only reinforced their explanatory robustness.

Finally, findings from all rounds were consolidated through a phase of Theoretical Integration, followed by Theoretical Structuring using the six C's conceptual model [6] (Contexts, Conditions, Causes, Covariances, Consequences, and Contingencies). This integrative phase articulated the relationships among categories and resulted in a coherent socio-technical grounded theory of source code rejuvenation. Collectively, the two rounds illustrated in Figure 3.1 demonstrate a continuous, multi-source, and theoretically integrated STGT process. The Round 1 is detailed in Section 3.4, and Round 2 and 3 in Section 3.5.

3.4 Basic Stage of Data Collection and Anlysis

In this study, the methodological process is initiated with a focused (lean) literature review aimed at identifying preliminary research gaps, terminological inconsistencies, and conceptual boundaries surrounding source code rejuvenation, in alignment with STGT methodological guidance [2]. This review, presented in Chapter 1, provided an initial contextual understanding of the phenomenon and informed the design of a pre-interview questionnaire and an initial interview guide, without constraining the emergence of the theory from the data.

In Round 1, participants were recruited through targeted outreach based on professional recommendations (e.g., referrals from senior researchers and practitioners with knowledge of relevant migration experiences). Invitation emails briefly introduced the study goals, the institutional affiliation, and the expected interview duration (20–30 minutes), and proposed scheduling according to each participant's availability.

All interviews were conducted online and followed a semi-structured protocol [5]. At the beginning of each session, the interviewer reiterated the purpose of the study, clarified participation conditions, and obtained explicit permission to record the audio for research purposes before proceeding. This design ensured ethical compliance while preserving the flexibility required to probe emerging concepts and collect rich socio-technical informations.

At the beginning of each interview, the researcher introduced the purpose of the study and the institutional affiliation, and explicitly requested permission to record the audio for academic research purposes only. Participants were asked to verbally confirm their consent before the interview proceeded.

The interview protocol was organized into four main groups of questions. First, initial questions captured contextual and background information, including current programming language(s), years of professional experience, current language version(s) in use, and the participant's job position and relationship with the codebase. Second, questions addressed developers' motivations for evolving source code, covering perceptions of language evolution, reasons for adopting modern language features, examples of adopted constructs, perceived benefits, update processes and tool support, and factors that may hinder rejuvenation. Third, questions explored challenges associated with evolving source code, including main technical and organizational difficulties, practices adopted to stay up-to-date with programming languages evolution, and perceived difficulty in keeping pace with new language versions. Finally, the interview addressed tool support for code evolution, including the use of automated tools, and issues experienced during tool usage,

[5]Interviews protocol questionnaire: `https://github.com/waltim/socio-technical-grounded-theory-scr/blob/main/interviews-protocol-questionnaire.pdf`

along with suggestions for improvement. The complete interview guide is available in the replication package of this thesis [7].

3.4.1 Recruitment

Data collection began with convenience sampling, followed by snowball sampling as the study progressed. As shown in Box 3.4.1, the invitation email was used to contact potential participants and introduce the study. Our target population consisted of professional software developers with multiple years of industry experience (minimum = 4 years; median = 9 years; mean = 11.8 years), ensuring that participants had substantial exposure to language evolution, legacy code constraints, and software maintenance efforts. Initially, we attempted to recruit contributors from well-known open-source repositories on GitHub; however, this approach produced a low response rate. We then expanded recruitment to professional networks, including LinkedIn, personal contacts, and referrals from participants.

Invitation email (excerpt)

Subject: Invitation to participate in an online interview
To: [Participant Candidate]

Dear [Participant Candidate],
My name is Walter Lucas, I am a student at the University of Brasília (UnB – Brasília, Brazil).

I am contacting you through the recommendation of [a previous participant / a colleague from our network / a collaborator from our research group]. I am conducting a research project to better understand how software systems and libraries evolve in response to the evolution of programming languages.

We are particularly interested in the factors that may lead experienced developers such as yourself to update open-source systems using modern language features.

To gather these insights, we would like to invite you to participate in a brief 20–30-minute online interview to discuss software evolution practices, to be scheduled according to your availability.

Sincerely,
Walter Lucas (University of Brasília – UnB)

During Round 1, 23 semi-structured interviews (P01-23) were conducted, with an average duration of 32 minutes (range: 11 to 76 minutes). We held the interviews online using Google Meet and Microsoft Teams, and recorded all sessions with OBS Studio. Table D.1 presents participant demographics, including years of professional experience, professional roles, and organizational sectors. The Round 1 interviews generated approximately 100,000 words of transcript data, providing the empirical foundation for our initial open coding, constant comparison, and memoing.

The participant pool comprised professionals with diverse roles, including senior developers, software architects, tech leads, founders, consultants, researchers, and professors, working across multiple organizational sectors such as enterprise software, public institutions, banking, academia, and software tooling. Their experience ranged from 4 to 30 years (median = 9 years), ensuring exposure to multiple language evolution cycles and rejuvenation initiatives. This heterogeneity strengthened the constructivist STGT process by enabling theoretical variation, supporting constant comparison across distinct technical and organizational contexts, and enhancing the robustness and explanatory power of the emergent theory.

3.4.2 Data Analysis

The data collection and analysis progressed iteratively, interleaving coding activities with the gradual refinement of the interview guides and the design of the theoretical sampling strategy. After the completion of the first round of interviews (Round 1), the thesis author transcribed all recordings and imported the transcripts into ATLAS.ti for analysis. The thesis author conducted open coding in pairs with an experienced researcher, holding regular coding sessions grounded in constant comparison. During these sessions, both researchers compared interpretations, discussed coding decisions, examined disagreements, and sought analytical agreement on the meaning and scope of codes, categories, and emerging concepts. The sessions were recorded so that they could be revisited later, supporting the refinement of codes, the consolidation of categories, and the clarification of conceptual relationships throughout the analysis.

The analytical process also benefited from the collaboration of four experienced researchers with complementary expertise in software development, software evolution, qualitative research methods, empirical studies on the adoption of programming language features, and statistics. Through iterative peer discussions, these researchers reviewed and

refined the emerging theory [7], contributing to the analytical rigor and theoretical robustness of the study. The early codes captured developers' motivations for rejuvenating code, the constraints that hinder the adoption of modern features, and detailed descriptions of concrete rejuvenation episodes.

Figure 3.2 illustrates this analytic progression from raw data to increasingly abstract analytic units. Selected interview excerpts (left) were first coded to capture (i) perceived readability and maintainability gains associated with language evolution (e.g., improved expressiveness and improved understanding), and (ii) constraints that hinder rejuvenation efforts (e.g., framework/library dependencies and poor internal code quality). Through constant comparison, these codes were grouped into three concepts—*Code Readability and Expressiveness*, *Technical Debt & System Quality Constraints*, and *External Dependencies & Environment Constraints*. Further abstraction connected these concepts to two categories (right): *Code Quality Improvements*, and *Challenges & Impediments*, exemplifying how early coding and memoing support the construction of theoretically meaningful structures in STGT.

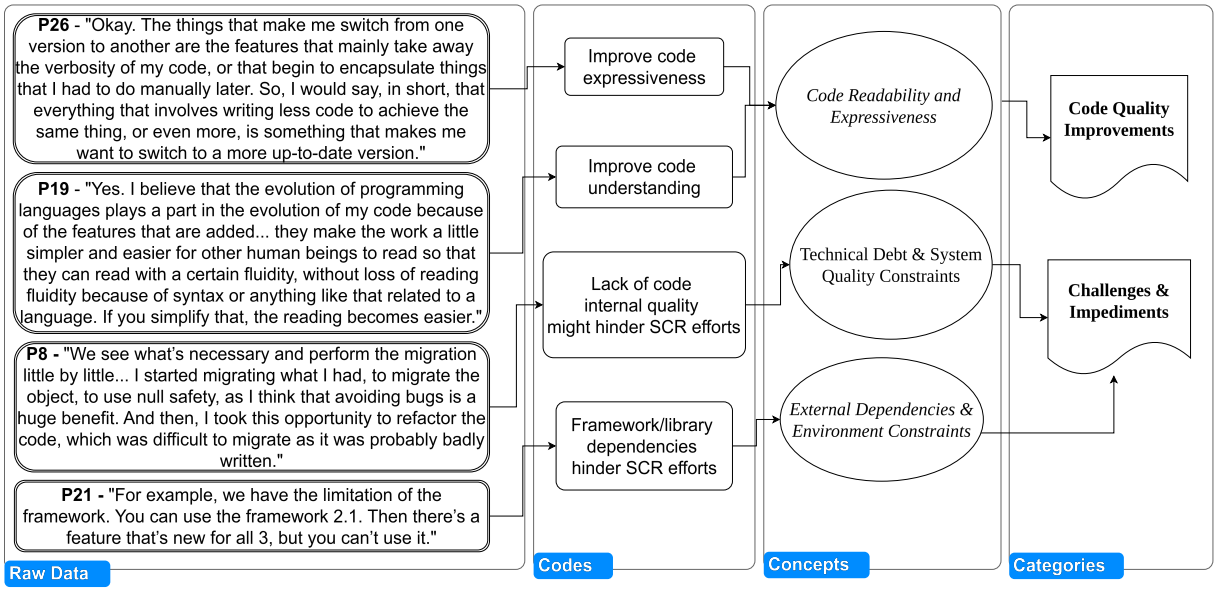


Figure 3.2: Example of the STGT coding progression, showing how raw data excerpts (left) were transformed into codes, grouped into concepts (centre), and abstracted into categories (right) through constant comparison.

3.4.3 Memoing and Early Theoretical Structuring

Still during the Round 1, analytic memos were written to capture emerging interpretations, methodological decisions, and early conjectures about relationships across codes and concepts. These early memos laid the groundwork for the construction of prelimi-

nary theoretical insights, illustrating how initial raw data excerpts were transformed into codes, concepts, and early theoretical propositions.

The example shown in Box 3.4.3 is representative of this early-stage theorisation. It illustrates the interpretive work conducted during Round 1 and subsequently refined through iterative constant comparison as new data were incorporated into the analysis. Throughout this process, we held extensive group discussions to consolidate interpretations, challenge emerging assumptions, and negotiate conceptual boundaries. These collective analytic sessions were central to achieving a more robust and shared understanding of the emerging categories, strengthening their conceptual clarity and theoretical coherence.

These early relationships suggest that rejuvenation is not merely a technical upgrade decision but a socio-technical judgement about sustaining internal code quality over time. Developers frame rejuvenation as a strategy to enhance clarity, expressiveness, and long-term maintainability, thereby reducing cognitive load for newcomers and supporting collective code comprehension.

#Memo on "Code Readability and Expressiveness"

Across multiple interviews, participants consistently framed language evolution and source code rejuvenation not merely as technical updates, but as mechanisms for improving internal code quality through enhanced clarity, expressiveness, and maintainability. A recurring theme is the intentional pursuit of readability as a long-term investment in comprehension and onboarding.

P19 explicitly associates language evolution with improved reading fluidity and simplification, emphasizing that newer constructs reduce syntactic friction and make code easier for "other human beings to read". This aligns directly with the codes *Improve code expressiveness* and *Improve code understanding*. The participant frames rejuvenation as a way to make "new developers... more comfortable maintaining that code", which strongly reflects *Reducing cognitive load for newcomers*. The focus is not just brevity, but interpretability over time.

Similarly, P4 describes adopting lambda expressions in .NET due to increased "productivity and legibility," emphasizing that simplified constructs make it easier "for you to understand it" and for someone else updating the code to grasp what is happening. Here, improved expressiveness directly translates into enhanced maintainability and onboarding efficiency.

P20 articulates a comparable evaluation process when deciding whether to adopt newer constructs: "if it gets simpler, safer or faster, you adopt it". Importantly, simplicity is treated as a primary pillar. P20 also notes that improvements such as

Range-based for loops (from C++ 11) make code "much simpler" and more direct. This suggests that internal quality is operationalized as reduction of unnecessary verbosity and prevention of misinterpretation (e.g., incorrect index usage), reinforcing the code *Improve code expressiveness*.
There is also evidence of an implicit *Improve code standardization* dynamic. When P20 describes systematically replacing recurring patterns with clearer constructs over time, it suggests that rejuvenation can homogenize style and modern idioms across a codebase, reinforcing internal coherence ...

Across iterative cycles of coding, constant comparison, memoing, and sustained analytic discussions, the research team progressively consolidated a coherent set of emerging explanatory categories linking motivations, constraints, risk perceptions, and execution strategies surrounding source code rejuvenation. By the conclusion of Round 1, these categories had achieved increasing conceptual density and relational articulation, revealing patterned interactions among technical incentives, organizational pressures, and perceived trade-offs. Following the emergent mode of STGT, this level of theoretical maturity motivated the transition from the Basic Stage to the Advanced Stage, in which subsequent data collection and analysis were guided by the explanatory needs of the emerging theory.

Within the Advanced Stage, Round 2 extended the analysis through theoretically motivated repository mining studies designed to elaborate, refine, and challenge the developing categories while preserving their grounded foundations. Subsequently, Round 3 introduced a new round of semi-structured interviews conducted as interview-based theoretical sampling. This final round addressed theoretical gaps and open questions that remained after Rounds 1 and 2, providing additional practitioner perspectives to further refine category properties, clarify relationships among concepts, and support theoretical saturation.

3.5 Advanced Stage of Theory Development

Although Round 1 produced conceptually dense categories, it also revealed analytical tensions and empirical gaps that could not be fully addressed through interview data alone. Developers consistently framed rejuvenation as a practice for *improving internal code quality*, such as enhancing *expressiveness*, *code comprehension*, and *maintainability*, yet the interviews did not provide longitudinal insight into how or when such improvements materialize at scale. It remained unclear whether these perceived benefits translated into discernible adoption patterns across systems, or whether rejuvenation efforts remained localized and incremental.

Similarly, factors related to *unexpected behavioral changes* that may render software unstable or inoperable raised an unresolved question: were these risks episodic perceptions tied to situated experiences, or did they reflect more systematic patterns observable during rejuvenation efforts? Without large-scale historical data, it was not possible to determine whether risk perception functioned primarily as a cognitive barrier or as an expression of recurring instability patterns associated with changes introduced during rejuvenation activities.

In addition, Round 1 also exposed an emerging gap related to the role of LLM-based tools in software development and source code rejuvenation. While the interviews captured developers' perceptions about the selective, risk-aware, and human-controlled use of traditional tools, recent studies [62, 63, 64, 65, 66] suggested that LLMs introduce a contemporary form of support. However, the first round of interview data alone were insufficient to explain how such support is actually used in rejuvenation efforts, how developers validate LLM-generated changes, and how risks related to correctness and confidentiality are managed in practice. However, the first round of interview data alone was insufficient to explain how such support is actually used in rejuvenation efforts, how developers validate LLM-generated changes, and how risks related to correctness and confidentiality are managed in practice.

These unresolved issues motivated the design of the theoretical sampling strategies conducted in the Advanced Stage. In Round 2, rather than merely seeking contrasting perspectives, the study operationalized triangulation by integrating large-scale longitudinal evidence from C++, JavaScript, and Python legacy systems. Across the three mining studies, the analyses quantitatively examined time-series trends in the adoption of modern programming language features to estimate when their use began, how long adoption took to emerge more broadly, and how adoption evolved across project histories. These longitudinal patterns were then related to qualitative evidence from the previous interviews, survey responses, and pull-request discussions, enabling the study to connect developers' perceptions with the motivations, constraints, and socio-technical conditions associated with the observed adoption trends. In addition, the C++ and JavaScript studies identified commits that characterized substantial source code rejuvenation efforts, allowing the analysis to examine concrete episodes in which systems adopted modern language features through observable code changes.

Building on the gaps and open questions that remained after Rounds 1 and 2, Round 3 introduced a new set of semi-structured interviews conducted as interview-based theoretical sampling. This round expanded the empirical basis of the theory by focusing on industry developers who had experienced source code rejuvenation and software migration efforts in practice. The interviews further explored issues that remained insufficiently

understood from the previous rounds, including developers' perceptions of rejuvenation risks, decision-making processes, practical constraints, and the role of tool support. In particular, Round 3 introduced a new question about the use of LLMs as potential support for source code rejuvenation and migration activities, directly addressing the emerging gap concerning contemporary AI-based assistance. This allowed the study to examine how developers perceive LLMs as tools that may support, constrain, or reshape rejuvenation efforts, while also capturing concerns related to correctness, confidentiality, overreliance, and the continued need for human review and contextual validation.

Through this integration, Rounds 2 and 3 did not simply confirm the categories derived from Round 1; they provided further grounding to address the analytical tensions and empirical gaps that remained after the initial interviews. Round 2 addressed these gaps by embedding developers' experiential reports within observable longitudinal patterns of feature adoption across C++, JavaScript, and Python systems. The mining studies revealed delays between feature release and generalized adoption, differentiated superficial presence from broader structural integration, identified early experimentation prior to stable releases, and exposed the influence of external contingencies such as end-of-life policies and framework migrations [3, 4, 5]. Round 3 complemented this evidence through interview-based theoretical sampling with industry developers who had experienced source code rejuvenation and software migration efforts, allowing the analysis to revisit unresolved questions, expand practitioner perspectives, and further refine the relationships among motivations, perceived risks, constraints, tooling support, and observed rejuvenation practices. By integrating longitudinal mining evidence with additional practitioner accounts, the theoretical model evolved from an initial interpretive understanding of developer reasoning into a broader socio-technical explanation of how, when, and under which conditions source code rejuvenation in the analyzed legacy systems.

3.5.1 Theoretically Grounded Cross-Language Sampling and Triangulation

Beyond their methodological role in theoretical sampling and triangulation, these languages were intentionally selected due to their relevance and influence within contemporary software development contexts in which source code rejuvenation occurs. Independent industry reports and large-scale developer surveys, including GitHub Octoverse[6] , the Stack Overflow Developer Survey[7] , the IEEE Spectrum Top Programming Languages

[6] `https://octoverse.github.com/`
[7] `https://survey.stackoverflow.co/`

ranking[8] , and the TIOBE Index[9] , have recurrently positioned JavaScript, Python, and C++ among the most widely used languages worldwide over the past decade.

Furthermore, prior empirical studies have already explored related aspects of language evolution and feature adoption in other major languages, such as Java [11, 13, 19, 25, 67, 68]. By contrast, C++, JavaScript, and Python offer complementary perspectives characterized by different compatibility policies, release cadences, and community practices, thereby enabling broader theoretical transferability while avoiding redundancy with existing Java-centered research.

The following subsections summarize the methodological design and findings of each mining study, emphasizing how large-scale longitudinal analyses and qualitative evidence were deliberately integrated as a theoretically driven extension of sampling. This integration served to triangulate experiential insights from Round 1, further explore the explanatory scope of emerging categories, and refine the developing socio-technical theory of source code rejuvenation.

3.5.2 Round 2 – Cross-Language Mining Studies

To address the analytical tensions identified in Round 1, three large-scale mining studies were conducted following a common methodological design across different programming language, such as C++, JavaScript, and Python. In all three cases, the quantitative component consisted of mining the longitudinal evolution history of curated sets of mature open-source systems to extract structured signals of modern language feature usage and diffusion over time. This design enabled the systematic observation of adoption onset, growth regimes, version gaps, and generational acceleration effects.

The C++ study analyzed 272 open-source systems from the KDE community, focusing on the diffusion of modern C++ features (e.g., C++11 and later standards). The JavaScript study examined 158 long-lived repositories, combining historical feature usage mining with large-scale pull-request data (122,248 PR comments, of which 447 were manually analyzed). The Python study extended this design to 424 mature repositories, mining feature usage over a multi-year period and collecting 395,702 pull-request comments, from which 494 were manually examined after staged filtering.

Methodologically, the three studies shared a core quantitative strategy: extracting occurrences of modern features usage from repository histories to model rejuvenation dynamics as observable temporal processes rather than isolated change events. In all cases, the mining component enabled the identification of heterogeneous adoption trajectories,

[8] `https://spectrum.ieee.org/top-programming-languages`
[9] `https://www.tiobe.com/tiobe-index/`

measurable delays between feature release and generalized use, and differences between superficial presence and structural integration of modern constructs.

The qualitative complement differed slightly across programming languages. In the C++ study, practitioner intent and rationale were captured through a targeted survey involving 11 developers engaged in identified rejuvenation efforts. In contrast, the JavaScript and Python studies derived qualitative evidence from naturally occurring pull-request discussions, applying staged filtering and manual thematic analysis to surface motivations, constraints, and trade-offs expressed during rejuvenation efforts. While the source of qualitative evidence varied, its analytical role remained consistent: to connect longitudinal adoption signals with developers' articulated reasoning.

Together, these studies operationalized triangulation as a theoretically driven extension of sampling. Quantitative mining made rejuvenation dynamics empirically observable, revealing systematic delays, early experimentation prior to stable releases, and acceleration effects associated with language maturity and external pressures. The qualitative layers, whether survey-based or review-based, clarified how developers justified rejuvenation in terms of code comprehension, maintainability, safety, performance, compatibility constraints, maintenance costs, and organizational considerations.

Rather than replacing the categories emerging from Round 1, the cross-language mining studies strengthened, and delimited them. They demonstrated that source code rejuvenation unfolds as a gradual socio-technical diffusion process. By embedding experiential evidence within large-scale temporal patterns, Round 2 refined the emerging theory into a cross-context explanatory model of how, when, and under which conditions source code rejuvenation unfolds in legacy systems.

Detailed methodological descriptions and full empirical results are provided in the Appendix and in the corresponding publications: Study A (C++ MSR) [3], Study B (JavaScript MSR) [4], and Study C (Python MSR) [5].

3.5.3 Round 3 – Interview-Based Theoretical Sampling

Round 3 was conducted as interview-based theoretical sampling within the Advanced Stage of the STGT process. This round was designed to address gaps and unresolved questions that remained after Rounds 1 and 2, particularly issues that required additional practitioner accounts rather than further repository-level evidence. In addition, Round 3 expanded the inquiry to contemporary forms of tool support, especially the use of LLMs in software development, rejuvenation, and migration activities. Since the earlier data collection rounds could not capture these recent practices in sufficient detail, the previous semi-structured interview guide was extended with a specific question about the use of LLM-based tools as support for these activities.

In total, Round 3 comprised six new interviews (P24–P29), whose participant details are presented in Table D.1. The participants had between 6 and 14 years of professional experience and worked as developers, senior developers, or technical leads in the enterprise software sector. They were selected because they worked in industrial software factories that had begun introducing LLM-based tools to support software development activities. Two participants came from the thesis author's professional network and worked in two distinct companies, while the remaining participants were recruited through snowballing based on invitations from these initial participants. This sampling strategy enabled the study to expand the empirical basis of the theory with developers who had practical experience with software rejuvenation, migration efforts, and emerging AI-assisted development practices.

The interviews were collected, transcribed, and analyzed sequentially, following the iterative logic recommended by STGT. The thesis author transcribed the interviews and conducted targeted coding, constant comparison, and checks for theoretical saturation. The analysis focused on comparing the new evidence with the categories, concepts, and relationships that had already emerged from Rounds 1 and 2, while also identifying new properties related to tool support and LLM-assisted practices. Finally, another researcher reviewed the codings and memos through analytical discussions with the thesis author until reaching agreement and concluding the analysis. This process supported the refinement of existing categories, the expansion of underdeveloped concepts, and the integration of Round 3 evidence into the evolving theoretical structure.

3.5.4 Theoretical Saturation and Final Structuring

In accordance with STGT guidelines, theoretical saturation is achieved when additional data collection "does not generate new or significantly add to existing concepts, categories or insights" [2]. In this study, Rounds 2 and 3 constituted theoretically driven extensions of sampling aimed at interrogating, refining, expanding, and strengthening the categories that had emerged from Round 1. While Round 1 produced the initial conceptual structure, it also revealed gaps and unresolved questions that required additional empirical evidence. Round 2 addressed these gaps through cross-language repository mining studies, while Round 3 extended the analysis through interview-based theoretical sampling focused on additional practitioner perspectives and contemporary tool support through LLMs.

Following the completion of the three mining studies (C++, JavaScript, and Python), all findings were reanalyzed in conjunction with the Round 1 interview data through focused coding, constant comparison, and integrative memoing. The mining studies addressed Round 1 gaps by combining quantitative time-series analyses of modern feature adoption trends with qualitative evidence from surveys and pull-request discussions.

These analyses helped estimate when the use of modern features began, how long adoption took to emerge more broadly, and how observed adoption trajectories related to developers' motivations, constraints, and perceptions. This cross-context integration did not lead to the removal or replacement of categories established in Round 1. Instead, it refined their boundaries, strengthened their relational connections, and expanded their explanatory scope across distinct programming languages, project histories, and empirical scales.

Round 3 further expanded and refined the emerging theory through a new set of semi-structured interviews conducted as interview-based theoretical sampling. This round addressed remaining gaps and open questions from Rounds 1 and 2, particularly those requiring additional practitioner accounts about decision-making, risks, practical constraints, and tool support. It also introduced LLM-based support as a contemporary extension of the investigation, enabling the analysis to examine how developers perceive the use of AI-based tools in source code rejuvenation and software migration efforts. Although Rounds 2 and 3 generated new categories, properties, and empirical illustrations, the final interviews, particularly P28 and P29, indicated that the central concepts and relationships had become sufficiently robust. The additional evidence mainly reinforced patterns already observed across the three rounds rather than substantially altering the emerging theoretical structure. Therefore, the thesis author and the two experienced researchers in the group agreed that theoretical saturation had been achieved and that no further data collection was necessary.

Following saturation, the study proceeded in Structured Mode, as illustrated in Figure 3.1. This involved (i) targeted data analysis through focused coding and conceptual memoing, (ii) confirmation of data saturation, and (iii) consolidation of categories into a coherent theoretical structure.

Finally, the mature theory is presented in Chapter 4 using the six C's template (Contexts, Conditions, Causes, Covariances, Consequences, and Contingencies) for structured articulation. This presentation formalizes the relationships consolidated during our two rounds while preserving their empirical grounding in both interview-based and large-scale repository evidence.

Chapter 4

A Theory About Source Code Rejuvenation

After integrating all rounds and confirming theoretical saturation, the emerging theory was consolidated into a coherent explanatory structure. To articulate the relationships among the identified categories, the theory is presented using Glaser's six C's template—Contexts, Conditions, Causes, Covariances, Consequences, and Contingencies [2, 6]. The Six C's coding family is commonly employed in STGT studies [69, 70] within software engineering to structure and relate emergent theoretical categories. By this stage, the study had already produced a set of core categories and concepts grounded in interview data and repository-mining studies. Rather than serving as a mechanism for generating categories, the six C's template provides a structured lens to explain how these categories interact within a socio-technical explanation of source code rejuvenation.

In our study, *Source Code Rejuvenation* emerged as the central phenomenon explaining how developers respond to programming language evolution while maintaining and evolving long-lived software systems. The analysis revealed that Source Code Rejuvenation is not a uniform or purely technical activity. Instead, it unfolds as a socio-technical process shaped by interacting technical constraints, organizational dynamics, developer knowledge, and the continuous evolution of programming languages and their surrounding tools and libraries.

These categories collectively explain the dynamics of source code rejuvenation. Among them, `Source Code Rejuvenation` in the Context of Programming Language Evolution emerged as the central category organizing the relationships among the others. The remaining categories—`Drivers for Source Code Rejuvenation`, `Challenges & Impediments`, `Disablers`, `Conditionally Factors`, `Enablers`, `Code Quality Improvements`, `Development Efficiency Gains`, `Execution and Operational Strategies`, `Human and Social Strategies`, `Strategic Decision-Making for`

`Rejuvenation`, `Handling Technical and System Constraints`, `Automating Validation through CI/CD Pipelines` and `Tooling Support for Source Code Rejuvenation`—capture the conditions, mechanisms, and outcomes that shape how developers negotiate Source Code Rejuvenation decisions in evolving programming language ecosystems.

Figure 4.1 shows how the categories that came up during the analysis fit into Glaser's six C's framework. The figure shows how the new ideas fit together in the theory by showing how contexts, conditions, causes, covariances, contingencies, and consequences all work together to explain the dynamics of source code rejuvenation. The next points give a short summary of the main theoretical roles shown in the figure. More specifically, the six C's helped us to:

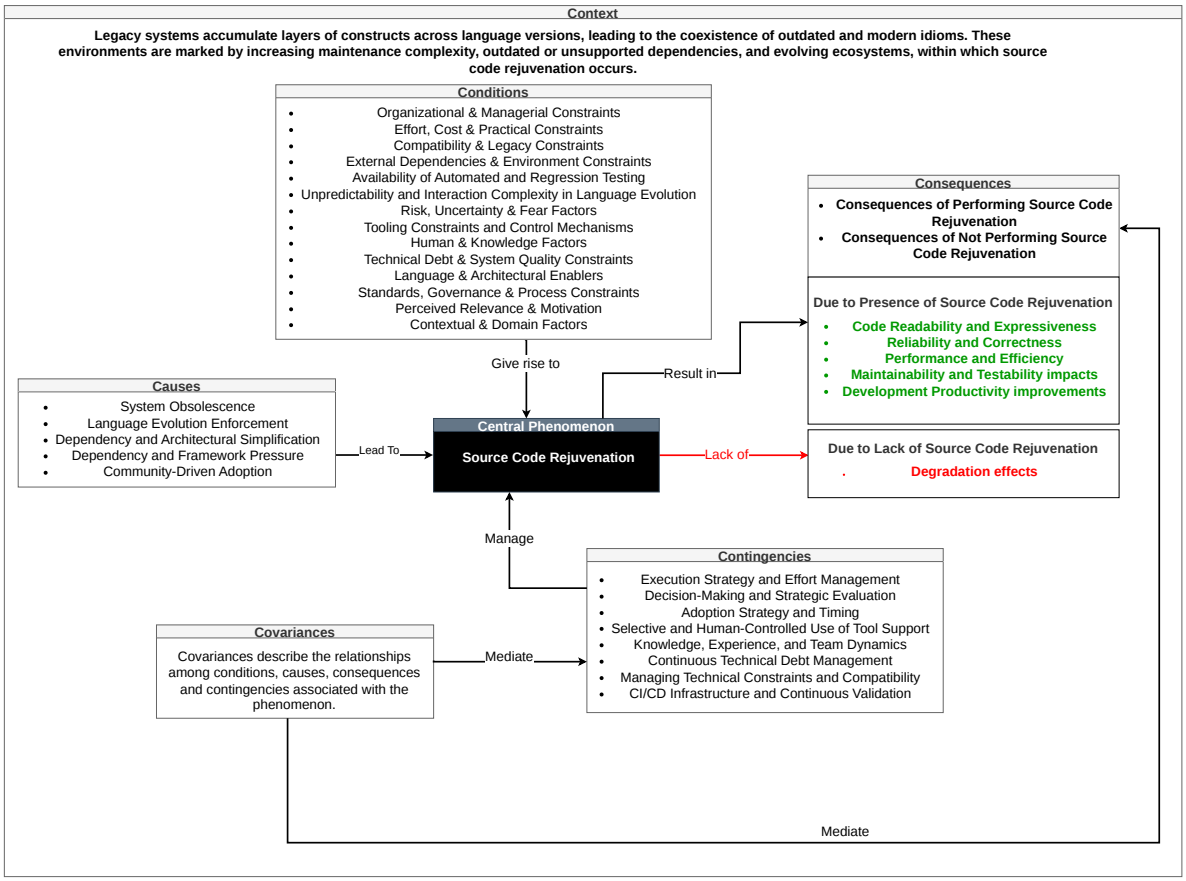


Figure 4.1: Mapping of the emergent concepts to Glaser's Six C's framework [6], showing how contexts, conditions, causes, covariances, contingencies, and consequences collectively explain the dynamics of source code rejuvenation. Consequences are divided into two groups: green denotes consequences of performing source code rejuvenation, whereas red denotes consequences of not performing source code rejuvenation.

- Identify the **contexts** in which source code rejuvenation occurs, i.e., the recurring technical and environmental conditions (e.g., legacy systems, systems with outdated

dependencies, systems depending on a version that is not supported anymore, and increasing maintenance complexity) that signal when source code rejuvenation becomes necessary or beneficial.

- Describe the **conditions** which explains factors that are prerequisites (enablers and disablers) for source code rejuvenation to manifest and alters the central phenomenon in some way;
- Characterize the **causes** that trigger source code rejuvenation efforts, such as framework, libraries, and language evolution pressures, language version support discontinuity, and system obsolescence;
- Articulate the **consequences** of source code rejuvenation efforts, particularly improvements in internal code quality such as readability, maintainability, performance, security, and the lack thereof;
- Identify the **contingencies** representing the strategies and mechanisms that developers use to perform source code rejuvenation efforts, including incremental source code rejuvenation, manual transformations, team dynamics, and strategic evaluation.
- Explain the **covariances** which represent the relationships between these concepts.

This integration strengthens the explanatory power of the emerging theory and ensures conceptual completeness consistent with the advanced stage of grounded theory development. The following sections present each of the 6Cs and explain how their interactions collectively shape the dynamics of Source Code Rejuvenation in legacy and evolving software systems. To improve clarity and support the interpretation of evidence, we use distinct icons to differentiate data sources in the presented excerpts.

The icon denotes quotes from semi-structured interviews, representing in-depth accounts of participants' experiences. The icon identifies excerpts from pull request discussion comments, capturing situated, interactional reasoning among developers. The icon indicates responses obtained through email-based surveys in the C++ study, reflecting structured, self-reported perceptions. The icon represents evidence derived from quantitative mining analyses, highlighting patterns and observations extracted from large-scale repository data.

In addition, the icon denotes grounded theory concepts, representing higher-level abstractions that group related codes within the six C's analytical structure. When concepts are accompanied by a circled superscript, it indicates the total frequency of occurrence of

the set of codes related to the concept from the dataset, providing an indication of its relative prominence. See Appendix E for a detailed hierarchical summary. The 🏷 icon is used to highlight empirical codes extracted during qualitative analysis. Square brackets within quotes (i.e., [ ]) indicate author-added comments intended to clarify or contextualize the original statements, supporting readers in better understanding the information and how the analyses were interpreted. Finally, within each Six C category, concepts are presented in descending order of frequency, from the most to the least frequently occurring.

4.1 Context

Context in the Six C's template refers to the setting in which the phenomenon takes place and becomes observable. In this study, context concerns the settings in which source code rejuvenation occurs and is implemented by developers. Source Code Rejuvenation tends to appear in software development environments involving long-lived systems, legacy constructs, outdated language versions, and codebases that coexist with evolving programming language standards. Across the quotes, developers describe settings in which older codebases are harder to maintain, understand, and extend. These contexts also include systems maintained under aging language versions, unsupported dependencies, or technologies close to end-of-life, which define the technical environment in which rejuvenation decisions are considered.

In addition, these environments are often characterized by accumulated technical debt, tightly coupled architectures, and reliance on outdated frameworks or libraries, which shape how systems can be maintained and evolved. Developers describe settings where performance constraints and increasing maintenance effort are part of the everyday development landscape. Overall, the quotes indicate that source code rejuvenation occurs in contexts marked by technical stagnation, ongoing system evolution, and growing maintenance demands, where the existing structure of the system constrains and influences development activities.

The contexts described above in which source code rejuvenation occurs were derived from a qualitative analysis of 735 quotes, coded using 74 distinct codes in the full Codebook [7]. By examining recurring patterns across these quotes, we identified key contextual conditions that signal when source code rejuvenation efforts are triggered or become necessary. Table 4.1 summarizes these contexts along with their frequency of occurrence in the dataset.

The identification and quantification of contexts in which source code rejuvenation occurs were conducted through a multi-step approach combining qualitative coding, automated processing, and lexical triangulation. During the coding process and the de-

Table 4.1: Contexts in which source code rejuvenation occurs, identified through qualitative analysis of 463 distinct quotes from the Full Codebook [7]. The Full Codebook contains 748 coded quote instances, since the same quote may be associated with multiple codes; for this lexical analysis, duplicate quote occurrences were removed and only unique quotes were considered. Contexts were derived by grouping recurring characteristics of the socio-technical settings described in the quotes, such as legacy systems, maintenance difficulties, dependency constraints, and unsupported technologies. Frequencies represent the number of distinct quotes in which each context was observed. Contexts are non-mutually exclusive.

Context	**Occurrences**
Maintainability / complexity issues	108
Dependency / library constraints	105
Legacy systems / outdated versions	68
End-of-life / unsupported technologies	12

velopment of the emergent theory, contexts were inductively identified through manual qualitative analysis that resulting in the Full Codebook, in which quotations were systematically examined and coded. Next, were then performed an automated analysis using Python scripts (leveraging libraries such as `pandas`, `scikit-learn`, and `re`) to extract, normalize, and deduplicate quotations, enabling the computation of unique quote occurrences per context.

Subsequently, we conducted a lexical analysis using term-frequency techniques (via `CountVectorizer`) and pattern matching to examine the extent to which the qualitatively identified contexts were explicitly reflected in the textual data. Rather than serving as a validation of the qualitative findings, this step was used as a triangulation mechanism to assess the alignment between semantic interpretation and observable lexical signals.

For example, several contexts were explicitly expressed by developers through direct lexical cues, such as references to *old version* and *obsolete*, which clearly indicate concerns related to outdated technologies and the need for rejuvenation. Similarly, expressions such as *technical debt* explicitly signal maintainability issues, while terms like *dependency* point to dependency-related constraints. These explicit terms provide observable indicators of the underlying contexts identified during qualitative analysis.

Following the automated and lexical analyses, we performed an additional manual verification step to reassess the classified quotations and ensure that the identified contexts remained consistent and theoretically sound. The lexical analysis achieved an average coverage of approximately 0.71, indicating that keyword-based detection provides a reasonably reliable proxy for identifying contexts, while reinforcing the necessity of qualitative analysis to capture implicitly expressed evidence.

Overall, the context of this study is characterized by industrial and open-source software development environments involving long-lived systems that rely on legacy constructs, outdated language versions, and evolving programming language ecosystems such as C++, JavaScript, and Python. Across these settings, source code rejuvenation becomes observable in situations marked by increasing maintenance complexity, dependency constraints, end-of-life or unsupported technologies, performance limitations, security concerns, and reduced developer productivity. This contextual diversity, grounded in recurring patterns observed across the data, provides the socio-technical setting for understanding the conditions under which source code rejuvenation is considered, discussed, and performed in practice.

4.2 Conditions

Conditions in this study describe the factors that enable, constrain, or hinder the application of Source Code Rejuvenation in software systems. These conditions represent the technical, organizational, and environmental forces that directly influence whether, when, and how rejuvenation efforts can be effectively carried out. We identified several concepts, including 💡 `Organizational & Managerial Constraints`(48), 💡 `Effort, Cost & Practical Constraints`(38), 💡 `Compatibility & Legacy Constraints`(35), 💡 `External Dependencies & Environment Constraints`(24), 💡 `Availability of Automated and Regression Testing`(23), 💡 `Risk, Uncertainty & Fear Factors`(20), 💡 `Unpredictability and Interaction Complexity in Language Evolution`(20), 💡 `Tooling Constraints and Control Mechanisms`(20), 💡 `Human & Knowledge Factors`(18), 💡 `Technical Debt & System Quality Constraints`(11), 💡 `Standards, Governance & Process Constraints`(5), 💡 `Language & Architectural Enablers`(5), 💡 `Perceived Relevance & Motivation`(3), 💡 `Contextual & Domain Factors`(2). The following subsections provide a detailed description of each concept.

4.2.1 Organizational & Managerial Constraints

Organizational and Managerial Constraints captures the organizational conditions that constrain how and when source code rejuvenation can occur. This includes management approval requirements, organizational priorities, operational demands, and institutional restrictions that directly limit developers' ability to modify existing systems. This concept compiles five factors: 🏷 `Organizational Influence source code rejuvenation efforts`, 🏷 `Management approval required`, 🏷 `Keep the systems running and evolutive maintenances are more important than source code rejuvenation`

`efforts`, `Company Restrictions may hinder source code rejuvenation efforts`, and `Companies may not consider source code rejuvenation efforts Relevant`.

Regarding organizational influence, participants (C++ Surveys, P20, P24, and P25) described source code rejuvenation as shaped by broader company priorities and structures rather than purely technical considerations. Decisions about Source Code Rejuvenation are often embedded within organizational processes, where multiple stakeholders contribute to evaluating whether and when changes should occur. For example, P20 explicitly linked Source Code Rejuvenation capacity to organizational culture. In environments that value internal software quality, developers may allocate time for maintenance and Source Code Rejuvenation. In delivery-driven contexts, internal improvement must be embedded opportunistically within other maintenance activities. The organizational culture shapes timing and feasibility.

> *"Depends on the modification that you'll make and the environment you're in. If your environment values internal software quality, you can take your time and perform certain kinds of maintenance. If you're in an environment that pressures you to release and there's an agreement that the code's internal improvement is not something that could bring general benefits, you need to take the opportunity in another maintenance to make this kind of change. It depends on the culture."*
>
> [**P20**]

Another example, P24 described both personal and organizational pressures. Professional competitiveness motivates updating skills and modern language features usage, when company also supports adoption of new features, which facilitates source code rejuvenation. Staying updated enables participation in code reviews and collaboration. Here, organizational alignment promotes source code rejuvenation efforts.

> *"As I work in a private company, there is always a pressure of the following. If you don't update as a professional, you may have problems when you leave your current job. This pressure is mine, it is not imposed by my company, but it exists, and it is one of the motivating factors to always update the language. Another question is, the company where I work supports a lot the implementation of these new features. So there is also a company motivation. If you don't update today, you will be behind your colleagues. You won't be able to help them, which for me is an important motivation. I couldn't help my colleagues. And you won't be able to do PR, review the PRs. You won't be able to interact with your colleagues."*
>
> [**P24**]

In terms of management approval, participants emphasized that adopting new language features or performing rejuvenation typically requires explicit authorization. Developers may propose changes, but final decisions depend on managerial evaluation, reflecting hierarchical decision-making processes. For example, P01 described that bosses

will choose whether to give their permission, or not, even if the team points out the benefits.

> *"Corporately, there's this notion that being in an older version of the language means being obsolete, since many versions are not supported anymore, so there's still the factor of wanting to be, theoretically, in somewhat of a leading position, and the factor of not wanting to be too obsolete, as you may end up in a version that is no longer supported by the operating system. And there's also this language's benefits thing, so I think these three things could be added to this bundle, and then the team decides, and some boss will choose whether to give their permission, or not, the boss only goes along with it. It's mostly settled on a micro team scale, though sometimes, the entire IT team could be involved, but in either case, it depends on these factors."*
>
> [**P01**]

Regarding to keeping the systems running and evolutive maintenances are more important than source code rejuvenation efforts, participants (C++ Surveys, P01, P03, P04, P06, P08, P09, P10, P11, P12, P13, P14, P16, P18, P20, P22, P23, P25, P27, P28, and P29) also highlighted that maintaining system operation and delivering ongoing improvements often take precedence over rejuvenation efforts. Activities related to keeping systems stable and addressing immediate demands are prioritized, leading to the postponement or deprioritization of source code rejuvenation tasks. Source Code Rejuvenation competes with feature development and business priorities. For example, P20 emphasized that one should not evolve code merely to adopt new features; stable systems should remain stable.

> *"Actually, you don't evolve code to support new language features. What you do is use the new features to upgrade your code. You shouldn't change your code just to get the lastest brilliant stuff in the language. You should pick apart the new things in the language, find what makes your code better and then adopt it. You see? So, this should be the way of thinking, at least for a stable system. If it's a toy system, then you can just play. If it's a serious system, you normally have to keep what's working."*
>
> [**P20**]

In another example, participants (P03, P09, P28) described evolution being deprioritized due to client demands and functionalities pressure.

> *"I think it's really hard to do it for every version, as we deal with many new functionality demands inside the project, so we need to balance the time to work on these demands and the time to evolve. Generally, when there are new demands, the evolution has a lower priority, and it gets harder to keep updated. "*
>
> [**P03**]

Additionally, company restrictions, such as internal policies, regulatory requirements, or process constraints, can directly limit the ability to evolve systems. These constraints may prohibit certain changes or require strict adherence to predefined scopes of work. Participants (P23 and P25) describe explicit organizational and regulatory constraints that prevent language evolution.

For example, P23 provided an example from the automotive industry. They described formally validated compilers and industrial regulations that restricted moving from C++ 2003 to C++ 11. Some clients "cannot purchase or order code that has C++ 11 features." In this case, language evolution conflicts with certification requirements and external contractual obligations.

> *"for example, we worked with NNG, which, in Europe, was, a leading industrial company which mainly worked on the GPS mostly for cars. Back then, I think they had the Android application, although I think that they deleted it already. They are now just reselling their system to car factories. They've had some big issues moving to C++ 11 because, in the car industry, it's not enough to have a new feature and, hooray, we are changing to that. They have several constraints and laws. They often have some formally validated or verified language compiler, which was C++ 2003 back then. So, they had issues changing to C++ 11 as several car companies told them that they cannot purchase or order code that has C++ 11 features on it because their restrictions made it impossible. So, the industrial restrictions and rules are one thing that might prevent the developer from evolving a system or changing to a newer version of a language. And, of course, sometimes people are lazy and they just don't want to modify an already existing codebase that works well and which is tested, and the tests are passing and everything's fine."*
>
> [**P23**]

Another example, P25 described internal process restrictions. In sprint-based workflows, developers "don't change things that are not related to a specific demand." Source Code Rejuvenation outside the defined task scope is not permitted. This reflects procedural limitation rather than technical impossibility.

> *"Usually, when we are working on company code, we don't change things that are not related to a specific demand. For example, we have a sprint process, and it was determined that I make a specific demand in this sprint. So, I'm going to act only within this demand, I'm going to finish it."*
>
> [**P25**]

Finally, participants noted that organizations may not perceive rejuvenation as relevant when it does not produce immediate business value. Improvements that are not directly visible to stakeholders or customers may be deprioritized in favor of deliverables tied to revenue or functionality. Both participants (P23 and P25) framed source code rejuvenation as potentially undervalued within company settings.

For example, P23 described situations in which companies question why they should "pay you a month's salary for doing practically nothing other than enhancing your code." In this view, source code rejuvenation is not easily monetizable. The company "cannot resell it just because it's written in the newer language." P25 reinforced this delivery-oriented framing. They explained that in a company scenario, "you need to deliver," and that improvements not explicitly requested may delay visible progress. Showing that a demand is completed can be more valued than internal improvements.

> *"And, of course, in my opinion, sometimes companies are mad about it: why would I pay you a month's salary for doing practically nothing other than enhancing your code or something like that. The company cannot resell it just because it's written in the newer language, or something like that. "*
>
> [**P23**]

Across these observations, source code rejuvenation is positioned as secondary to business deliverables. The issue is not technical feasibility but perceived business relevance. If source code rejuvenation does not clearly translate into customer-visible value or revenue, it may be deprioritized. As a condition, Organizational and Managerial Constraints shapes source code rejuvenation by embedding technical decisions within organizational priorities and governance structures. This condition moderates the phenomenon by constraining autonomy, prioritizing short-term deliverables, and making rejuvenation contingent on managerial approval and perceived business value.

4.2.2 Effort, Cost & Practical Constraints

Effort, Cost and Practical Constraints captures the resource-related conditions surrounding software development. This includes the availability of time, effort, and resources, as well as competing priorities and practical limitations that characterize the development environment. This concept compiles three factors: `Source code rejuvenation is a non-trivial effort`, `Version gap increases source code rejuvenation cost`, and `Project complexity might hinder source code rejuvenation efforts`.

Regarding effort, participants (P01, P04, P05, P06, P07, P08, P09, P11, P22, P25, P26, P27, and P29) consistently described source code rejuvenation as a non-trivial activity that requires significant time and coordination. Source Code Rejuvenation is not perceived as a simple task, but rather as an effort that competes with feature development and other project demands, often leading to its deprioritization. For example, P06 emphasized that updating systems can be difficult when multiple environments and users depend on the software being continuously available. In this condition, stopping systems for Source Code Rejuvenation may not be feasible.

> *"It's not always possible to update, first of all, because you work with some mixed environments. This means that, in the development area, you have validation, development and production environments. And it is not always easy for you to tell every user of these environments, using a certain technology/language, that they will have to stop for a day, or a week, so the system can undergo an update. Or for us to refactor the system according to newer versions of languages, components, library, APIs, etc."*
>
> [**P06**]

In terms of programming language version gaps, participants (P17 and P29) highlighted that delaying updates increases the cost and difficulty of rejuvenation. Larger gaps between current and target versions accumulate changes, requiring more extensive adaptations, validations, and potential rework. As a result, postponed Source Code Rejuvenation tends to become progressively more expensive and complex over time. For example, P17 explained that when a system remains on very old versions of a language, developers may lose the opportunity to gradually address deprecations and incremental changes. As a result, migrating to a much newer version may require a large and disruptive update rather than a series of small adjustments.

> *"And if we are going from a very old to a new one that had lost this period of when things were only deprecated, that you could give them a warning and then they would be dropped, then you would have to perform the whole operation, you can't make it piece by piece."*
>
> [**P17**]

Finally, participants (P05, P06, P07, P08, P10, P11, P13, P14, P17, P20, P21, P22, P26, and P27) emphasized that project complexity itself can hinder rejuvenation efforts. Large, intricate systems with many dependencies and interactions increase the effort required to safely introduce changes. This complexity raises the risk of unintended side effects and makes it harder to plan and execute Source Code Rejuvenation incrementally. For example, P07 highlighted that stable systems demand revalidation of everything before migration, which raises cost and effort.

> *"I think the main challenge is the code base size. If your code base is big, you may need to test a lot of code. I'll number it: code base size, application stability. If you have a quite stable application, even if it uses a previous version of the language, it's tough to pay the price of having to revalidate everything that was in that application."*
>
> [**P07**]

As a condition, Effort, Cost and Practical Constraints shape source code rejuvenation by constraining when and how source code rejuvenation activities can be carried out under

limited resources and competing priorities. This condition reflects how the non-trivial effort required, the accumulation of version gaps, and the complexity of existing systems jointly increase the cost and risk associated with rejuvenation. As a result, Source Code Rejuvenation is often deferred, approached incrementally, or selectively avoided when the required effort and coordination are perceived to outweigh its potential benefits.

4.2.3 Compatibility & Legacy Constraints

Compatibility and Legacy Constraints captures the conditions under which legacy systems and backward compatibility requirements constrain and shape how source code rejuvenation can be performed, limiting the adoption of modern language features and influencing when and how changes can be introduced. This includes systems that rely on older language versions, deprecated features, or constraints that limit the ability to modify or evolve existing implementations. This concept compiles three factors: 🏷 `Backward compatibility constraints hindering source code rejuvenation`, 🏷 `Lack of retrocompatibility might hinder source code rejuvenation efforts`, and 🏷 `Transpilation overhead constraining adoption`.

Regarding backward compatibility, participants (C++ Surveys, JavaScript PR discussions, Python PR discussions, and P25) consistently described it as a structural constraint that limits the adoption of modern language features. In several cases, developers explicitly avoided or reverted newer constructs because projects were required to support older runtimes, environments, or user bases. For example, contributors in pull request discussions highlighted the need to maintain compatibility with legacy platforms, leading to the use of older syntax and postponement of source code rejuvenation until support for previous versions is dropped. This indicates that rejuvenation is often gated by explicit compatibility policies rather than purely technical considerations. For example, Some developers comment in a pull requests:

> 💬 *"ES6 is good but not all supported platform supported 'let'. (e.g.: PaleMoon or Chrome < 41)"*
>
> [**JavaScript PR discussions**]

> 💬 *"Ideally, we'd use yield from here, but sadly Python 2."*
>
> [**Python PR discussions**]

In terms of lack of retrocompatibility, participants (P07, P08, P20, and P26) emphasized the operational risks associated with evolving systems that are not designed to accommodate newer language versions. Changes may break existing behavior, requiring extensive validation, coordination, and sometimes redesign. This introduces additional

effort and uncertainty, discouraging rejuvenation efforts when stability of the current system is prioritized.

P20, for example, shows how backward compatibility and dependency chains limit the source code rejuvenation efforts. It also stresses that adding new features often requires coordinated updates across compilers and upstream dependencies. In this case, the "upstream chain" is the order of external parts that a system needs to work, like compilers, libraries, and frameworks. These parts need to change first before systems that depend on them can safely use newer language features. Rejuvenation relies on these earlier updates across the dependency stack instead of just changes made in one place.

> *"Therefore, the change is not simple, as to perform it you need to change the compiler, because new language features are not available in the one you have. I don't recall exactly what version our compiler was, but it's certainly not from this year. Our compiler is already old. But then binary compatibility issues arise. You'll only be able to modify your version when the upstream chain [compilers, libraries, and frameworks] changes. That way, until it reaches you, everyone, every library in use will need to change, and then you can change yours too."*
>
> [**P20**]

Finally, regarding code transpilation, in the JavaScript PR discussions, The contributor described transpilation overhead as a practical constraint that limits the adoption of modern language features. Although newer constructs (e.g., ES6) are available, their use may require additional build steps—such as transpilation by tools like Babel—that significantly increase the size and complexity of the generated code. As a result, developers may deliberately restrict the use of modern features to avoid unnecessary overhead. This highlights how tooling limitation can discourage broader adoption, even when developers are otherwise willing to conduct source code rejuvenation efforts.

> *"Yeah, I am very new to ES6 modules and I was in a hurry so the format I used was mostly determined by a search-and-replace. Also I had to be careful to only use ES5 features _outside_ the modules because I found that using 'babel' to transpile from ES6 to ES5 added a huuuge amount of overhead even if I was only using one or two features in one or two places."*
>
> [**JavaScript PR discussions**]

As a condition, Compatibility and Legacy Constraints shapes source code rejuvenation by anchoring technical evolution to existing systems, environments, and user commitments. This condition moderates the phenomenon by delaying or preventing source code rejuvenation, enforcing conservative adoption strategies, and making rejuvenation rely on the deprecation of legacy requirements rather than on the availability of new language features, as well as on the practical overhead introduced by compatibility mechanisms such as transpilation tools.

4.2.4 External Dependencies & Environment Constraints

External Dependencies and Environment Constraints captures the conditions under which external libraries, frameworks, community support, and runtime environments constrain and shape how source code rejuvenation can be performed. These conditions arise from factors such as outdated, unsupported, or tightly coupled dependencies, as well as inactive or weakly maintained communities that limit the availability of updates, documentation, and technical support. In addition, environment-specific configurations impose operational constraints that restrict how systems can be modified and evolved. Together, these factors influence the feasibility, timing, and scope of source code rejuvenation efforts. This concept compiles three factors: `Non-active communities might hinder source code rejuvenation efforts`, `Framework/library dependencies might hinder source code rejuvenation efforts`, and `Environment configurations might hinder source code rejuvenation efforts`.

Regarding community activities, P19 compared the Python and C++ communities with respect to documentation and knowledge dissemination. Python was described as having extensive literature and rapid community activity. In contrast, C++ was characterized as having slower consolidation of modern knowledge. Community vitality appears as an enabling condition for Source Code Rejuvenation. When modern practices are well-documented and widely discussed, adoption becomes more justifiable and accessible. When literature lags behind language evolution, hesitation might increases.

> *"The literature for Python is extensive, and the people working in the Python community do it with celerity. There is a delay in C++; it is natural because each community is different from the other, people differ. In the literature for C++, the most modern stuff focused on what we have today falls short. It is not as common as other languages. But I think that would be the main point to adopt new language version [and modern features], the adopting—because it's not just adopting for the sake of adopting, it's adopting and having a theoretical basis so that it makes sense in the context of work."*
>
> [**P19**]

Regarding framework and libraries dependencies, it appear as structural constraints that slow down or block source code rejuvenation. Participants (C++ Surveys, P05, P13, P20, P21, P23, and P26) described situations in which development frameworks and third-party libraries evolve at a different pace from the programming language, requiring coordinated updates across the "up-stream chain". For example, P23 highlighted deprecated or legacy dependencies that block language updates, requiring replacement or adaptation of each outdated component.

> *"For example, with this JavaScript project, we couldn't move on for a long time since we had dependencies that are deprecated or that are now legacy. And there*

weren't any newer versions that we could use. So that's another reason I experienced that might also prevent the code or the language evolution."

[**P23**]

In another examples, C++ Surveys responses showed that Qt provides its own concurrency primitives and abstractions, reducing incentives to adopt equivalent C++ standard features. In some cases, compatibility with older Qt versions used by known users prevented adoption of newer C++ constructs.

"Framework libraries influence source code rejuvenation Since the Qt API is mostly frozen in C++98/C++03 era (and C++11 support in Qt can mostly be reduced just to auto and lambdas), there's little incentive for the community to actively use latest C++ features."

[**C++ Survey responses**]

Finally, regarding environmental configurations, participants (P06, P09, P12, and P25) described environmental constraints as multi-layered limitations on source code rejuvenation, spanning hardware vendors, client dependencies, and deployment infrastructures. In low-level contexts, developers may be constrained by toolchains or hardware manufacturers, limiting their flexibility in adopting newer language versions.

In client-facing systems, compatibility requirements discourage changes that could break external integrations. Operational factors, such as multi-stage environments and the need for coordinated downtime, further complicate upgrades. Overall, rejuvenation depends not only on code changes but also on infrastructure readiness and synchronized environment updates, which act as significant constraints.

For example, P12 explained that in low-level programming contexts, they use the language version defined by the toolchain or hardware producer. They stated that they are often "locked by the producer," and that basic language functions rarely change enough to justify version switching. This reflects vendor-imposed constraints.

"Since I work with low-level [programming], that is my programming area, I use the language or the language version the toolchain or the producer, or the controllers provide. That evolves with time, of course, but most of the time, at this point, I'm stuck with the versions that the producer defines. To be honest, I don't care that much about a language version when I'm programming. Firstly, because I'm often locked by the producer. Secondly, because the functions I use, I mean, the functionalities, methods, etc. that I use are very common. They are generic functions that hardly change from a version to another."

[**P12**]

As a condition, External Dependencies and Environment Constraints shape source code rejuvenation by constraining how and when source code rejuvenation activities can

be carried out within a broader socio-technical setting composed of frameworks, communities, and runtime environments. This condition reflects how dependency chains, varying levels of community support, and environment-specific constraints jointly influence the feasibility, timing, and scope of rejuvenation efforts. As a result, developers' ability to adopt modern language features is often limited, leading to delayed, coordinated, or incremental upgrades, and, in some cases, preventing adoption altogether when the surrounding technical environment is not ready to support such changes.

4.2.5 Availability of Automated and Regression Testing

Availability of Automated and Regression Testing captures the conditions related to the presence and maturity of testing practices that support safe system evolution. This includes the availability of automated test suites, regression testing mechanisms, and validation processes that allow developers to verify behavioral preservation during changes. This concept compiles one factor: `Relevance of regression/automated testing`.

Participants (JavaScript PR discussions, P01, P02, P03, P04, P07, P08, P10, P12, P14, P15, P20, P22, P23, and P27) described that the availability of automated and regression testing acts as a critical condition for enabling source code rejuvenation. When robust test coverage is present, developers are more confident in performing changes, as they can verify that system behavior remains consistent after modifications.

Testing provides a safety net that supports the detection of regressions and reduces uncertainty during source code rejuvenation. For example, P08 described writing automated tests before rejuvenating parts of an application to gain confidence that failures would be detected immediately. P07 noted that higher test coverage significantly increases confidence during source code rejuvenation and enables tools to be used more safely.

> *"Interviewer: I see. So, you consider legacy code hard to evolve. When adopting a new language feature, does badly written code make it hard to evolve?*
>
> *P08: It doesn't get hard to migrate [source code rejuvenation] solely because it's badly written, but if it's badly written, it will always be hard to migrate. With badly written code, you'll always be afraid of it crashing with any change, if it doesn't get a lot of tests. Something I did in this app's migration was: there was a part that came from its API client, that had no automated tests, so I left it for last when migrating, as I was afraid the stuff under production would crash. Before migrating, I wrote tests for the whole part or the API client, and when I finished them, I felt more confident with the migration. I changed the language's version, and when something crashed, I saw it right away and was able to fix it."*
>
> [**P08**]

Conversely, when such testing infrastructure is absent or insufficient, developers face increased risk and are less willing to modify existing code, often postponing or avoiding rejuvenation efforts. For example, P20 emphasized that without tests, modifications become risky reengineering rather than safe refactoring or Source Code Rejuvenation efforts.

> 💬 *"For untested code, it's not refactoring , but reengineering. When you say reengineering, you terrorize the one in charge of managing. When they say: let's refactor it, I'll refactor it - the guy in management will think that it's something with no risks involved. Right? But this is only possible if you already have tests; if you don't, it's not refactoring. "*
>
> [**P20**]

As a condition, Availability of Automated and Regression Testing shapes source code rejuvenation by influencing the level of confidence developers have in performing changes. This condition moderates the phenomenon by enabling safer and more proactive rejuvenation when testing is available, and by constraining or delaying it when validation mechanisms are limited or absent.

4.2.6 Risk, Uncertainty & Fear Factors

Risk, Uncertainty and Fear Factors captures the perception and presence of uncertainty associated with modifying existing systems. This includes concerns related to potential regressions, system instability, unknown side effects, and the perceived reliability of legacy codebases. This concept compiles three factors: 🏷 `Source code rejuvenation efforts may introduce bugs`, 🏷 `Fear / risk aversion about upgrading`, and 🏷 `Trustness and reliability in legacy systems may hinder source code rejuvenation efforts`.

Regarding the introduction of bugs, participants (JavaScript PR discussions, P12, P18, P22, and P24) consistently described rejuvenation as an activity that may unintentionally introduce defects into previously stable systems. Even small changes can propagate unexpected issues, requiring careful validation to ensure that existing behavior is preserved. This reinforces a cautious stance toward modifying working code. For example, P18 explained that when developers update code due to external changes, such as programming language updates, the modification process itself may introduce new bugs. This can occur when developers are unfamiliar with the new constructs or when changes are performed quickly without sufficient planning.

> 💬 *"If the update leads to an incompatibility in the code, and I need to implement it, there is a problem because I will have to change the usual development flow to*

make this adaptation. I see this type of work as unproductive. You are required to update your code due to a change that you didn't even demand. Then, this problem is more related to the time available to implement the changes. Another challenge I see is the introduction of bugs in the source code. You are required to update what was once functioning well, but in the process of evolving the code, new bugs might be introduced. Maybe because you don't know the new structure [new language construct or syntax] well enough or because you apply it in a hurry, as this wasn't planned, you end up introducing a bug. Those are the main difficulties I see in this process."

[**P18**]

In terms of fear and risk aversion, source code rejuvenation was frequently framed as a potentially disruptive action. Participants (JavaScript PR discussions, P01, P02, P08, P09, P12, P24, P25, and P29) reported concerns about instability, regressions, and unintended consequences, especially in large or widely used systems. These risks increase the perceived cost of change and lead developers to postpone or carefully stage source code rejuvenation efforts. For example, P25 described managerial reluctance rooted in concern about delaying demands or introducing production issues.

"And in most cases that I suggested an improvement, an evolution in the Java version, for example, it was rejected because of the fear of the managers above. Sometimes he didn't even have technical knowledge, but he said, it's working like this because I'm going to migrate to something and risk delaying several demands. You could present it to him, but it may have a performance gain of so many percent, it may solve future problems."

[**P25**]

Finally, participants (P12 and P20) highlighted that trust in legacy systems plays a significant role in limiting rejuvenation. When existing code is perceived as reliable and stable, developers may prefer to maintain the current implementation rather than introduce uncertainty through change. This perceived reliability reduces the urgency to adopt new features and reinforces conservative decision-making. For example, P20 draws an analogy with long-standing engineering practices to emphasize how proven reliability in legacy systems discourages the adoption of newer, less-tested solutions, particularly in critical systems.

"Nowadays, some cars utilize parts, engines, or gears that have been used for 40 or 50 years, because they were already tested and are reliable enough for that critical task; when you deal with low-level, that are critical tasks on their own, you must value reliability. It is not that functionality that is new, it is not reliable, but until it becomes reliable, it will hardly be implemented into a low-level."

[**P20**]

This condition captures how the perceived risks of change, including potential regressions, risk aversion, and trust in legacy systems, jointly shape the feasibility and timing of source code rejuvenation efforts, often resulting in conservative decision-making and delayed adoption.

4.2.7 Unpredictability and Interaction Complexity in Language Evolution

Unpredictability and Interaction Complexity in Language Evolution captures how the continuous evolution of programming languages introduces interacting features, expanding specifications, and emergent behaviors that are difficult to anticipate and control in practice. This includes the combinatorial interaction between new language features, the difficulty of keeping up with rapid evolution of programming languages, and the emergence of unintended side effects such as hidden regressions, compiler warnings, or subtle behavior changes. It also reflects how these factors increase uncertainty in system behavior, require additional validation effort, and often lead developers to delay adoption until modern features become stable and better understood.

This concept compiles several factors: `Difficulty in following language evo lution`, `Feature interactions creating unexpected side effects`, `Language evolution introducing silent or unintended behavior changes`, `Language evo lution triggering hidden regressions`, `Compiler evolution exposing latent issues (e.g., warnings)`, `Perceived risk and uncertainty in upgrading`, and `Delayed adoption due to instability and lack of maturity`.

Regarding difficulty in following language evo lution, participants (P01, P03, P22, P23) report difficulty in following language evolution, driven by rapid release cycles, which demand continuous learning and are hard to sustain alongside daily development tasks. For example, P03 linked difficulty to balancing new feature demands with maintenance tasks. P01 described annual or semiannual version releases as overwhelming, especially when working across multiple languages.

> *"I think it's really hard to do it for every version, as we deal with many new functionality demands inside the project, so we need to balance the time to work on these demands and the time to evolve. Generally, when there are new demands, the evolution has a lower priority, and it gets harder to keep updated."*
>
> [**P03**]

In terms of language evolution may introduce breaking changes and subtle behavioral shifts, participants (P05, P12, and P22) that source code rejuvenation demands broader structural adaptations, such as reorganizing projects or updating configurations. For

example, P12 refers to changes in function signatures, where modifications in parameters or structure in modern features cause previously valid calls to fail, leading to breaking changes in dependent parts of the system.

> *"Well, the first challenge is to not break anything; to not break the code. It can have an impact on other places or cause some conflicts because some function or prototype [method signature] changed... I've seen it happen. Some functions changed their prototype and broke the code. This is a big challenge: not to let the code break."*
>
> [**P12**]

Another recurring aspect concerns about silent behavior changes and hidden regressions, which increase perceived risk during language evolution. Participant (P22) described situations in which identical code compiles without warnings across language versions but produces different runtime behavior, illustrating semantic drift without syntactic breakage. These effects are particularly concerning in large codebases, where even rare divergences can scale into significant issues. This reinforces the uncertainty associated with language version upgrades which are prerequisite for code revitalization efforts.

> *"One of my peers is working on language evolution stuff, because we were aware, and we have detected several side effects of the new C++ version. Let's imagine you have a C++ code that compiles (and notifies), you get warnings of any side effects, it's absolutely valid C++ code in C++ 89 or 98 version, it's a standard C++. When it comes to the new version, the new C++ 11 version, the very same code is compiled, and it's again, without warnings, and the program behaves completely differently. "*
>
> [**P22**]

Regarding the impact of new language features on debugging, participant (P11) described that the impact of new language features on debugging varies according to their complexity. While simpler constructs (e.g., functional operations like map) are perceived as straightforward and easy to adopt, more advanced features, particularly those related to concurrency, threading, and metaprogramming, introduce significant debugging challenges. For example, P11 emphasized that runtime errors may appear without clear traceability, and debugging tools may provide limited assistance. The example of Ruby metaprogramming illustrates this concern.

> *"When Java added features from functional programming, for example, the map function, I started using it immediately, without hesitation. Using loops for everything is a thing of the past, right? But some other features take a while. When a feature relates to processing, like threading, or things that have to do with concurrency, parallel computing, and things like that, we try to be a little more careful, because it's very hard to figure out the root of a problem when something goes wrong. And when I say that it's hard to find, I mean it: debugging doesn't help; you start*

getting runtime errors. By the way, let's take a look at Ruby and metaprogramming. I find it awesome, but I can't always use it for it gets startling whenever there's a problem, since it's hard to find where the problem is."

[**P11**]

Additionally, about the upgrading language implementations (e.g., compilers) or adopting modern constructs may introduce unexpected performance degradation, even when improvements are anticipated. Participants (JavaScript PR discussions and P20) reported cases where newer compiler versions resulted in slower execution under certain workloads, as well as concerns that modern language features may increase runtime overhead (e.g., memory allocation or garbage collection). These uncertainties reinforce the need for careful performance evaluation during upgrades and may lead developers to postpone source code rejuvenation when performance impacts are unclear. For example, in a PR comment, a contributor emphazed a problemas with garbage collector in a JavaScript project:

"suggestion ... don't optimize this loop with the 'arrayEach()' method or arrow functions ... otherwise performance will decrease because of garbage collection ... using the '...rest' syntax (ES6 and later) will decrease performance as well"

[**JavaScript PR discussions**]

Finally, regarding programming language complexity, participants (C++ Surveys, P20, P22, and P29) emphasize that the intrinsic semantic complexity of modern programming languages constitutes a significant barrier to source code rejuvenation efforts. In particular, the interaction between language constructs and the presence of subtle, nonlocal effects require deep semantic reasoning. As a result, transformations are difficult to perform safely without risking unintended side effects. This limitation directly constrains the effective use of automation. While tools may assist with syntactic transformations, they remain insufficient for handling semantically complex changes, reducing developers' trust in automated support.

For instance, in C++ Surveys responses, a contributor echoed concerns about tool trust. Although IDEs provide automated functions, the participant stated reluctance to rely on them in C++. Pointer hierarchies and static analysis limitations were cited as complicating factors. In contrast, the participant indicated more willingness to trust automation in less complex languages.

"The tool that I use is an editor, an IDE which is Clion, I don't know if you have heard of it....And from time to time I also use Kdevelop, which is from KDE, but so, as much as I like open source, kdevelop is one level below Clion. Not that Clion is better, but it is below the level of other tools from the company, but it is better than Kdevelop which I also use...No, it shows, I could, have the automated functions but I don't trust them. Even more in C + +, in Java, I would even trust

it, but C + + is a slightly more complicated language I would say, pointer hierarchy issues, things that it is difficult for you to have a more correct static analysis."

[**C++ Survey responses**]

As a condition, Unpredictability and Interaction Complexity in Language Evolution constrains source code rejuvenation by increasing uncertainty and reducing developers' ability to safely reason about code transformations. The interaction between evolving language features, the emergence of unintended side effects, and the limited capability of automated tools to handle semantically complex changes make rejuvenation efforts harder to plan, validate, and execute. This condition elevates the perceived risk and effort associated with source code rejuvenation, leading developers to adopt more cautious strategies, such as delaying adoption, prioritizing incremental changes, or avoiding transformations that require deep semantic understanding. Consequently, it not only slows the pace of rejuvenation but also limits the scope of changes that developers are willing to perform, directly shaping how and when source code rejuvenation occurs in practice.

4.2.8 Tooling Constraints and Control Mechanisms

Tooling Constraints and Control Mechanisms captures how developers' perceptions of trust, risk, and control over tools shape the feasibility and extent of tool-supported source code rejuvenation. Rather than assuming tools as inherently reliable enablers, developers consistently describe tooling as constrained by limitations in correctness, predictability, and transparency. These conditions influence whether and how tools are adopted, often restricting their role in rejuvenation activities. To this concept, we identified two key factors: 🏷 `Lack of trust in programming transformation tools` and 🏷 `Tool configurability influencing source code rejuvenation adoption and control`.

Regarding lack of trust in programming transformation tools, this concept emerges as a limiting condition for developers to count on the support of automated tools for apply source code rejuvenation. Participants (C++ Surveys, P01, P04, P05, P07, P20, P23, and P27) frequently associate automated transformations with the risk of introducing subtle bugs, unexpected behavior, or loss of control over the codebase. This concern is particularly pronounced in large, legacy, or critical systems, where even small unintended changes may have cascading effects. As a result, developers avoid relying on tools for complex or semantically meaningful transformations, constraining the scope of tool-supported rejuvenation. For example, P01 emphasized manual intervention in mission-critical systems, reinforcing that runtime risk constrains automation.

> *"Well, as I've already worked on less critical systems, and more critical ones, in this critical system it had to be done by hand, because you need to have criteria to evaluate whether that will cause a problem in runtime."*
>
> [**P01**]

Another example, a contributor for a KDE project argues in a comment that is not simple to replace a macro such as Q_FOREACH from Qt by range-based for loops from C++11 and did not recommend automation for this kind of transformation.

> *"Rarely. And usually I don't use any tools. Often you need to look and think, so can't be easily automated with a tool. E.g. Q_FOREACH can't be just replaced with range-for (`https://www.kdab.com/goodbye-q_foreach/`), you have to be careful what is const and what is not const."*
>
> [**C++ Survey responses**]

At the same time, tool configurability plays a dual role as both an enabling and controlling condition. Participants (P01, P02, P05, P11, P22, and P23) emphasize that tools become more acceptable when their behavior can be configured, allowing them to disable certain checks, restrict automatic transformations, and control how suggestions are presented. This configurability mitigates some of the risks associated with automation by reinforcing human oversight and selective adoption. For example, P11 explained that automated tools that modify code without explicit confirmation may introduce confusion or unexpected changes. For this reason, tools that present suggestions and allow developers to manually confirm modifications are often preferred.

> *"P11: Linters give good hints for language evolution, Microsoft's ReSharper is the best, I have used it a lot. It's the best. JetBrains does a great job in this regard, all of its tools are good. The best tool I have used so far is ReSharper from JetBrains.*
>
> *Interviewer: How does he do this process? Just to have an idea. Does it suggest to update the language version or update just the code?*
>
> *P11: It gives suggestions. Some code snippets are shown like this: "the best suggestion is this, this here will depreciate, this here would be better this way". It runs on suggestions, it doesn't change everything for you, as you navigate through the code it suggests improvements. For those who program in Java and C#, there can be several versions of that framework installed on your machine, and you might wonder: will it be compatible with such and such, and when you have a very old code base in a very old version, version five Java. Suddenly you are compiling it in version eight, it will suggest to you: this section can be changed in this version. This guides you. I was recently using Prettier and Lint from Javascript that changes that. I'm a little bit wary of doing things that I don't know now what has done it, you know? Depending on the place and rule, I can might get confused. The tools get confused too. I use it, but with restraint and a certain distrust."*
>
> [**P11**]

As a condition, Tooling Constraints and Control Mechanisms emphazed that tools which present suggestions and allow developers to manually confirm modifications are often preferred, according to these participants. As a condition, these findings indicate that tooling in Source Code Rejuvenation is fundamentally conditioned by trust and control considerations. Rather than acting as a fully automated tool, they are often perceived as imperfect and potentially risky, which may make it difficult to rejuvenate efforts when adequate, reliable, and trustworthy tooling support is lacking, especially in environments where developers require assurance that their modifications will not introduce errors.

4.2.9 Human & Knowledge Factors

Human and Knowledge Factors captures the conditions related to the distribution of knowledge, expertise, shared understanding, and openness to change within development teams that influence how and when source code rejuvenation can occur. This includes team familiarity with modern features, alignment of technical knowledge, and resistance to adopting new practices or language constructs. This concept compiles two factors: 🏷 `Uniform knowledge` and 🏷 `Resistance to change`.

Regarding uniform knowledge, participants (P01, P02, P05, P09, P11, P13, P24, P26, and P28) described that the level and distribution of knowledge within teams condition the feasibility of rejuvenation efforts. When knowledge is uneven or limited, developers may lack confidence to adopt modern features or perform transformations safely. Conversely, lack of shared understanding across the team can hinder coordinated changes. For example, P01 noted that developers sometimes avoid adopting advanced features until knowledge about them is widely shared within the team. This helps ensure that future maintenance tasks can be performed by any team member.

> 💬 *"I try to avoid it, including in other jobs I had in the past, with C++ too. Like, if knowledge on that resource is not shared by the team, not only me, but many people, try to not adhere to that knowledge while it's not uniform. Someday it will need maintenance, and if that person doesn't know about it, it will not help. So, I think that this shared vision is very present, so in my team, and in other teams I have worked with, advanced resources created in Java and C++ are not added until that knowledge is uniformized."*
>
> [**P01**]

Additionally, resistance to change acts as a behavioral constraint, where developers may avoid modifying familiar systems due to discomfort with new constructs or reluctance to alter established practices. According to participants (C++ Surveys, JavaScript PR discussions, P20, P24, and P29) these factors reduce the likelihood of initiating or successfully executing rejuvenation. For example, P24 reported that when lambda expressions

were introduced in Java, developers initially resisted adopting them because they changed familiar programming patterns such as loops.

> 💬 *"Our next evolution was with Java 8. At first, there was a resistance, between me and my colleagues, regarding Lambda, because it was very disruptive in relation to the repetition loops."*
>
> [**P24**]

As a condition, Human and Knowledge Factors shapes source code rejuvenation by constraining both the capability and the willingness of development teams to adopt and apply modern language features. The lack of uniform knowledge limits developers' confidence and ability to safely perform transformations, while misalignment within the team hinders coordinated and sustainable changes. At the same time, resistance to change reduces openness to new constructs and discourages the adoption of unfamiliar practices. Together, these factors increase uncertainty, slow down decision-making, and reinforce conservative behaviors. As a result, this condition moderates the phenomenon by delaying adoption, restricting the scope of rejuvenation efforts, and shaping how source code rejuvenation is negotiated and implemented within teams.

4.2.10 Technical Debt & System Quality Constraints

Technical Debt and System Quality Constraints captures the internal state of software systems in terms of accumulated technical debt and code quality. This includes characteristics such as high complexity, low modularity, and structural deficiencies that define the condition of the codebase. This concept compiles three factors: 🏷 `Non modular architecture might hinder source code rejuvenation efforts`, 🏷 `Source code rejuvenation might difficult code understanding`, and 🏷 `Lack of code internal quality might hinder source code rejuvenation efforts`.

In terms of architecture, participants emphasized that non-modular systems hinder rejuvenation efforts. Tightly coupled components and lack of clear boundaries make it difficult to isolate changes, causing modifications in one part of the system to propagate across multiple areas. For example, P17 described system architecture as influencing the feasibility of incremental rejuvenation. In monolithic systems with extensive coupling, migration from very old versions to newer ones may require a large, coordinated effort. Non-modular architecture is therefore framed as a structural barrier to incremental source code rejuvenation. It raises the scope and risk of source code rejuvenation efforts.

> 💬 *"Perhaps I'd put the dimension of the impact of the change; if it has an extensive application or a monolith, it is a big procedure that you have to perform. And if we are going from a very old to a new one that had lost this period of when things*

> *were only deprecated, that you could give them a warning and then they would be dropped, then you would have to perform the whole operation, you can't make it piece by piece."*
>
> [**P17**]

Regarding source code rejuvenation might difficult code understanding, participants (C++ Surveys, JavaScript PR discussions, Python PR discussions, P16, and P24) also highlighted that rejuvenation itself may make code harder to understand, especially when modern constructs are introduced into already complex or poorly structured systems. This can increase cognitive load and reduce maintainability if not carefully managed. Some quotations highlighted the tension between expressiveness and readability. For example, P16 explained that certain constructs such as lambda expressions simplify implementation but may also make code harder to read, especially for developers unfamiliar with the syntax.

> *"Lambda Expression facilitates a lot, but it can also make the readability harder. Some syntax gets more elaborate. The inline function itself has existed since the early days of Java, but it makes readability very difficult. Less experienced people have difficulty. Lambda expressions have the same problem. Evolving code has pros and cons like everything else in life, but it simplifies many things."*
>
> [**P16**]

Finally, about internal code quality, participants (C++ Surveys and P08) noted that low internal code quality—such as poor structure, high complexity, and lack of clarity—acts as a barrier to source code rejuvenation efforts. Internal quality appears as a precondition for smooth rejuvenation. When absent, it transforms source code rejuvenation from a feature adoption effort into a broader restructuring process. For example, in C++ Surveys responses, developers described organizational contexts where source code rejuvenation is postponed until the code quality has already degraded significantly.

> *"So in a business context it's relatively difficult to get time/ budget for doing these kind of things, usually it's being done when the code quality has degraded quite a bit already, i.e. when it's too late or people finally get too annoyed working with the legacy code base."*
>
> [**C++ Survey responses**]

As a condition, Technical Debt and System Quality Constraints shape source code rejuvenation by constraining the ability to safely modify and evolve systems with poor internal structure. This condition moderates the phenomenon by increasing risk, effort, and uncertainty, often limiting rejuvenation to localized changes or discouraging it altogether in low-quality codebases.

4.2.11 Standards, Governance & Process Constraints

Standards, Governance and Process Constraints captures the presence of organizational standards, coding conventions, and formal processes that regulate development practices. This includes rules, policies, and governance mechanisms that define acceptable changes within the system. This concept compiles one factor: 🏷 `Code standards might prevent source code rejuvenation efforts`.

Regarding coding standards, participants described how established conventions and project guidelines can restrict the adoption of modern language features. In several cases, developers avoided introducing newer constructs to maintain consistency with the existing codebase, adhere to agreed patterns, and ensure readability for the broader contributor community. Changes that introduce stylistic or syntactic divergence are often discouraged unless they are applied systematically across the entire project.

Participants also highlighted that such standards are not purely technical but reflect governance practices aimed at maintaining stability, accessibility, and coherence in collaborative environments. As a result, Source Code Rejuvenation efforts may be rejected or postponed when they conflict with established norms or are not aligned with project-wide decisions. For example, the PR comments below draws this conditions:

> 💬 *"Well I love ES6 (and TypeScripe) but when I always try to follow the pattern of existing code and found that existing code is using 'var' mostly and using snake case variables so used 'var'. Thanks anyways. Updated this for new code."*
>
> [**JavaScript PR discussions**]

> 💬 *"Please use regular functions rather than arrow functions for describe it etc. Both for consistency and so that we don't have to change the type of function later on if we want to use user contexts."*
>
> [**JavaScript PR discussions**]

As a condition, Standards, Governance and Process Constraints shapes source code rejuvenation by regulating which changes are acceptable within a project's socio-technical boundaries. This condition moderates the phenomenon by enforcing consistency and collective agreement, often limiting or delaying the adoption of modern features when they conflict with established standards.

4.2.12 Language & Architectural Enablers

Language and Architectural Enablers captures the technical conditions that facilitate source code rejuvenation through inherent characteristics of programming languages and system architectures. This includes flexibility of language constructs and architectural styles that support modular and incremental change. This concept compiles two

factors: `Service oriented computing favors source code rejuvenation efforts` and `Programming language flexibility`.

Regarding service-oriented architectures, Participant (P4) emphazed that modularization and separation of concerns, allowing changes to be applied incrementally within isolated components. These characteristics lower the barriers to Source Code Rejuvenation and support safer evolution of the system. For example, P04 emphasized that modern software architectures make this selective evolution easier. By decomposing systems into services or smaller components, developers can rejuvenate individual parts of the system without affecting the whole application. This architectural structure reduces the risk associated with modernization and enables gradual evolution of the code.

> *"It depends a lot on the requisites, you know? For example, when we're working on a system that's already pretty old, we avoid refactoring features that are already under production, unless it's strictly necessary. Or else, you would have to go through that whole process again to revalidate the implementation. So, we'll update certain parts of the software, only the necessary ones. Nowadays, we can break down the software into services, it's not that giant stack anymore, which was very difficult to change. Now, you can divide it into micro sets, and it helps a lot in maintaining and evolving the software. Because you can migrate and add new features only to parts where it's safe to do it, you don't need to do it as a whole."*
>
> [**P04**]

Similarly, programming language flexibility, participant (P1) described that certain technical properties can enable rejuvenation by reducing the complexity and risk of change. Flexible programming languages allow developers to integrate new features more easily, adapting code without extensive restructuring. For example, P01 contrasted older and newer languages in terms of flexibility and openness to change. Older languages were described as inflexible, with long periods in which version upgrades did not occur. The example of a C++ project remaining effectively tied to older standards for many years illustrates this stability, or inertia. According to the participant, newer languages appear more flexible, and the developers who work with them also tend to accept change more readily.

> *"Interviewer: Sorry, I have a question. You commented that in the Java community, people are more open to evolve the code following language evolution than the C++ community, for example. Do you have any idea of why it happens, or other examples of communities that may be open, or not, to these code changes?*
>
> *P01: Well, it's just an opinion, because I only know these two communities, as I don't really follow the others as a programmer, just from the outside. But I think that older languages have always been inflexible for change. These newer, modern languages are more flexible, and consequently, so is who works with them, in my opinion. For example, take C++, it's as I said, we worked with the 98 version until*

2014, and nobody changed the base code. The project wasn't in 98, it was in 2014, but from 2004 to 2014, nobody changed the language version.

Interviewer: And in 2004, there was even a newer version of the language, 2003 version, but they chose not to use it.

P01: Right. But it was not considered mature, it had just a few things, but even so, there was fear too. It was only from the 2011 version that people began to consider doing it, and also when the 11 version was launched, in 2013, 2014. But I left the project, so I don't know if they upgraded to 14, 17, 20, that's coming now. But anyways, it took a long time, because the people who used it are a little held back in terms of evolution, see? In my experience, they are people who usually like the thing being functional. And for the people who use Java, as there are various, small systems, the susceptibility for evolving is higher. But only because the susceptibility is higher, that doesn't mean that evolving it is a small job, as there's this whole thing of getting stuck in the framework. But I believe it has to do with newer and older languages."

[**P01**]

As a condition, Language and Architectural Enablers shapes source code rejuvenation by providing favorable technical conditions that support incremental and controlled change. This condition moderates the phenomenon by facilitating adoption, reducing risk, and enabling more frequent and manageable rejuvenation efforts.

4.2.13 Perceived Relevance & Motivation

Perceived Relevance and Motivation captures the conditions under which language evolution and Source Code Rejuvenation are considered important, or not, within the development context. This includes situations where Source Code Rejuvenation is perceived as optional, secondary, or not immediately necessary, directly influencing whether rejuvenation efforts are initiated. This concept compiles one factor: `Language evolution is not that relevante to motivate maintenance efforts`.

Participants (P16 and P18) described that the perceived relevance of language evolution acts as a conditioning factor in decision-making. When source code rejuvenation is not seen as providing clear or immediate benefits, developers are less motivated to allocate time and effort to rejuvenation. In such cases, maintaining the current system and focusing on other priorities becomes more justifiable than introducing changes.

This perception reduces the urgency to evolve the codebase, particularly when existing implementations are considered sufficient for current needs. For example, P18 explicitly stated that language evolution is "not a concern" when deciding whether to evolve code. Maintenance efforts were instead driven primarily by changing requirements. The evolution was associated with adapting to new functional demands rather than adopting language features.

> *"Interviewer: Do you consider language evolution an important factor for evolving your code?*
>
> *P18: it's not a concern.*
>
> *Interviewer: I see. What factors would lead you to evolve your code?*
>
> *P18: in general or because of a change in a language?*
>
> *Interviewer: Could you explain why do you not consider language evolution a relevant factor to evolve your code? If it is not a relevant factor, could you tell me which factors do you consider important to evolve your code?*
>
> *P18: Changing your requisites, for sure. A new requirement that makes you refactor your code to be more general in order to accommodate this new requirement or a change. A requisite that used to be one way but is now different forces you to refactor your code. That's the main reason."*
>
> [**P18**]

As a condition, Perceived Relevance and Motivation shapes source code rejuvenation by influencing the perceived necessity of change. This condition moderates the phenomenon by reducing initiation and prioritization of rejuvenation efforts when language evolution is not considered sufficiently relevant to justify the associated effort and risk.

4.2.14 Contextual & Domain Factors

Contextual and Domain Factors captures the conditions imposed by the application domain that define the operational constraints under which software systems evolve. This includes domain-specific requirements such as performance sensitivity, reliability expectations, and criticality of system behavior. This concept compiles one factor: `application domain may influence SCR efforts`.

Participants described that the application domain directly conditions rejuvenation decisions by defining acceptable levels of risk and change. In domains involving low-level or critical systems, where failures can have significant impact, developers adopt a more cautious approach to source code rejuvenation. Even beneficial features may be postponed until their reliability is well established. For example, P12 emphasized that working at a low level changes the tolerance for change. They stated that "when there are low-level programming difficulties, that disturbs the chain of a whole product," and used the example of a flaw in the Linux kernel that "destroys everything from there."

> *"Usually, when there are low-level programming difficulties, that disturbs the chain of a whole product. Sometimes, even other products, for example: if there is a flaw in Kernel Linux, it destroys everything from there. It disturbs everything. So, at this level, you will adopt a new functionality if it is advantageous and if its impact is not big, of course."*
>
> [**P12**]

As a condition, Contextual and Domain Factors shapes source code rejuvenation by constraining adoption according to domain-specific risk tolerance and reliability requirements. This condition moderates the phenomenon by enforcing stricter evaluation criteria and delaying Source Code Rejuvenation in critical or sensitive contexts.

Across the identified conditions, source code rejuvenation emerges as a phenomenon strongly constrained by a combination of technical, organizational, and human factors that shape when, how, and whether source code rejuvenation occurs. External dependencies, compatibility requirements, and environment configurations embed systems within broader technical environments that limit autonomy and require coordinated evolution. At the same time, the unpredictability and interaction complexity in language evolution introduce uncertainty, increasing the risk of regressions, performance issues, and debugging difficulties derived from modern features. These risks are further amplified by fear factors and trust in legacy systems, which reinforce conservative decision-making, while the availability of automated and regression testing acts as a key enabling condition by increasing confidence in safely validating changes.

Practical constraints such as effort, cost, and project complexity reduce the feasibility of rejuvenation, especially when large version gaps or low-quality codebases are involved. Organizational structures, governance mechanisms, and perceived business relevance further regulate and often deprioritize source code rejuvenation efforts, while human factors and knowledge distribution influence teams' capability and willingness to adopt change. Domain-specific requirements and architectural characteristics further define acceptable levels of risk and flexibility for evolution.

Additionally, tooling-related conditions introduce a distinct layer of constraint, as the perceived lack of trust, limited transparency, and need for configurability restrict the extent to which automated support can be relied upon. Rather than functioning as fully autonomous enablers, tools are typically adopted in a controlled and selective manner, primarily providing guidance, suggestions, or partial support under human supervision. This reinforces a cautious approach to Source Code Rejuvenation, particularly in complex or critical systems, where developers prioritize predictability and control over automation.

Together, these conditions interact to moderate source code rejuvenation by delaying, limiting, or selectively enabling it, typically favoring incremental, low-risk, and human-controlled changes over large-scale or fully automated transformations.

4.3 Causes

Causes in this study consist of the factors that drive source code rejuvenation efforts. Source Code Rejuvenation is primarily triggered by the need to prevent system obso-

lescence, comply with language evolution, adapt to dependency and framework changes, simplify architectural complexity, and align with community-driven practices. We identified five concepts, including `System Obsolescence`(14), `Language Evolution Enforcement`(10), `Dependency and Architectural Simplification`(6), `Dependency and Framework Pressure`(5), `Community-Driven Adoption`(4). The following subsections provide a detailed description of each concept.

4.3.1 System Obsolescence

System Obsolescence captures a trigger factor, which developers rejuvenate code to avoid system obsolescence and ensure long-term viability. This concept compiles one factor: `Avoiding system obsolescence`.

Regarding system obsolescence, participants (C++ Surveys, P01, P01, P01, P03, P05, P06, P11, P11, P19, P22, P23, P24, and P26) framed source code rejuvenation as a response that is triggered when systems begin to exhibit signs of aging, such as becoming outdated, unsupported, or increasingly difficult to maintain. Rather than acting proactively, developers reported feeling motivated to rejuvenate code once the negative consequences of obsolescence become visible in practice, including reduced maintainability, compatibility issues, and growing technical constraints.

Several participants explicitly linked source code rejuvenation to long-term maintainability. For example, P11 discussed about to avoid the "trauma of migration" after falling several versions behind. In this sense, avoiding obsolescence is not merely reactive. It is also preventive. Regular updates reduce the accumulated cost of large source code rejuvenation efforts.

> *"One of the most important aspects of evolving code, as the language evolves, is the need for migrating [source code rejuvenation]. One notorious language for giving developers headaches because of its ways of evolving is Scala. Its philosophy is: whatever becomes deprecated today, tomorrow will be lost. Code becomes unusable from one version to another, and the cost associated with leaving your code behind by two or three versions of the language is very high, both in time and energy spent in updating it. As a developer, it is very important to your day-to-day work to follow along with the language evolution, because it diminishes the cost of migrating an old codebase, which is hard to deal with, and saves you time down the road. (. . .) It is critically important that you are evolving your codebase as languages evolve so that you get new features, performance gains, security bug fixes, and avoid the trauma of migration."*
>
> [**P11**]

In another example, the participant P23 similarly noted that legacy code written years ago often requires adaptation to support new language features, especially when deprecated constructs remain in use.

> *"The C++ project we have is always an ongoing project, it's never finished, and I think it will never be. But yeah, from time to time, we have to modify source codes that were written ten or maybe more years ago, by some people who are not working here right now, who left us maybe like five or more years ago. So, from time to time we have legacy codes that we have to understand, and maybe we have to modify them. For example, as we are working on some static analysis toolset, we have to support new features as the language gets a newer version. So, we have to support that. And we often have to modify the existing code base to support new features. As for me, if I deep-dive into a method that builds a cross-reference, or something like that. If I see that the code uses deprecated features, or features that are implemented in a better way now, or can be used in a better way. For example, an enhanced for-loop, or some smart pointers or something like that, I often modify the existing code base to the newer version of this language."*
>
> [**P23**]

This concept therefore captures a longitudinal concern. Developers and teams rejuvenate not only to gain immediate benefits, but to prevent progressive decay, loss of support, incompatibility with platforms, and increasing migration cost. Avoiding obsolescence functions as a strategic motive that extends beyond single feature adoption.

As a cause, the system obsolescence drives source code rejuvenation by acting as a trigger when systems become outdated, unsupported, or increasingly difficult to maintain. As language versions evolve and older constructs become deprecated or incompatible, developers are compelled to update their code to avoid rising migration costs and loss of support. This pressure intensifies over time, especially when systems fall multiple versions behind, triggering source code rejuvenation as a necessary action to restore compatibility, maintain operability, and reduce accumulated technical burden.

4.3.2 Language Evolution Enforcement

Language Evolution Enforcement captures how changes in programming language versions and implementations trigger developers to update their code to maintain compatibility and correctness. This concept compiles two factors, including `Language version support discontinuity` and `Bug fixes in the language implementation`.

Regarding discontinuation of language support, participants (Python Study [5], P07, P08, P11, P13, P17, and P29) consistently described discontinuation of language support as a strong trigger for source code rejuvenation. For example, P17 linked end-of-life status with security risks and corporate pressure.

> *"If that new feature allows you to give a new aspect for the product, or perhaps a performance or security improvement. Being in a corporate environment puts some pressure on it if you're using something that has surpassed its End-of-Life; it is not supported anymore, security breaches can appear and not be fixed. Many times*

> *we think something like: "Alright, I need to change this language version, I need to change this functionality because this version we use is not supported anymore.". Constantly, when you change versions, that feature evolves to other things, and you have to follow it."*
>
> [**P17**]

Although brief, the statement of P11 positions bug fixes in the language or platform as a direct motivation for updating applications. In their report, updating is not framed as optional enhancement but as routine maintenance. Bug fixes are listed alongside aging and loss of support, suggesting that defects and maintenance gaps accumulate over time.

> *"I'm always updating my application for several reasons, among them: bug fixes, and other problems arising from the aging of the code, and the lack of technical support over time."*
>
> [**P11**]

As a cause, Language Evolution Enforcement drives source code rejuvenation by imposing external pressures derived from language version discontinuity and accumulated defects in language implementations. When support for a version ends or bug fixes are introduced, developers are compelled to update their code to maintain compatibility, security, and correctness. During these efforts, developers may also perform source code rejuvenation to take advantage of improvements introduced by new language features. This cause directly triggers source code rejuvenation by transforming it from an optional activity into a necessary maintenance action, often forcing timely adaptation to evolving language versions and implementations.

4.3.3 Dependency and Architectural Simplification

Dependency and Architectural Simplification captures how reducing reliance on third-party libraries motivates the adoption of language-native or standard library features to simplify system architecture. This concept compiles one factor: `Minimizing dependency on third-party libraries`.

Regarding dependency and architectural simplification, participants (C++ Study, P13, and P22) also emphasized practical operational advantages of reducing external dependencies. For example, P22 illustrate this pattern in the context of C++. Before C++11, developers often relied on external libraries for capabilities such as multithreading or regular expressions. With the introduction of these features in the language standard, teams began replacing third-party solutions with standard implementations. According to the participant, this change reduces long-term maintenance problems, compatibility issues, and dependency management complexity.

> *"C++11 also introduced several standard libraries, like regular expression libraries. Before that, there were no regular expression libraries, you had to go for third-party libraries. And there were, again, the maintenance issues, the no compatibility issues. It was 10 years ago, and that project, only now, decided to replace these libraries. You see that it wasn't something urgent, but sooner or later, they decided "Okay, that's enough" it took too much work to maintain these third-party libraries."*
>
> [**P22**]

Similarly, P13 described adopting Kotlin's built-in serialization functionality instead of relying on widely used external libraries such as Jackson. In this case, the decision was influenced by the availability of a stable language-supported alternative and the perceived reliability of the official implementation. These findings suggested that the reduction of external dependencies appears as a recurring motivation and trigger for adopting modern language features.

As a cause, Dependency and Architectural Simplification drives source code rejuvenation by triggering changes when developers seek to reduce the complexity and maintenance burden associated with external dependencies. As language-native or standard library features become available, developers are prompted to replace third-party solutions with built-in alternatives. This transition is often triggered by accumulated maintenance effort, compatibility issues, or dependency management overhead, leading developers to perform Source Code Rejuvenation to simplify the architecture, reduce external reliance, and improve long-term system stability.

4.3.4 Dependency and Framework Pressure

Dependency and Framework Pressure captures how evolving frameworks and libraries drive developers to adopt modern language features to remain compatible. This concept compiles one factor: `Framework/libraries dependencies motivate source code rejuvenation efforts`.

Regarding dependency and framework pressure, participants (C++ Surveys, Python Study, P11, and P25) consistently describe the evolution of dependencies, such as frameworks and libraries, as key triggers for source code rejuvenation. In particular, updates in these dependencies often enforce or encourage the adoption of more recent language standards. As a result, when developers align their systems with more modern adopted and up-to-date language patterns, they also gain the opportunity to leverage modern language features through rejuvenation efforts. For example, C++ survey responses highlighted that Qt framework upgrades act as enablers for adopting modern C++ constructs, as transitions between Qt versions frequently require newer language standards. In this

context, the Source Code Rejuvenation of KDE systems was described as occurring alongside major Qt transitions, where dependency evolution not only imposed changes but also opened pathways for source code rejuvenation.

> *"Nowdays the Source Code Rejuvenation efforts on C++ go a bit hand in hand with another effort as we are in the middle of the transition between Qt5 and Qt6, which will also bring a major, source and binary incompatible major release of the KDE frameworks, that leads to a general Source Code Rejuvenation effort."*
>
> [**C++ Survey Responses**]

As a cause, Dependency and Framework Pressure drives source code rejuvenation by imposing compatibility requirements derived from the evolution of external libraries and frameworks. As these dependencies evolve, developers are compelled to update their code to align with new versions, often requiring the adoption of more recent language and library features. During these transitions, source code rejuvenation efforts are performed not only to maintain compatibility but also to leverage capabilities enabled by newer language standards. This cause directly triggers source code rejuvenation by tying code evolution to dependency upgrades, making rejuvenation a necessary step in keeping systems aligned with evolving frameworks.

4.3.5 Community-Driven Adoption

Community-Driven Adoption captures how discussions and practices within the developer community influence the adoption of modern language features. This concept compiles one factor: `Discussions of developers community influences the adoption of modern features`.

Regarding discussions of developers community influences the adoption of modern features, parcitipants of the C++ Surveys described informal community influence on source code rejuvenation. Participants reported that talks at Akademy and blog posts on Planet KDE help spread awareness of modern C++ features.

> *"Many KDE developers are also subscribed to Planet KDE, which sometimes contains blog posts on modern C++ features, for example from KDAB."*
>
> [**C++ Survey Responses**]

> *"People tend to start to rejuvenate the code of their own projects when they learn about the new possibilities [modern features], and if I remember correctly there were modern C++ talks at Akademy, which certainly helped adoption."*
>
> [**C++ Survey Responses**]

As a cause, Community-Driven Adoption drives source code rejuvenation by disseminating knowledge and influencing developers' awareness of modern language features through community interactions. Informal channels such as talks, blog posts, and mailing list discussions expose developers to new possibilities and shape their understanding of potential benefits and trade-offs. As developers learn about these features, they become more inclined to apply them in their own projects, triggering source code rejuvenation efforts. This cause also reflects that adoption is not immediate or automatic, but emerges from collective deliberation within the community, where decisions to evolve code are informed by shared experiences, discussions, and negotiated trade-offs.

The identified causes suggest that source code rejuvenation is not driven by a single factor but emerges from the interplay between external pressures (e.g., language evolution and dependencies) and internal system concerns (e.g., maintainability, longevity, and architectural complexity). Together, these drivers indicate that source code rejuvenation is often perceived as a necessary and continuous response to avoid obsolescence, ensure compatibility, and reduce long-term maintenance risks, rather than an optional improvement. Moreover, the influence of community practices and dependency highlights that source code rejuvenation decisions are socially and technically situated, frequently negotiated and constrained by broader development contexts. Overall, these findings position source code rejuvenation as a strategic activity embedded in ongoing softwares evolution.

4.4 Consequences

Consequences in this study describe the impacts of source code rejuvenation in legacy systems. We identified positive impacts when source code rejuvenation occurred and the impact from lack of source code rejuvenation in the software systems. We identified six concepts, including `Code Readability and Expressiveness`(125), `Reliability and Correctness`(33), `Performance and Efficiency`(25), `Maintainability and Testability`(22), `Development Productivity improvements`(21), and `Degradation effects`(4). The following subsections provide a detailed description of each concept.

4.4.1 Code Readability and Expressiveness

Code Readability and Expressiveness improvements captures how modern language features improve the clarity and expressiveness of source code, making it easier for developers to understand intent. The developers frequently associate source code rejuvenation with more concise and readable constructs, which contribute to improved comprehension and reduced cognitive effort during maintenance and evolution. This is

the most frequently consequence described by developers across our analysis. Participants highlighted three benefits from source code rejuvenation, including `Reducing cognitive load for newcomers`, `Improve code understanding`, and `Improve code expressiveness`.

Regarding reducing cognitive load for newcomers, participants (C++ Surveys and P19) framed source code rejuvenation as a way to make code more accessible to new developers. For example, P19 described source code rejuvenation as a continuous evolution process. According to the participant, code written in more recent language versions is more comfortable for newcomers to read and maintain.

> *"first of all, source code rejuvenation is a process of evolution. Ultimately, it is so that new developers who read the code in a more recent language version will be more comfortable maintaining that code. There is the issue of degradation; the code, over time, degrades; this degradation, with new versions, unit tests, and coverage, ceases. It is constant code maintenance."*
>
> [**P19**]

Related to improve uderstandability, participants and mining studies (C++ Study, C++ Surveys, JavaScript PR discussions, Python PR discussions, P01, P02, P03, P04, P05, P07, P10, P11, P12, P13, P14, P16, P17, P18, P19, P22, P23, P26, P27, and P29) repeatedly emphasized readability and clarity improvements. For example, P18 and P14 explicitly connected new constructs with simplified learning curves and elimination of problematic patterns such as callback nesting.

> *"I can recall a change in Java in which the* For *was significantly simplified. There was a way of doing For Collection and it was greatly simplified by the introduction of the colon [foreach]. This one I think I've used quite a bit. The main reason is to improve the code's readability, decrease verbosity, and be more concise."*
>
> [**P18**]

In another example, C++ Surveys respondents [3] frequently referenced lambdas, auto, and range-based loops as reducing boilerplate and making intent clearer. Also in C++ Study presented evidences of automated refactoring that introduced *auto-typed variables* to improve readability across many files. In the same way, in the Python PR discussions [5] some contributors highlighted *f-strings, type hints, and yield from* as clearer alternatives.

Finally, related to expressiveness improvement, participants (C++ Study, C++ Surveys, JavaScript PR discussions, JavaScript Study, Python PR discussions, and P01, P04, P05, P06, P07, P08, P10, P11, P12, P13, P14, P16, P17, P18, P19, P20, P24, P25, P26, P27, P28, P29) consistently linked language evolution to reduced verbosity, clearer structure, and more concise representations of intent. Common examples include *lambda*

expressions, range-based loops, auto type inference, annotations, records, async/await, coroutines, f-strings, arrow functions, and import statements.

For example, The participant (P01) frames language evolution not only as a mechanism for fixing problems, but as a driver of improvements in code expressiveness and long-term maintainability. This highlights that developers perceive source code rejuvenation efforts as having both corrective and qualitative benefits, where enhancements in expressiveness and maintainability are considered central to sustaining software over time.

In other examples, Empirical evidences from C++ Study [3] and JavaScript Study [4] illustrate large-scale transformations that replaced older constructs with more modern ones to achieve conciseness and clarity. Finally, Python PR discussions [5] shown an example of contributor suggesting to use f-string to improve readability.

> *"Commit f519ce07 of kid3 (commit date on September 29, 2018, and message Use auto where it improves the readability) corresponds to another example of code rejuvenation, introducing more than 400 auto-typed variables in 88 files."*
>
> [**C++ Study evidence**]

> *"Use f-strings for more readable and efficient string formatting."*
>
> [**Python PR discussions**]

As a consequence, Code Readability and Expressiveness Improvements captures how source code rejuvenation leads to clearer, more concise, and more expressive code, improving developers' ability to understand and work with the system. By adopting modern language features, developers reduce verbosity, clarify intent, and eliminate complex or outdated patterns, which lowers cognitive effort and facilitates onboarding of new contributors. These improvements also enhance long-term maintainability by aligning code with current language conventions and developer expectations. As a result, source code rejuvenation produces more understandable and expressive codebases.

4.4.2 Reliability and Correctness

Reliability and Correctness improvements captures how source code rejuvenation contributes to more robust and dependable systems. Participants highlight that adopting modern language features helps prevent bugs, enforce safer patterns, and reduce error-prone constructs, thereby improving overall system reliability and correctness. This concept presente two benefits, including `Help on bug prevention` and `Improve code security`.

Regarding help on bug prevention, participants (C++ Surveys, Python PR discussions, P03, P05, P08, P10, P16, P19, P20, P21, and P22) described modern language features are

framed as mechanisms for reducing errors and preventing defects. Developers highlighted features such as smart pointers, enhanced for loops, lambda expressions, final and override markers, and null safety systems. P22 explicitly linked smart pointers to reducing memory leaks. P21 associated evolution with data protection and memory safety. For example, P08 described null safety as preventing null pointer exceptions at compile time.

> *"It 's basically like this: When you access "type a, dot, a method", a type system will tell you that "this method does not exist in type a", the typing system will prevent you from doing something like this. But the standard typing system, like in the last version of Dart, won't prevent what's null. If you have something that assigns null so it can access the dot method, it will cause an error, it will show NullPointerException or "this method does not exist in type null". With null safety, you have to explicitly say "this object is type a", or "this object is type a and null, I want it to be null too". And then the typing system will check on the spot if you call a method of something that could be null, it will tell you that this method does not exist in type null, or "this is possibly null, please check it beforehand", and you will set if that thing exists."*
>
> [**P08**]

In another examples, Python PR discussions excerpts showed reviewers recommending keyword-only arguments, async/await, type annotations, and yield from to prevent incorrect calls, improve exception handling, and enforce type checking. Finally, P20 described range-based loops as preventing incorrect index usage.

Regarding security improvements, participants (C++ Surveys, P01, P03, P11, P12, P17, P20, P21, P22, P23, P25, and P27) described the improvement in code security as a recurring and structural consequence of the source code rejuvenation process, being reported by various participants as one of the main gains associated with the adoption of new language versions, modern features, and the replacement of legacy practices.

For example, participants indicate that source code rejuvenation directly contributes to the mitigation of critical technical vulnerabilities. For example, P21 describes severe problems in legacy systems, such as memory leaks and concurrency failures that compromise data integrity, highlighting that Source Code Rejuvenation reduces these risks by introducing safer mechanisms for memory management and synchronization. Complementarily, P03, P11, and P01 emphasize that the more recent versions of the languages incorporate security patches and bug fixes, making updates essential to avoid exposure to known vulnerabilities, especially when older versions are no longer supported.

> *"Well, in Java 8, especially if you compare it with Java 6, there are a lot of new resources, including file reading, that lambda thing, functional programming, and many performance upgrades too. So, we use these a lot, the collections, optional, also the upgrades in security. And the older versions start to lose support (. . .) We need to start planning [rejuvenate] from now."*
>
> [**P03**]

Another relevant aspect appears in the reports of P25 and P27, which highlight that modern features (such as records, immutability, authentication mechanisms, and encryption) introduce structural guaranties that reduce human errors and strengthen data protection.

> *"But I also wanted to emphasize the issue of performance and security as well. I think I've seen a lot of evolution in the security part, authentication, data encryption, privacy. This has evolved a lot in the last 5, 10 years, the issue of privacy, data information. So today there are languages and libraries that promote this type of improvement as well, which makes it very easy for the developer. We don't need to do things by hand."*
>
> [**P27**]

Finally, P17 and P12 emphasize that corporate environments and compliance requirements increase the pressure to adopt updated versions, as discontinued software tends to accumulate unpatched vulnerabilities, making Source Code Rejuvenation a necessary practice for maintaining operational security.

As a consequence, Reliability and Correctness Improvements captures how source code rejuvenation leads to more robust and dependable systems by reducing defects and enforcing safer programming practices. By adopting modern language features, developers prevent common errors such as memory leaks, null pointer exceptions, and incorrect API usage through mechanisms like type checking, null safety, smart pointers, and safer iteration constructs. Additionally, source code rejuvenation contributes to improved security by incorporating updated language versions with built-in protections, patches, and safer abstractions for data handling, authentication, and concurrency. As a result, Source Code Rejuvenation enhances both correctness and security, reducing technical vulnerabilities and increasing confidence in system behavior.

4.4.3 Performance and Efficiency

Performance and Efficiency improvements captures how source code rejuvenation enables improvements in execution performance and concurrency resource utilization. This concepts present two benefits, including: `Improve code performance` and `Improve concurrency/asyncronous programming`. Developers associate modern language features with more efficient implementations, particularly in areas such as concurrency and optimized standard libraries, leading to measurable gains in system performance.

In terms of performance improviments, participants (C++ Surveys, JavaScript Study, Python PR discussions, P02, P03, P11, P12, P17, P20, P23, P25, P27, and P29) frequently associated language evolution with measurable performance gains. For example, P23 explicitly referred to C++11 move constructors as reducing memory consumption in large

projects. C++Surveys respondents mentioned range-for loops generating better machine code and removing unnecessary copy constructions. Python PR discussions noted that f-strings are marginally faster than older formatting approaches.

> *"For performance updates or upgrades, using C++ 11 introduces the move constructor, which helps a lot with big projects with millions of lines of code, such as the ones we have. You'll gain a lot of memory by decreasing consumption."*
>
> [**P23**]

In another examples, developers frequently associate performance improvements in advances in language implementations, particularly in areas like garbage collection and asynchronous execution, were associated with tangible runtime gains across languages such as Ruby, Java, and JavaScript. For example, P20 stated that adoption depends on whether a feature makes code simpler, safer, or faster.

> *"Actually, the kind of evaluation that I do, and that I've seen people doing, is: if it gets simpler, safer or faster, you adopt it. You won't change the code to make it less simple. You wouldn't modify a code to make it slower - unless it gets a lot safer, simpler, and speed is not a concern on that specific part of the code. But for me, these are the three pillars: either it gets simpler, a better performance, or safer."*
>
> [**P20**]

Regarding the improve concurrency/asyncronous programming, participants(P13, P14, P21, P22, P24, and P27) consistently framed source code rejuvenation as a safer and more manageable concurrency models. P13 and P08 (through Kotlin coroutines and async/await discussions) connected asynchronous constructs with expressive and maintainable concurrency. P24 associated newer APIs with easier thread management compared to older Java versions. P27 linked asynchronous constructs to both performance and developer facilitation.

> *"The use of promise made it possible to make asynchronous calls, similar to what JavaScript does, which is something that I, as a Java programmer, envied for a long time. Because in Java 6, we had a thread problem, a competition problem, a decoration problem, and it was very difficult to do. This new API made it much easier, this thread management part."*
>
> [**P24**]

As a consequence, Performance and Efficiency Improvements captures how source code rejuvenation leads to measurable gains in execution performance and more efficient use of computational resources. By adopting modern language features, developers benefit from optimized constructs such as move semantics, improved iteration patterns, and more efficient string handling, which reduce memory consumption and enhance runtime

efficiency. Additionally, advances in language implementations—particularly in garbage collection and asynchronous execution—contribute to performance improvements across different languages. source code rejuvenation also enables more expressive and easier-to-use concurrency mechanisms, where features such as async/await, promises, and coroutines simplify thread management and asynchronous programming. As a result, source code rejuvenation enhances performance and efficiency, supporting faster execution and improved handling of concurrent operations.

4.4.4 Maintainability and Testability

Maintainability and Testability impacts captures how source code rejuvenation enhances the ease of maintaining and validating software systems over time. By adopting modern constructs and practices, developers report improvements in code organization, modularity, and testability, enabling more efficient updates and reducing the effort required to ensure correctness. In contrast, some participants report a negative impact from lack of source code rejuvenation. To this concept, we identified three factors, including `Improve code maintenance`, `Improve code testability`, and `Improve code standardization`.

Regarding improvements in code maintenance, participants (C++ Surveys, P01, P07, P11, P12, P14, P16, P19, P24, P25, P27, and P29) consistently described Source Code Rejuvenation as a means to improve long-term code maintenance. Modern language features were reported to reduce verbosity, clarify intent, and simplify system structure, making code easier to understand and evolve. Developers highlighted that constructs such as annotations, final, override, lambdas, and auto help reduce boilerplate, improve readability, and facilitate refactoring and testing. Additionally, participants emphasized that code quickly becomes legacy, reinforcing the need to adopt features that support future maintainability. For example, P07 emphasized that updates aim to improve long-term maintenance.

> *"The main one is implementing new functionalities and with this, the maintainability. The most important is how much this adds when making a new functionality. In general, there have been many functional language traits included in the mainstream interactive languages. I see that as a very positive thing, as you have concise, but at the same time legible and maintainable ways of doing things."*
>
> [**P07**]

Regarding improving code testability, P11 described how newer language features facilitate smaller methods, reduced coupling, and improved design. They emphasized that today's code becomes tomorrow's legacy and must support future testing and restructur-

ing. Testability emerges here as a downstream benefit of clearer structure and reduced verbosity.

> *"The code I write today will be legacy tomorrow, from then on I will take advantage of these new features to rebuild the work I've done in the past in a more succinct and maintainable way. This helps a lot in facilitating the extraction of functions and methods from legacy code as well as building tests for it."*
>
> [**P11**]

In terms of improving code standardization, participants (Python PR discussions, P03, and P24) associated modern language features with more consistent coding patterns. P03 linked newer constructs such as optional and streams to preventing common mistakes and promoting standardized practices. P24 described legacy systems that mix older and newer styles, arguing that standardization would ease onboarding.

> *"Especially in legacy systems, this would be very useful, because during the development of the system, if the system is very old, you will have this mix of old and new code. If everything was standardized, it would be much easier for someone to learn."*
>
> [**P24**]

Finally, in PR discussions referred to adding type annotations and updating return types to align with modern typing practices. This change was described as positive and consistent with current standards. Standardization here concerns reducing variation across codebases and aligning with updated conventions. Participants emphasized clarity and fewer recurring mistakes.

As a consequence, Maintainability and Testability Impacts captures how source code rejuvenation improves the long-term evolution and validation of software systems by making code more structured, consistent, and easier to work with. By adopting modern language features, developers reduce verbosity, clarify intent, and simplify system structure, which facilitates maintenance and refactoring. These improvements also support testability, as clearer and more modular code enables easier extraction of functions and construction of tests. Additionally, modern constructs promote more standardized coding patterns, reducing inconsistencies and making systems easier to understand, especially in legacy contexts where multiple styles coexist. As a result, source code rejuvenation enhances maintainability and testability by improving code clarity, reducing structural complexity, and aligning implementations with more consistent and modern practices.

4.4.5 Development Productivity Improvements

Development Productivity improvements captures how source code rejuvenation supports faster and more productive development processes. Developers describe that

modern features streamline implementation tasks, reduce boilerplate code, and simplify common operations, allowing developers to deliver functionality more efficiently. This concept compiles one factor: `Improve development productivity`.

In terms of improve in development productivity, Participants (C++ Surveys, Python PR discussions, P03, P04, P08, P11, P12, P14, P16, P19, P20, P24, P25, P26, and P27) repeatedly associated modern language features with reduced verbosity, syntactic simplification, and faster development cycles. P04 and P19 emphasized writing fewer lines of code to achieve equivalent functionality. P16 and P25 discussed annotations and records eliminating boilerplate code such as getters, setters, constructors, and DTO implementations.

For example, regarding Lambda expressions (C++Surveys, P04, P16) were frequently cited as improving conciseness and local reasoning. PyPullRequests highlighted type annotations enabling IDE inference and smoother workflows. Finally, P11 and P20 described productivity as multidimensional: faster implementation, easier maintenance, and improved design modularity.

> *"Lambda expressions come very handy in Qt's signal and slot connections, the code is more compact in such places...range based for-loops also allow for a more compact code. Same for the auto-keyword, especially when dealing with enums inside of class namespaces or so - this is where the auto-keyword can save a lot of space and a lot of typing work."*
>
> [**C++ Survey Responses**]

> *"Definitely! The languages add features that help with productivity, help you write simpler code, or even better: before, you needed to write many lines of code, so you could, for example, solve a problem, and now you don't need to. The language offers you ways of writing the same functionality with less code. So, it's really important for the developer to follow the language's evolution."*
>
> [**P4**]

Productivity is articulated through concrete constructs: lambda expressions, annotations, records, async/await, coroutines, and auto typing. The discourse emphasizes reduced manual effort, improved legibility, and acceleration of routine development tasks.

As a consequence, Development Productivity Improvements captures how source code rejuvenation supports more efficient development practices by reducing manual effort and simplifying implementation tasks. By adopting modern language features, developers write fewer lines of code, eliminate boilerplate constructs, and simplify common operations through mechanisms such as annotations, records, lambda expressions, and type inference. These features improve conciseness and local reasoning, making code easier to write and modify.

4.4.6 Degradation Effects

Degradation effects captures how the absence or delay of source code rejuvenation leads to the accumulation of technical debt in software systems. This concept compiles one factor: `Lack of source code rejuvenation may introduce technical debt`.

Regarding lack of source code rejuvenation, Participants (P08, P11, and P13) consistently framed delayed source code rejuvenation as increasing future cost and code complexity evolution. For example, P13 contrasted proactive updating with leaving legacy systems unchanged due to high current cost. The decision to postpone is explicitly linked to accumulated future burden.

> *"I work on a very fluid project that reacts well to changes. We always try to put the dependencies [frameworks/modern library features] on the latest stable versions. We try to pay attention to it, to avoid the cost of doing this later on. I also work on a legacy project to which the cost is quite high, so we decided to leave it the way it is because it would be costly."*
>
> [**P13**]

In another example, P11 emphasized that failing to track language evolution leads to painful and energy-intensive source code rejuvenation. The cost of skipping versions accumulates, particularly in technical environments with aggressive deprecation policies (e.g., Scala).

> *"One of the most important aspects of evolving code, as the language evolves, is the need for migrating. One notorious language for giving developers headaches because of its ways of evolving is Scala. Its philosophy is: whatever becomes deprecated today, tomorrow will be lost. Code becomes unusable from one version to another, and the cost associated with leaving your code behind by two or three versions of the language is very high, both in time and energy spent in updating it. As a developer, it is very important to your day-to-day work to follow along with the language evolution, because it diminishes the cost of migrating an old codebase, which is hard to deal with, and saves you time down the road."*
>
> [**P11**]

Finally, P08 stressed that postponing updates makes change progressively harder. Technical debt here is not abstract; it is temporal. Delayed source code rejuvenation increases source code rejuvenation cost, complexity, and risk over time. Previous studies [71, 72] in the literature state that technical debts "add more constraints on future maintenance tasks and make modification more difficult, costly, and unpredictable," and that the short-term advantage of technical debt is obtained "at the cost of extra work in the future." The case study at IBM defines technical debt as "increased cost of changing or maintaining a system" due to expedient shortcuts and states that these shortcuts have

long-term effects that can hinder evolution and maintainability, which may also impact source code rejuvenation efforts.

As a consequence, Degradation Effects captures how the absence or delay of source code rejuvenation leads to the accumulation of technical debt, increasing the cost, complexity, and risk of future changes. Participants described that postponing source code rejuvenation causes systems to fall behind language evolution, making source code rejuvenation more difficult and resource-intensive over time. As a result, the lack of source code rejuvenation increases maintenance burden and makes future adaptations more costly and unpredictable.

Across the identified consequences, source code rejuvenation play a central role in sustaining software evolution by systematically improving code readability, reliability, performance, development efficiency, and maintainability, while simultaneously mitigating long-term risks associated with legacy systems. Across these dimensions, source code rejuvenation is not perceived as a purely aesthetic or optional activity, but as a structural mechanism that reduces cognitive load, prevents defects, enforces safer programming patterns, and accelerates development workflows. Importantly, the findings also reveal a temporal dynamic: while source code rejuvenation produces immediate benefits in code quality and developer productivity, its absence leads to the accumulation of technical debt, increasing future migration costs, system complexity, and maintenance effort. This positions source code rejuvenation as a preventive and strategic practice, where continuous alignment with language evolution not only enhances current system quality but also avoids costly and disruptive source code rejuvenation efforts in the future, reinforcing its role as a key driver of long-term software sustainability.

4.5 Contingencies

Contingencies in this study describe the strategies, actions, and interactional practices through which developers respond to the conditions surrounding source code rejuvenation. These contingencies capture how rejuvenation is approached, negotiated, and operationalized in practice. We identified several concepts, including 💡 `Execution Strategy and Effort Management`(61), 💡 `Decision-Making and Strategic Evaluation`(52), 💡 `Selective, Risk-Aware, and Human-Controlled Use of Tool Support`(36), 💡 `Adoption Strategy and Timing`(31), 💡 `Knowledge, Experience, and Team Dynamics`(18), 💡 `Continuous Technical Debt Management`(3), 💡 `Managing Technical Constraints and Compatibility`(3), 💡 `CI/CD Infrastructure and Continuous Validation`(2). The following subsections provide a detailed description of each concept.

4.5.1 Execution Strategy and Effort Management

Execution Strategy and Effort Management captures how developers operationalize source code rejuvenation through planning, structuring, and organizing the execution of changes. Rather than applying rejuvenation in an ad hoc manner, developers adopt strategies to control effort, manage complexity, and reduce risk during the execution of changes. These practices reflect how source code rejuvenation is carried out in a controlled and structured way. To this concept, we identified three factors, including `Source code rejuvenation requires careful planning`, `Incremental source code rejuvenation effort`, and `Manual source code rejuvenation effort`.

Regarding careful planning, participants (P03, P06, P07, P08, P11, P12, P13, P19, P20, P22, P23, P25, P26, and P27) described rejuvenation as an activity that must be prepared in advance rather than executed spontaneously. This includes anticipating impacts, organizing the sequence of changes, and ensuring that necessary conditions—such as validation and compatibility—are considered before execution. In practice, planning functions as a strategy to avoid unintended consequences and to coordinate changes in a controlled manner. For example, P19 described a process of studying language updates and evaluating whether they fit the current maintenance task.

> *"First of all, before doing the tasks, I take what the newest version of the language offers. I try to understand the problem I am working on, and I confirm if I can use the features added to the language can be used in the maintenance I am doing. It's looking at the problem, investigating the latest language features, and understanding the context in which I'm working make it possible to apply those latest features if they suit what I'm doing at the moment."*
>
> [**P19**]

In terms of incremental efforts, participants and studies (C++ Study, C++ Surveys, JavaScript PR discussions, JavaScript Study, Python PR discussions, Python Study, P03, P04, P07, P08, P09, P14, P20, P21, P22, P23, and P27) report that they adopt a strategy of performing rejuvenation in small, manageable steps rather than large, disruptive changes. This incremental approach allows changes to be introduced gradually, making it easier to monitor their effects and reduce the risk of introducing issues. By structuring rejuvenation as a sequence of smaller updates, developers maintain control over complexity and facilitate safer integration. For example, in C++ Survey responses, a participant argue that source code rejuvenation is naturally incremental for him.

> *"More recent versions of the C++ standard offered smaller, more incremental changes. Thus, the modernization of code has also become more incremental for me. If I see code that would benefit from newer features while working on it, I take a note and do a follow-up commit to the change that I actually worked on. I'm not using any tool specifically for these smaller modernisations."*

[**C++ Survey responses**]

Furthermore, two mining studies [3, 4] revealed concentrated rejuvenation episodes: in C++, commits replaced more than 200 and 300 Qt `foreach` occurrences with range-based `for`, and another introduced more than 400 `auto`-typed variables across 88 files, explicitly supported by `clang-tidy`. In JavaScript, large commits introduced more than 2038 `const` declarations and 152 `let` declarations to replace `var`, as well as dozens of arrow functions and other modern features. Therefore, while developers commonly perceive rejuvenation as incremental, opportunistic, and controlled, mining evidence shows that such practices can also appear as large-scale, concentrated source code rejuvenation episodes when recurring outdated constructs are systematically replaced across the codebase.

> *"Commit 2e508ac of kdenlive (commit date on April 20, 2017, and message Use modern for) replaces more than 200 occurrences of the foreach Qt statementk by the new range-based for statement in 59 files."*
>
> [**C++ Study evidence**]

> *"Commit 5c28a3e5 of labplot (commit date on December 3, 2017, and message Replace foreach macros with range-based for loops in many places) is another example of a large transformation that replaces more than 300 occurrences of the foreach macro by the modern range-based for statement."*
>
> [**C++ Study evidence**]

> *"For instance, a large effort to adopt modern JavaScript in commit 1166d10e4 introduces more than 2038 Const Declarations and 152 Let Declarations to replace Var Declarations . . . Is also part of a test case from Jasmine project, focusing on a scenario where a fallback to setTimeout occurs."*
>
> [**JavaScript Study evidence**]

However, the mining studies [3, 4, 5] further support developers' perception that source code rejuvenation is frequently incremental, heterogeneous, and feature-dependent. While the C++ and JavaScript studies identified concrete large-scale rejuvenation commits, the broader quantitative evidence shows that modern feature adoption is uneven across projects, features, and time. In C++, features such as lambda expressions, `auto`-typed variables, and range-based `for` are widely used, but their distribution is not uniform across KDE projects. In JavaScript, highly adopted features such as `const`, `let`, arrow functions, async declarations, and template strings coexist with features that remain rare adopted. In Python, the adoption timelines also vary substantially, with some features reaching generalized adoption shortly after release and others taking several years to enter a broader growth phase.

> *"However, the KDevelop project alone contains 4441 occurrences of auto-typed variables (13.36% of the total). We observe the same lack of uniformity for lambda expressions and range-based for."*
>
> [**C++ Study**]

> *"In particular, the Handsontable project alone accounts for 18569 occurrences of Arrow Function Declarations, which represents 16.9% of the total, and 21434 occurrences of Const Declarations, making up 9.8% of the total."*
>
> [**JavaScript Study**]

> *"For example, Formatted String Literals (f-strings) exhibit a median of 31 occurrences per project, while Variable Annotations and Function Annotations present medians of 1 and 3, respectively. In contrast, most other features display a median of zero. The distribution is also highly uneven, as reflected by the large standard deviations. For instance, in the sentry project, Function Annotations reach a maximum of 26,605. We also observe a similar lack of uniformity in the usage of Variable Annotations and f-strings across projects."*
>
> [**Python Study**]

These findings align with the interview evidence that developers often address recurring rejuvenation opportunities gradually, during maintenance or through controlled migration steps, rather than as a single uniform process. Thus, although repository mining reveals occasional large-scale rejuvenation episodes, the overall empirical pattern suggests that source code rejuvenation commonly unfolds through incremental, localized, and context-sensitive adoption of modern language features.

Concerning manual effort, participants (C++ Surveys, P01, P02, P03, P11, P12, P23, P28, and P29) described that certain rejuvenation activities require direct human intervention. These efforts involve understanding the code context, applying changes carefully, and validating outcomes beyond what can be automatically handled. In practice, manual execution reflects the need for contextual reasoning and careful handling of changes that cannot be fully delegated to automated processes. For example, P28 argues that prefer do updates manually than automated source code rejuvenation efforts.

> *"Nowadays I don't usually use them [automated tools], it's very rare to be using any of them. But because sometimes when I tried to use them, they brought me results that, when I really went to implement, I had to change a lot of things, so I prefer to do it manually. But I imagine that in the near future they will be part of the development. Even more than they are today."*
>
> [**P28**]

As a contingency, Execution Strategy and Effort Management captures how developers structure the execution of source code rejuvenation through planning, incremental application, and manual intervention. These practices shape the phenomenon by enabling

controlled and deliberate execution of changes, reducing the likelihood of disruption while managing effort and complexity. As a result, source code rejuvenation is carried out as a structured and effort-aware process, guided by preparation, staged execution, and human judgment.

4.5.2 Decision-Making and Strategic Evaluation

Decision-Making and Strategic Evaluation captures how developers operationalize source code rejuvenation through deliberate evaluation practices that balance cost, risk, effort, and expected benefits. Rather than acting automatically, developers engage in structured decision-making processes in which rejuvenation is assessed, negotiated, and justified before being performed. These practices reflect how source code rejuvenation is strategically evaluated within technical and organizational constraints. To this concept, we identified three factors, including 🏷 `Company/Team decision-making`, 🏷 `Analyse costs/benefits before rejuvenate`, and 🏷 `Recommendation to stick with LTS versions for complex systems`.

Regarding comany/team decisions, participants (C++ Surveys, P01, P03, P05, P07, P20, P29) described rejuvenation as a collective and structured process rather than an individual action. Decisions are typically proposed by developers, discussed within teams, and validated through organizational or project-level criteria. In practice, this involves coordination across roles, alignment with internal processes, and, in some cases, approval by leadership. This strategy reflects how rejuvenation is mediated by governance structures and shared decision-making practices. For example, P1 emphazed that team decides when evolve for a new language version.

> 💬 *"Well, in my experience, this update is usually done in corporate terms, so the corporation, the team decides that it would be best to follow the language's evolution, and then we get the permission, or better said, reach the understanding that these systems should be updated in that new version of the language."*
>
> [**P01**]

In terms of costs/benefits evaluation, participants (C++ Surveys, JavaScript PR discussions, Python PR discussions, P02, P03, P04, P05, P07, P08, P09, P12, P13, P14, P17, P18, P20, P21, P22, P23, P25, P26, P29) consistently described evaluating whether rejuvenation is worthwhile before acting. This includes assessing expected gains such as improved readability, performance, or safety against costs such as time, effort, instability, and the need for revalidation. In practice, developers consider factors such as time constraints, competing priorities, dependency compatibility, and potential risks. Rejuvenation is performed when perceived benefits clearly justify the associated costs under

the specific conditions of the project. For example, P22 identified specific benefits that justify change, such as improved safety, reduced external dependencies, or readability or the learnability.

> *"I think the major factors are very similar for all programming languages. So, basically, you calculate risk, you calculate gains, you calculate the effort you have to put into it. The major triggers are safety; how can I say, the cut of external dependencies, I think that it doesn't matter if you are speaking about a Java or a Python program, you don't want to use unnecessary third-party libraries, which can be changed, which can be outdated, which can be not supported. That's a problem for Java, Python, or C++, also because, of course, the safety and the readability or the learnability of the codes are similar factors. I see the factors are the same, the thresholds are different."*
>
> [**P25**]

For recommendation to stick with LTS versions, participants (P11 and P25) adopt a conservative strategy in which long-term support versions are preferred in contexts where stability is critical. This practice reflects a prioritization of reliability and predictability, particularly in systems where changes may have broader impact. By relying on versions that are more established and supported over time, developers reduce exposure to instability and avoid introducing unnecessary risk. For example, P11 explicitly stated using Java 11 LTS for mission-critical projects. The participant contrasted LTS versions with recently released versions that introduce new features but may still accumulate bug fixes over time.

> *"I'm currently working on Ruby version 2.6 because our project needs it. For new projects, I recommend the 3.0 version. I use Java version 11 for mission-critical projects since the LTS version, which is a long-time service unless I'm mistaken. Anyway, they are the ones that have an extended warranty from companies like Oracle, but for those who want it for projects that are not mission-critical, meaning that they don't have sensitive data, then Java version number 15 is the latest version with several cool features like Record, among others. It depends on many characteristics of a project to choose a version. For mission-critical software, it's not so good to use the latest version. It's more interesting to use a more mature version. I avoid using recently released versions, for example, a 3.0.0 version, because it's so new, and then you will see there that the bug fixes will increase significantly, and as it gains maturity: 3.1; 3.2; 3.1.5; 3.1.10; 3.1.20, you will find that the language is much more robust."*
>
> [**P11**]

As a contingency, Decision-Making and Strategic Evaluation captures how developers structure and regulate source code rejuvenation through evaluative and decision-oriented practices. These strategies shape the phenomenon by ensuring that rejuvenation is not performed indiscriminately, but instead emerges from negotiated trade-offs involving cost,

risk, effort, and system constraints. As a result, source code rejuvenation becomes a deliberate and context-sensitive process, guided by collective decision-making, systematic evaluation, and differentiated handling of changes based on their characteristics.

4.5.3 Selective, Risk-Aware, and Human-Controlled Use of Tool Support

Selective and Human-Controlled Use of Tool Support captures how developers operationalize source code rejuvenation through selective, controlled, and human-mediated use of tooling. Instead of relying on fully automated transformations, developers consistently describe the tools as a support mechanism that assists in identifying opportunities and partially automating limited tasks, while maintaining human responsibility for decision-making and validation. To this concept, we identified three key factors: `Reliance on static analysis to detect source code rejuvenation opportunities`, `Using tooling to support and partially automate source code rejuvenation`, and `LLM-based support for source code rejuvenation`.

In terms on rely on static analysis tools to detect potential rejuvenation opportunities, participants (C++ Surveys, Python PR discussions, P04, P05, P10, P11, and P20) emphazed that tools such as Clang-Tidy, Clazy, Sonar, IntelliJ inspections, and language-specific linters are used to identify outdated constructs, suggest modern alternatives, and highlight areas of the code that could benefit from source code rejuvenation efforts. For example, P10 explained that linters often include auto-fix capabilities that help update legacy code to better practices, although the participant acknowledged that human review remains necessary because automated changes may alter behavior.

> *"emphasized the role of linters integrated into development pipelines. According to the participant, linters encode community-recommended practices and can enforce these practices automatically during development. In the described workflow, linting rules are treated as build errors rather than warnings, meaning that code that violates these rules cannot pass the build process. This practice encourages developers to align their code with updated language practices and helps detect issues when new language features or standards are introduced."*
>
> [**P10**]

Another example, in Pull request discussions, developers showed practical examples of these suggestions being followed, such as replacing string formatting with f-strings in Python projects.

> *"I will use f-strings and also move to a variable before fixing lint."*
>
> [**Python PR discussions**]

Additionally, tooling is utilized to facilitate and partially automate Source Code Rejuvenation, especially for transformations that are well-defined, low-risk, and mechanically verifiable. Participants (C++ Survey, P07, P08, P23, and P26) stressed that automation is usually only used when the results are known and can be easily checked. On the other hand, more complicated changes that need semantic understanding or architectural reasoning are always done by hand.

For instance, P23 said that tools should be seen as an extra layer of help, and developers should actively review and guide changes. P08 said that tools are helpful for finding transformation and rejuvenation opportunities. However, some parts of the codebase still needed manual work.

> *"Yeah, mostly I use JetBrain products, such as IntelliJ IDEA, Rider, Webstorm, and so on. These tools often advise you to use a newer feature or a simplified thing that will end up in a newer feature. And we previously used several code analysis tools with some rules introduced into them. For example, the auto keyword. The repository for that is on GitHub and it converts your code to use as much Auto or AutoReference or AutoPointer as possible. I did also check those tools, although, for our real project, I did not use them."*
>
> [**P23**]

> *"When I was updating from Python 2 to 3, it wasn't on that project that we did it. It was in a project that we had decided not to update, and I used a script that stated it could do this, though not 100%. It performed the update where it was possible, and then you had to change some other parts manually."*
>
> [**P08**]

This selective use of tooling reflects a human-in-the-loop strategy, where developers maintain control over the rejuvenation process and rely primarily on manual efforts to perform source code rejuvenation, as discussed in 4.5.1. Although automated tools can support well-defined and low-risk transformations, participants consistently emphasized that most rejuvenation activities require direct human intervention, including understanding code context, applying changes carefully, and validating outcomes. In practice, automation is typically applied in a partial and assistive manner, with developers reviewing, adapting, and completing transformations to ensure correctness and suitability. As a result, tool support is integrated as a complementary mechanism to reduce effort and highlight opportunities, rather than as a replacement for developer-driven rejuvenation work.

Regarding LLM-based tools, participants (P26, P27, P28, and P29) perceive AI tools not only as transformation mechanisms, but also as comprehension-oriented support for source code rejuvenation. Participants P26, P27 reported using LLMs to understand differences between older and newer language idioms, to interpret confusing legacy code,

and to obtain suggestions before manually applying changes. In these cases, LLMs support sensemaking and decision-making rather than fully automating migration.

> *"And then I had to do this process of putting a part of the code [putting part of the coding in a LLM prompt], it wasn't everything. I usually took specific parts of the code to understand what was different, and from there I replicated it in other places. So it was more of a way to understand the changes between the versions, and from there I did the migration. I didn't use it as a migration tool, that is, I didn't put all the code there, it migrated and I copied it. It was more of a way to understand the specificities of each version."*
>
> [**P26**]

Also, P27 emphasized the perceived benefits of LLMs, associating them with substantial productivity gains, faster ticket resolution, and support for broader development activities such as test creation, integration, pipeline execution, deployment, and environment setup. In this view, LLMs reduce manual implementation effort and shift developers toward a more business-focused orchestration role.

> *"I feel that the quality of the code is improving. The quality of test creation is also improving. Just to complement what I was saying before. Today, automated tests, integration tests, the whole process of running the pipeline, checking if there is any error. All this structural part of the business is making it a lot easier, the LLM."*
>
> [**P27**]

However, P28 and P29 emphasized the limitations and risks of LLM-based support. P28 noted that LLM-generated results often require substantial adaptation before implementation, while P29 argued that LLMs are more useful for simple tasks than for complex development work. P27 similarly stressed that AI-generated code still needs human refinement to improve readability and align with team conventions, reinforcing the human-in-the-loop character of tool-supported source code rejuvenation.

> *"Nowadays I don't usually use them, it's very rare to be using any of them. But because sometimes when I tried to use them, they brought me results that, when I really went to implement, I had to change a lot of things, so I prefer to do it manually."*
>
> [**P28**]

The participants also associated LLM use with risks beyond technical correctness. P28 and P29 raised concerns about exposing confidential code, banking data, business logic, and internal project information to external AI services. In addition, P27 and P29 suggested that trust in LLM-generated output is mediated by developer experience: while some developers may rely heavily on these tools, more experienced developers tend to

treat them as assistants that require careful review. Excessive reliance, especially by less experienced developers, was perceived as potentially harmful to foundational programming skills.

> *"Because we know that quality is not the preference of most developers. In fact, it's just delivering. So, I think ok, I think it's cool who uses it, to a certain extent. Why? Because if you want to stay mediocre, you will always use it for everything. If you know how to use it and get the best out of it, no. You can really take advantage of it. You go to work with Junior [a less experienced developer], for example, and you have to improve the code all the time. Very few times will you have a really good code coming out of the AI."*
>
> [**P29**]

As a contingency, these findings indicate that developers adopt a cautious, risk-aware, and human-controlled approach to tool usage in Source Code Rejuvenation. While traditional tooling is primarily used to support and partially automate well-defined and low-risk transformations, the overall rejuvenation effort remains predominantly manual, guided by human evaluation, contextual reasoning, and validation. Even with the rise of LLM-based tools that provide more interactive and context-aware support, current evidence suggests that these approaches can complement traditional syntactic and rule-based techniques in important migration tasks, such as source code rejuvenation. This is particularly visible in their ability to leverage broader contextual information and assist in automating a larger portion of required edits.

However, these improvements do not eliminate the need for human oversight. Instead, LLM-based support is increasingly used as an enhanced form of assistance, contributing to higher migration coverage, reduced manual effort, and support for more complex transformations. At the same time, these approaches still require review, testing, and contextual validation to ensure correctness and avoid unintended changes. Thus, while LLM-based tools extend the capabilities of automation in source code rejuvenation, developers remain responsible for validation and final decision-making.

4.5.4 Adoption Strategy and Timing

Adoption Strategy and Timing captures how developers strategically decide when and how to introduce modern language features into existing systems. Rather than treating source code rejuvenation as an immediate or uniform process, developers operationalize rejuvenation through delayed adoption, staged experimentation, opportunistic integration during ongoing work, and, in some cases, early experimentation with pre-release features. These practices reflect how adoption is managed under constraints of stability, maturity,

and contextual readiness. To this concept, we identified four factors, including `Delayed adoption of new versions`, `Reactive approach`, and `Earlier adoption of modern features (even before their official release)`.

Regarding delayed adoption of new versions, participants and study (C++ Surveys, JavaScript PR discussions, JavaScript Study, Python Study, P01, P09, P11, P13, P20, P22, P24, P26, P27) described a deliberate strategy of postponing adoption until new versions become stable, mature, and supported by the systems. Rather than immediately integrating newly released features, participants reported waiting for consolidated standards, improved tooling support, and reduced uncertainty. This practice reflects a risk-aware approach in which rejuvenation is scheduled only after sufficient validation in real-world contexts, aligning adoption with organizational readiness and minimizing disruption.

For example, P20 described intentionally staying behind newer C++ standards to ensure stability. P22 reiforced it and explicitly mentioned "three, four years delays" before using new C++ features.

> *"Typically, we use three, four years delays. Currently, C++ 20 is one year old. We just started to try out C++ 20, so I don't expect that we will apply them in the next two years. Perhaps three or four years after the feature appeared, we started to use them. That was a long-term experience. Our C++ 11 features, we started using C++ 13, or 14, or something like that."*
>
> [**P22**]

In terms of reactive approach, participants (C++ Surveys, Python PR discussions, P01, P22, and P25) consistently framed rejuvenation as embedded within other maintenance or development activities rather than as a standalone effort. Adoption of modern features typically occurs when code is already being modified due to bug fixes, feature implementation, refactoring, or external pressures such as deprecations and loss of support. This opportunistic strategy reduces the cost and risk of isolated source code rejuvenation efforts by coupling rejuvenation with necessary code changes. For example, in C++ Surveys responses, the participants emphazed:

> *"it's more continuous and low intensity, new code tends to use more modern c++ features, old code is ported more when it really needs to be rewritten."*
>
> [**C++ Survey responses**]

Finally, about earlier adoption of modern features, a participant and studies (C++ Study, JavaScript Study, Python Study, and P13) described a proactive strategy of experimenting with features during alpha, beta, or partially supported stages. In these cases, developers adopt modern constructs before formal standardization when they are closely

following language evolution and are willing to work with evolving implementations. This practice reflects an exploratory mode of rejuvenation, often associated with advanced users or projects seeking to remain at the forefront of technological change.

For example, P13 report that they try to follow language evolution and adapt their code:

> *"We started using it in the experimental version, so we adapted some of our code as it progressed from experimental to a stable version with the stable API."*
>
> [**P13**]

In addtion, the mining studies findings (C++, JavaScript, and Python) presented evidence that developers adopt modern features before their official release. For example, in the JavaScript study:

> *"Arrow Function Declarations was released on Firefox and Firefox for Android 4. Other features, such as Const Declarations, were added to Safari in 2011 and Opera in 2006 5. "*
>
> [**JavaScript Study**]

As a contingency, Adoption Strategy and Timing captures how developers actively manage the temporal and operational dimensions of source code rejuvenation through different strategic approaches. These practices shape the phenomenon by balancing risk, learning, and effort: delayed adoption prioritizes stability, reactive integration aligns rejuvenation with existing workflows, and early adoption promotes exploration of new capabilities. Together, these strategies demonstrate that source code rejuvenation emerges as a context-dependent and strategically enacted process, rather than a uniform or immediate transformation.

4.5.5 Knowledge, Experience, and Team Dynamics

Knowledge, Experience, and Team Dynamics captures how developers operationalize source code rejuvenation through practices shaped by shared knowledge, prior experience, and team interaction. Rather than being performed in isolation, source code rejuvenation is influenced by how developers exchange knowledge, coordinate decisions, and leverage existing expertise within the team. These practices reflect how collaboration and experience structure both the adoption and execution of rejuvenation efforts. To this concept, we identified three factors, including `Use code review to recommend adoption of modern features`, `Team cohesion`, and `Prior knowledge reduces adoption effort`.

Regarding the use of code review to recommend adoption of modern features, participants (avaScript PR discussions, Python PR discussions, and P19) described code review as a practical mechanism through which suggestions for adopting newer constructs are introduced and discussed. In practice, reviewers propose changes, recommend alternatives, and guide adoption during the evaluation of contributions. This positions code review as a collaborative strategy that embeds rejuvenation into existing development workflows. For example, in Pull request discussions provided multiple examples of this practice. Reviewers recommended replacing older constructs with modern alternatives such as optional chaining in JavaScript or type annotations in Python.

> *"Maybe you can use the optional chaining operator for things like these? E.g. instead of having an 'if' have 'this.chatview.emoji_dropdown?.toggle()'"*
>
> [**JavaScript PR discussions**]

> *"please add return type annotations to all functions and methods: def etots(self) -> np.ndarray"*
>
> [**Python PR discussions**]

In terms of team cohesion, participants (P01, P14, P20, and P24) highlighted how alignment and interaction within the team influence how rejuvenation is carried out. Shared understanding, communication, and coordination among team members shape how decisions are made and how changes are introduced. This reflects a practice in which rejuvenation depends on collective agreement and coordinated action rather than individual initiative. For example, P24 described situations in which developers advocate for modernization during project planning, proposing upgrades when they see potential improvements in maintainability or development efficiency.

> *"I started a project and I voted for us to enter Java 17, precisely because I saw that we would have a lot of DTO and that this part of Java 17 would make it much easier."*
>
> [**P24**]

Concerning prior knowledge reducing adoption effort, developers described how familiarity with language features and prior experience facilitate the introduction of changes. When developers already understand newer constructs, the effort required to apply them decreases, making adoption more feasible in practice. This highlights a strategy in which accumulated knowledge directly supports the execution of rejuvenation activities. For example, responses from the C++ survey indicate that prior familiarity with conceptually similar constructs can significantly ease the adoption of modern language features. One participant noted that experience with Qt's Q_FOREACH macro facilitates the transition to C++ range-based for loops, as both constructs share similar iteration semantics.

> *"In my opinion it is to have easier to read and simpler code. E.g. if you have some long type (especially with C++ template types it might be harder to read than just auto). range-for idea is not that different from Qt's Q_FOREACH which was already used a lot, so it makes sense to move to C++ standard version (and range-for generates slightly better machine code than Q_FOREACH). Lambda expressions are again useful to write simpler code, especially with Qt signals and slots."*
>
> [**C++ Survey responses**]

As a contingency, Knowledge, Experience, and Team Dynamics captures how developers rely on collaboration, shared understanding, and prior expertise to carry out Source Code Rejuvenation. These practices shape the phenomenon by embedding rejuvenation within team interactions and learning processes, enabling coordinated adoption and reducing effort when knowledge is already available. As a result, source code rejuvenation is influenced by how teams communicate, share expertise, and build on existing experience during both decision-making and implementation.

4.5.6 Continuous Technical Debt Management

Continuous Technical Debt Management captures how developers operationalize source code rejuvenation as an ongoing strategy to manage and reduce accumulated technical debt in evolving systems. Rather than treating technical debt as a static problem or addressing it only in isolated efforts, developers incorporate debt reduction into continuous development practices, using source code rejuvenation as a mechanism to improve code quality and mitigate future maintenance costs. This concept compiles one factor: `Reducing technical debt`.

Regarding reducing technical debt, the participant (P3) described rejuvenation as a deliberate strategy to address accumulated deficiencies in the codebase over time. In practice, this involves source code rejuvenation efforts, updating outdated patterns, and aligning the code with more modern language features to improve structure and maintainability. This strategy reflects a forward-looking approach, where developers act not only to fix immediate issues but also to prevent the escalation of maintenance complexity. By continuously addressing technical debt through Source Code Rejuvenation, developers aim to reduce long-term costs, improve code sustainability, and maintain system quality as the software evolves.

> *"For example, today, the projects that we work with are not up to date, they have many older versions' frameworks, like two to three versions older. In this case, we need a detailed plan to evaluate the impacts, how much time would it take to migrate, if we have enough people to execute it, to test it, to put up an introduction. It depends a lot on the company. I have worked for three different companies here in Quebec, and the one I'm working for right now is pretty obsolete in terms of technology, so*

any migration process, or any new thing needs very detailed planning, containing the changes' impact. Before this company, I was in another one that had a much better developed method. We worked on an 80/20 scale, being 80% for new features, and 20% for technical debt of the project. We always worked on the languages' framework updates, performance improvement, error tracking, logs, this kind of thing. So, it depends a lot on the company's working process."

[**P3**]

As a contingency, Continuous Technical Debt Management captures how developers embed rejuvenation into ongoing efforts to control codebase degradation. This practice shapes source code rejuvenation as a continuous and preventive process, where source code rejuvenation is used to systematically reduce accumulated complexity and avoid future maintenance burdens. As a result, source code rejuvenation contributes to sustaining code quality over time by integrating technical debt reduction into regular development activities.

4.5.7 Managing Technical Constraints and Compatibility

Managing Technical Constraints and Compatibility captures how developers operationalize source code rejuvenation by adapting their strategies in response to technical limitations. Rather than applying changes freely, developers adjust, defer, or reshape rejuvenation efforts to accommodate constraints imposed by existing code, tooling, or compilation requirements. These practices reflect how source code rejuvenation is carried out under restrictive technical conditions. To this concept, we identified two factors, including 🏷 `Legacy constraint removal enabling source code rejuvenation` and 🏷 `Disabling warnings to keep systems compiling`.

Regarding legacy constraint removal enabling, developers described that certain rejuvenation efforts only become feasible after removing or resolving existing constraints in the codebase. In practice, legacy elements or prior limitations can prevent the introduction of newer constructs, requiring preliminary adjustments before changes can be applied. This reflects a staged strategy in which constraint removal is a prerequisite for enabling rejuvenation. For example, In JavaScript PR discussions, contributors explicitly frame source code rejuvenation as part of a broader shift away from legacy platform support, arguing that deprecating older APIs goes hand in hand with encouraging the adoption of native language features (e.g., ES6 classes).

💬 *"video.js Since Video.js components are now un-transpiled ES6 classes extend() needs to be able to handle them since videojs.extend(videojs.getComponent('Component') {...}) is the most common usecase for this method. This string check is apparently the only way to detect ES6 classes. It isn't future proof and supposedly will not work on some older platforms but that is likely not a huge concern since we are*

1) dropping support for most older platforms and 2) deprecating this method and encouraging the use of native classes instead."

[**JavaScript PR discussions**]

In terms of disabling warnings to keep systems compiling, developers reported adopting temporary adjustments to maintain build continuity when introducing changes. When newer language standards or compiler updates expose warnings or incompatibilities, instead of immediately resolving all issues, developers may disable warnings to allow the system to continue compiling. This practice reflects a trade-off strategy, where maintaining operational continuity takes precedence while issues are deferred for later resolution. For example, P20 argue about experienced of migrate to C++ 17 that they needed to disable the compiler's warnings.

"For example, when trying to migrate to C++ 17 in that other company, they had to disable the compiler's warnings. Because there were incorrect things in use that were already that way in version 14, but the compiler didn't warn us back then. Then the version 17 compiler started to not accept this kind of thing anymore, and to issue warnings. What also happens when you evolve language on C++, is that the standard starts demanding warnings. For example, there's a situation where you use the language incorrectly and this causes issues, but they didn't know about it. That way, as you use the language, they start to notice that this and that could be warned by the compiler. So, in the next version, they will include it in the standard, that this situation needs to be reported by the compiler. And then, a warning you didn't know starts showing up. It normally happens when there's an incorrect use of the language, when you're messing up."

[**P20**]

As a contingency, Managing Technical Constraints and Compatibility captures how developers adapt source code rejuvenation to operate within technical limitations by applying workarounds, sequencing changes, and temporarily relaxing constraints. These practices shape the phenomenon by enabling progress under restrictive conditions, while also introducing deferred work and partial solutions. As a result, source code rejuvenation emerges as an adaptive process in which developers continuously adjust their strategies to align with the technical realities of the system.

4.5.8 CI/CD Infrastructure and Continuous Validation

CI/CD Infrastructure and Continuous Validation captures how developers operationalize source code rejuvenation through continuous integration and validation practices embedded in automated pipelines. Rather than serving only as supportive infrastructure, developers described CI/CD pipelines as a central operational mechanism that governs

whether rejuvenation can be safely introduced. This concept compiles one factor: `Relevance of CI/CD to source code rejuvenation efforts`.

Regarding CI/CD infrastructure, participants (P10 and P27) highlights that the integration environment acts as a gatekeeper, where new features must pass automated validation processes such as builds, tests, and static analysis. In practice, developers must ensure that modern constructs are compatible with pipeline configurations, including tooling, language versions, and validation rules. When incompatibilities arise, such as CI rejecting features like type hints, developers are required to either adapt the infrastructure, adjust configurations, or postpone adoption. This establishes a practice in which rejuvenation is not solely a code-level decision, but one that must be synchronized with the continuous validation environment.

For example, P10 illustrates a strategy in which CI/CD pipelines are actively used as a validation mechanism to support refactoring and source code rejuvenation efforts. In this approach, developers rely on automated tests integrated into continuous integration workflows to ensure that changes preserve system behavior.

> *"The challenges. Well, it's usually the legacy code. That's the main issue. Because we have to do a refactoring, and it has to go through some tests. Sometimes you have to go through the whole QA again; if you don't have the unit tests, you have to be careful to do a refactoring of the code. So, as we have CI/CD nowadays, this has become easier, because you already write the tests in a way that it will run as soon as you modify, and the merge will only be done if this test passes on the pipeline side, on the side of the tool you are using to maintain your code. So, I think that was a nice touch, applying these pipelines on CI/CD. And the challenge is refactoring and maintaining this software, and not degrading it. Do a refactoring and use this to make your life easier, and don't get any bugs in the process."*
>
> [**P10**]

As a contingency, CI/CD Infrastructure and Continuous Validation captures how developers incorporate infrastructure constraints into their rejuvenation practices. CI pipelines act as a mediating mechanism that regulates change by requiring alignment between new constructs and existing validation processes. These practices enable controlled and validated integration of changes, but also constrain source code rejuvenation when infrastructure is not prepared to accommodate newer features. As a result, rejuvenation becomes dependent on the readiness and configuration of continuous integration environments.

The identified contingencies indicate that source code rejuvenation is implemented via intentional, organized, and human-centric strategies that dictate the execution of changes in practice. Developers always use careful planning, step-by-step execution, and constant evaluation of effort and expected outcomes when working on source code rejuvenation. This makes sure that changes are made in a controlled and manageable way.

Instead of being used all at once, rejuvenation is done at specific times and often as part of ongoing development work. This makes it possible to adopt new constructs in a way that is opportunistic and doesn't cause too much disruption. These practices are further strengthened by the selective and human-controlled use of tools, where automation is used to help with certain tasks while developers are still in charge of checking and improving the work.

Collaboration, prior knowledge, and team dynamics are also very important for coordinating and supporting rejuvenation efforts. Continuous technical debt management, on the other hand, makes rejuvenation a part of ongoing development practices. Lastly, developers change their plans to deal with technical problems and use CI/CD pipelines to check and control changes.

When you look at all of these contingencies together, they show that source code rejuvenation is a coordinated set of strategies and practices that structure, guide, and regulate its execution. This means that control, gradual evolution, and developer-driven decision-making are all important.

4.6 Covariances

Covariances describe the relationships between categories, where variations in one category are consistently associated with variations in another [2, 6]. In the context of source code rejuvenation, covariances capture the interdependent relationships through which causes, conditions, contingencies, and consequences jointly shape and influence rejuvenation practices.

Following Glaser's theoretical coding approach and the use of the six C's template in STGT studies [6, 73, 70], we identified the covariances after developing, refining, and theoretically integrating the main categories of the theory. The analysis examined the coded data, theoretical memos, and cross-category comparisons to identify recurring relationships in which variation in one category was associated with variation in another. In particular, this thesis compared how different causes were associated with distinct consequences, how restrictive and enabling conditions were associated with variations in consequences, and how different contingencies were adopted in response to the conditions surrounding source code rejuvenation. We included these relationships as covariances only when the empirical material repeatedly grounded them and when they helped explain the socio-technical dynamics of source code rejuvenation without requiring a causal or temporal interpretation. The following subsections further elaborate each covariance, grounding these relationships in the data and in the concepts derived from the analysis.

4.6.1 Causes and consequences

The comparison between Causes and Consequences suggests that the outcomes of source code rejuvenation vary according to the type of trigger involved. When rejuvenation is motivated by system obsolescence, developers mainly perceive consequences related to maintainability, code comprehension, reliability, and the avoidance of degradation. In these cases, outdated code, unsupported versions, deprecated constructs, and accumulated migration costs make rejuvenation a way to keep the system viable over time. Developers reported that old codebases become harder to understand and maintain, especially for newcomers, and that replacing deprecated or obsolete constructs with modern alternatives can improve readability, expressiveness, and long-term maintenance.

When rejuvenation is triggered by language evolution enforcement, the most evident consequences are related to reliability, correctness, security, and maintainability. Developers emphasized that when a language version reaches end-of-life, loses technical support, or receives bug fixes and security improvements in newer versions, rejuvenation becomes a necessary maintenance action rather than an optional improvement. In these situations, updating the code helps avoid unsupported versions, unpatched vulnerabilities, compatibility problems, and the increasing cost of keeping the system several versions behind. Thus, language evolution is associated not only with adopting new features but also with preserving correctness, operational safety, and future maintainability.

When the trigger is dependency and architectural simplification, the consequences are more strongly related to maintainability and long-term system stability. Developers described situations in which third-party libraries became difficult to maintain, outdated, incompatible, or unnecessary after equivalent functionality became available in the language or standard libraries. In these cases, rejuvenation is perceived as a way to reduce external dependencies, simplify the architecture, decrease compatibility problems, and lower the effort required to maintain the system. The expected benefit is less about adopting modern syntax for its own sake and more about reducing maintenance burden and dependency-related complexity.

When rejuvenation is driven by dependency and framework pressure, the consequences are mainly associated with compatibility and maintainability. Developers reported that frameworks and libraries evolve at their own pace and may require projects to adopt newer language versions, APIs, or coding patterns. In this case, rejuvenation occurs as part of keeping the system aligned with the surrounding technical ecosystem. The main perceived consequence is the ability to continue evolving the system together with its dependencies, avoiding misalignment with frameworks, libraries, and platform requirements.

Finally, when rejuvenation is influenced by community-driven adoption, the consequences are more closely related to awareness, learning, readability, and expressiveness.

Developers mentioned that talks, blog posts, discussions, and community practices expose them to modern language features and help them understand when these features may be useful. As developers learn about these possibilities, they may adopt constructs that make code shorter, clearer, easier to understand, or more aligned with current practices. In this sense, community influence contributes to the diffusion of rejuvenation practices, but the perceived consequences depend on which features are adopted and how they are applied in each project.

Overall, these covariances indicate that the consequences of source code rejuvenation are not uniform across all situations. Different triggers tend to be associated with different perceived or anticipated outcomes. Therefore, the relationship between causes and consequences should be interpreted as variation across categories, not as a fixed causal sequence.

4.6.2 Conditions and consequences

The comparison between Conditions and Consequences suggests that restrictive conditions tend to vary with degradation effects when they delay, limit, or prevent source code rejuvenation. Organizational and managerial constraints reduce developers' autonomy and often make rejuvenation secondary to business demands, delivery pressure, and short-term functionality. When companies do not perceive rejuvenation as relevant or when managers require approval without prioritizing internal quality, teams tend to postpone these efforts. Over time, this postponement contributes to technical debt, higher migration cost, and greater difficulty in maintaining outdated systems.

Effort, cost, compatibility, and legacy constraints also reinforce this degradation pattern. When rejuvenation requires high effort, extensive validation, interruption of environments, or coordination across multiple users and systems, developers tend to avoid broad changes or perform only small updates. Large gaps between language versions increase this difficulty because developers lose the opportunity to evolve the code gradually. Similarly, compatibility requirements with old platforms, runtimes, dependencies, or clients restrict the adoption of modern features. As a result, systems remain tied to obsolete constructs and become harder to evolve, which increases future maintenance burden and makes later rejuvenation more complex and risky.

External dependencies, environment constraints, standards, governance rules, and domain-specific restrictions further shape this relationship. Outdated libraries, inactive communities, vendor-imposed toolchains, deployment environments, and framework dependencies can block or delay rejuvenation even when developers want to modernize the code. Coding standards and project conventions may also discourage the introduction of modern constructs when they conflict with existing patterns. In critical or low-level

domains, developers adopt an even more conservative stance because failures can affect the whole product chain. Under these conditions, teams tend to delay rejuvenation until the technical environment, organizational rules, or domain constraints allow safer change. This delay may intensify degradation effects by increasing technical debt, complexity, and future adaptation cost.

Risk, uncertainty, fear, poor internal quality, and limited knowledge also constrain rejuvenation and may contribute to degradation when teams avoid change for long periods. Developers may fear introducing bugs, regressions, silent behavior changes, performance problems, or debugging difficulties. When they trust the legacy system because it already works, they may prefer to keep it unchanged instead of introducing uncertainty. Low-quality code, non-modular architectures, and accumulated technical debt make the situation worse because developers cannot isolate changes easily. In addition, when knowledge about modern features is not shared across the team, developers avoid adopting them to prevent future maintenance problems. These conditions slow down decisions, restrict the scope of rejuvenation, and can leave systems progressively harder to maintain.

In contrast, enabling conditions tend to vary with positive consequences such as maintainability, productivity, reliability, and correctness. Automated and regression testing gives developers confidence to change code because tests help detect regressions and verify that behavior remains consistent. When teams have strong test coverage, they can conduct rejuvenation more safely, fix problems earlier, and reduce uncertainty during migration or refactoring. This facilitates improvements in maintainability and reliability because developers can modernize code while preserving expected behavior.

Language and architectural enablers also facilitate rejuvenation and support positive outcomes. Modular architectures, service-oriented structures, and separation of concerns allow developers to update only specific parts of the system instead of changing the whole application. This reduces risk, supports incremental evolution, and makes rejuvenation more manageable. Flexible languages and architectures make it easier to adopt modern features, improve code organization, reduce complexity, and support long-term maintainability. When developers can apply changes in isolated and controlled parts of the system, rejuvenation becomes more feasible.

Overall, restrictive conditions tend to covary with degradation effects because they delay or prevent rejuvenation, allowing systems to accumulate technical debt, compatibility problems, and higher future maintenance costs. Enabling conditions tend to covary with positive consequences because they make rejuvenation safer, more controlled, and easier to justify. Testing, modular architecture, flexible languages, shared knowledge, and trustworthy tool support help developers conduct rejuvenation with lower risk and greater confidence. In these situations, rejuvenation can contribute to improved maintainability,

productivity, reliability, correctness, and long-term software sustainability.

4.6.3 Conditions and Contingencies

The comparison between Conditions and Contingencies suggests that developers adopt different strategies depending on the restrictive or enabling conditions surrounding source code rejuvenation. Organizational and managerial constraints appear associated with decision-making and strategic evaluation practices, since developers need to negotiate rejuvenation with business priorities, management approval, delivery pressure, and perceived organizational value. Under these conditions, teams tend to evaluate costs and benefits, seek collective agreement, and embed rejuvenation into planned or ongoing maintenance activities instead of performing it as an isolated technical improvement.

Effort, cost, and practical constraints appear associated with execution strategy and effort management. When rejuvenation requires substantial time, coordination, revalidation, or interruption of development and production environments, developers tend to plan changes carefully, apply them incrementally, and limit the scope of transformations. These conditions also appear associated with adoption strategy and timing, since teams may delay rejuvenation until the effort becomes justifiable or until the code is already being modified for another reason.

Compatibility and legacy constraints appear associated with strategies for managing technical constraints and compatibility. When systems need to support older platforms, runtimes, clients, or language versions, developers tend to adapt the scope of rejuvenation, postpone modern constructs, remove legacy constraints before applying changes, or use temporary workarounds to keep the system compiling and operating. These conditions also vary with delayed or reactive adoption, since rejuvenation often depends on dropping legacy requirements or aligning the system with compatible versions of dependencies and environments.

External dependencies and environment constraints appear associated with compatibility management, strategic evaluation, and timing decisions. When frameworks, libraries, toolchains, vendors, or runtime environments limit the use of modern features, developers tend to coordinate upgrades across the dependency chain, evaluate whether the surrounding ecosystem is ready, and adopt rejuvenation incrementally. When communities are active and documentation is available, these same conditions may facilitate adoption by reducing uncertainty and supporting better-informed decisions.

The availability of automated and regression testing appears associated with CI/CD infrastructure and continuous validation. When test suites, regression tests, and validation pipelines exist, developers can introduce rejuvenation with greater confidence because they can check whether the expected behavior remains stable. This condition also varies

with execution strategy and tool-supported rejuvenation, since testing makes incremental changes, automated suggestions, and partial transformations safer to review and validate.

Risk, uncertainty, and fear factors appear associated with cautious decision-making, staged execution, and delayed adoption. When developers fear bugs, regressions, instability, or production failures, they tend to evaluate risks more explicitly, prefer smaller changes, and avoid broad transformations. Trust in legacy systems also reinforces conservative strategies, since teams may decide to preserve working code unless the expected benefits of rejuvenation clearly justify the risk.

Unpredictability and interaction complexity in language evolution appear associated with delayed adoption, incremental execution, and human-controlled tool use. When new language features interact in unexpected ways, introduce silent behavior changes, affect performance, or complicate debugging, developers tend to wait until the features become mature, better understood, and supported by tools. They also tend to avoid fully automated transformations in semantically complex situations, relying instead on manual review, contextual reasoning, and careful validation.

Tooling constraints and control mechanisms appear directly associated with selective, risk-aware, and human-controlled use of tool support. When developers do not trust automated tools, they use automation only for simple and easy-to-check changes. Tools become more acceptable when they provide suggestions, allow configuration, and keep developers responsible for final decisions. Thus, tooling conditions vary with a human-in-the-loop strategy in which tools support detection and partial automation but do not replace developer judgment.

Human and knowledge factors appear associated with knowledge, experience, and team dynamics. When knowledge about modern language features is unevenly distributed, developers tend to avoid adopting constructs that other team members may not understand or maintain. In this context, code review, team discussion, shared learning, and prior experience become important strategies for coordinating rejuvenation. When the team already has familiarity with modern constructs, the effort required to adopt them tends to be lower, making rejuvenation more feasible.

Technical debt and system quality constraints appear associated with continuous technical debt management and careful execution strategies. When systems present low internal quality, non-modular architecture, or accumulated complexity, developers tend to treat rejuvenation as part of a broader effort to manage technical debt. They may apply changes gradually, isolate safer parts of the system, and prioritize improvements that reduce long-term maintenance burden. In low-quality codebases, rejuvenation often requires additional planning because rejuvenating constructs without addressing structural problems may increase comprehension difficulties.

Standards, governance, and process constraints appear associated with strategic evaluation and team-based coordination. When coding standards, project conventions, or governance rules limit the adoption of modern constructs, developers tend to discuss changes collectively, follow existing patterns, and postpone rejuvenation until there is agreement to update project-wide practices. Under these conditions, rejuvenation becomes constrained by consistency, readability for the broader team, and alignment with accepted development processes.

Language and architectural enablers appear associated with incremental execution and more manageable adoption strategies. When systems use modular architectures, service-oriented structures, or flexible languages, developers can apply rejuvenation to isolated components instead of changing the whole system at once. These enabling conditions vary with strategies that favor localized changes, gradual source code rejuvenation, and controlled evolution. They also support continuous technical debt management because teams can improve parts of the system without exposing the entire application to unnecessary risk.

Perceived relevance and motivation appear associated with decision-making and team dynamics. When developers or organizations do not perceive language evolution as relevant, rejuvenation tends to lose priority. In these situations, teams need to justify rejuvenation through expected benefits such as maintainability, reliability, productivity, or reduction of future costs. When motivation is stronger and the benefits are clearer, developers can more easily negotiate and initiate rejuvenation efforts.

Contextual and domain factors appear associated with cautious strategic evaluation, delayed adoption, and continuous validation. In critical, low-level, or reliability-sensitive domains, developers tend to adopt stricter criteria before applying modern features. They may wait for maturity, require stronger evidence of reliability, and depend more heavily on testing and CI/CD validation. Under these conditions, rejuvenation tends to occur only when the expected benefits align with the domain's tolerance for risk and operational impact.

Overall, these covariances suggest that contingencies vary according to the conditions that restrict or enable source code rejuvenation. Restrictive conditions, such as organizational pressure, high effort, compatibility requirements, fear of regressions, tooling limitations, low code quality, and domain criticality, appear associated with cautious strategies such as cost-benefit analysis, delayed adoption, incremental execution, manual validation, and compatibility workarounds. Enabling conditions, such as automated testing, modular architecture, flexible languages, active communities, shared knowledge, and configurable tools, appear associated with more feasible and controlled rejuvenation strategies, including continuous validation, localized changes, team-based coordination,

and selective tool support. Therefore, developers do not apply a single uniform strategy for rejuvenation; instead, they adjust their contingencies according to the conditions that shape the technical and organizational context of each effort.

Chapter 5

Discussions

This chapter discusses the emergent socio-technical theory of source code rejuvenation, situating it within the existing body of knowledge and reflecting on its contributions, implications, and limitations. First, Section 5.1 revisits the research questions and explains how they are addressed through the developed theory. Next, Section 5.2 presents theoretical hypotheses derived from the relationships among the Six C's categories, synthesizing how the phenomenon tends to emerge, vary, and produce consequences in practice. Then, Section 5.3 outlines the implications of the findings, highlighting how they support and guide developers and organizations. Section 5.4 positions the emergent theory in relation to prior studies, highlighting points of convergence, divergence, and theoretical contribution. Finally, Section 5.5 discusses the limitations of this study, reflecting on its scope and potential constraints.

5.1 Answering the Research Questions Through the Emergent Theory

Although the research questions guided the early stages of data collection and exploration, the final explanations emerge from the grounded theory developed in this thesis. Responses below are derived from the resulted theory, specifically the theoretical model articulated through the six C's framework presented in Chapter 4. This approach ensures that the answers reflect the integrated explanation produced by the theory rather than a descriptive summary of the data. The following subsections present the answers to each research question introduced in Section 1.2.

5.1.1 RQ1: What are the motivations that lead software developers to rejuvenate their software?

Within the emerging theory, the motivations for source code rejuvenation correspond primarily to the causes (Section 4.3) dimension. These causes represent the pressures that trigger rejuvenation activities as software systems evolve within changing technological environments. In addition to these external pressures, developers are also motivated by the perceived benefits associated with the outcomes of rejuvenation, which correspond to the consequences (Section 4.4) dimension of the theory.

Across the data, rejuvenation is frequently motivated by the need to prevent software obsolescence and maintain long-term system viability. As programming languages evolve and introduce new constructs, outdated idioms gradually accumulate in codebases, increasing maintenance difficulty and reducing compatibility with modern development environments. Developers therefore perform rejuvenation to keep systems aligned with contemporary language capabilities and development practices.

A second motivation arises from the pace of programming language evolution itself. As new constructs become available, developers often recognize opportunities to simplify implementations, improve code expressiveness, or reduce technical complexity. These changes are rarely performed purely for stylistic reasons; instead, they are often triggered when developers revisit existing code during maintenance tasks.

Support discontinuities also play a critical role. The end-of-life of language versions, frameworks, or external dependencies creates practical pressure for rejuvenation, forcing teams to update legacy constructs in order to maintain security updates, compatibility, and integration with modern software environments.

Warnings and alerts from compilers, linters, and development tools frequently act as additional catalysts for rejuvenation. Deprecation notices, and automated suggestions highlight outdated constructs and encourage developers to source code rejuvenation toward newer language features.

Beyond these technological pressures, developers are also motivated by the expected improvements that rejuvenation can bring to the codebase. Perceived benefits such as clearer and more expressive code, code understanding improvements, improved maintainability, reduced reliance on external library dependencies, and gains in development efficiency frequently encourage developers to adopt modern language constructs. These expected outcomes reflect the positive consequences identified in the theory and often influence developers' decisions when evaluating whether rejuvenation efforts are worthwhile.

Taken together, these motivations indicate that rejuvenation emerges from a combination of external pressures and perceived benefits. While technological changes and development tool signals push developers toward rejuvenation, expectations of improved code

quality, maintainability, and development efficiency also pull developers toward adopting modern language constructs.

5.1.2 RQ2: What are the challenges that hinder software developers from rejuvenating their legacy code?

The challenges associated with rejuvenation are explained in the theory through a combination of context (Section 4.1) and conditions (Section 4.2). Together, these dimensions reveal the socio-technical constraints that complicate rejuvenation efforts.

From a contextual perspective, rejuvenation takes place within legacy and outdated systems, complex software environments characterized by compatibility requirements, dependency networks, and evolving programming language ecosystems. Many systems must remain compatible with older runtimes, external services, or long-lived libraries, which restricts the adoption of newer language constructs. Similarly, frameworks, plugins, and third-party libraries may evolve at different speeds, creating dependency chains that complicate source code rejuvenation efforts.

Operational conditions further hinder rejuvenation activities. Large and long-lived codebases often contain millions of lines of code, complex architectures, accumulated technical debt, and limited documentation, making modifications difficult and risky. source code rejuvenation efforts frequently require extensive regression testing and careful validation to avoid unintended behavioral changes. In addition, effort and cost constraints, along with limited time and competing organizational priorities, often restrict developers' ability to allocate resources to rejuvenation activities.

Organizational and governance-related constraints also play a significant role. Developers may face restrictions imposed by management decisions, contractual obligations, service-level agreements, or regulatory requirements, which can delay or prevent rejuvenation efforts even when technical benefits are evident. These constraints are often reinforced by the need to maintain system stability and ensure continuous operation in production environments.

Furthermore, risk, uncertainty, and fear factors influence developers' decisions. Concerns about introducing bugs, unintended side effects, or destabilizing reliable legacy systems contribute to a cautious approach, often discouraging proactive rejuvenation. The complexity and unpredictability of programming language evolution may further increase this uncertainty.

Finally, rejuvenation introduces cognitive and coordination challenges. New language constructs may be unfamiliar to some team members, increasing the cognitive load required to understand and maintain rejuvenated code. Differences in expertise, lack of

shared knowledge, and misalignment within teams may slow adoption or create disagreement about when and how rejuvenation should occur.

5.1.3 RQ3: What practices do developers follow or propose to deal with the challenges of source code rejuvenation?

The practices employed by developers to address the challenges of source code rejuvenation are presented in contigencies (Section 4.5). The findings indicate that these practices are not ad hoc, but rather structured as a coordinated set of strategies through which developers operationalize rejuvenation efforts in practice. These strategies reflect how developers manage risk, complexity, and system constraints while integrating modern language features into existing codebases.

source code rejuvenation is operationalized through deliberate execution strategies and effort management practices. Developers emphasize careful planning, anticipating the impact of changes, and organizing their application in a controlled manner. Rejuvenation is typically performed incrementally, allowing developers to introduce changes step-by-step, reducing disruption and facilitating validation. In addition, manual transformations remain central, reflecting the need for precision and control when modifying legacy systems.

Decision-making and strategic evaluation also play a central role in guiding source code rejuvenation. Developers continuously assess trade-offs between effort, risk, and expected benefits, ensuring that rejuvenation is aligned with system constraints and project priorities. These decisions are not purely technical but involve contextual judgment regarding when and how changes should be applied.

Furthermore, developers adopt specific strategies regarding timing, applying source code rejuvenation opportunistically as part of ongoing development activities. Instead of performing large-scale transformations, rejuvenation is often integrated into regular maintenance tasks, enabling gradual adoption of modern language features while minimizing disruption to stable systems.

Tool support is employed in a selective, risk-aware, and human-controlled manner. Rather than relying on full automation, developers use tools to suggest changes and identify opportunities for rejuvenation, while maintaining responsibility for reviewing and validating these suggestions. This reflects a preference for decision support over automation, preserving human control over the transformation process.

Collaboration, knowledge sharing, and team dynamics are also essential practices supporting source code rejuvenation. Developers rely on shared understanding and prior experience to coordinate changes and ensure that modern constructs can be effectively

adopted and maintained across the team. This highlights the social dimension of rejuvenation efforts.

In addition, developers incorporate continuous technical debt management into their practices, treating source code rejuvenation as an ongoing activity rather than a one-time effort. By progressively addressing outdated constructs, they reduce long-term maintenance costs and avoid the accumulation of technical debt.

Developers also actively manage technical constraints and compatibility requirements, adapting their strategies to deal with legacy dependencies, environment limitations, and system restrictions. These practices ensure that rejuvenation efforts remain feasible within existing technical boundaries.

Finally, CI/CD infrastructure and continuous validation mechanisms are used to support source code rejuvenation by ensuring that changes preserve system behavior. Automated tests and integration pipelines act as safeguards, enabling developers to apply rejuvenation with greater confidence while controlling the risk of introducing defects.

Overall, these findings show that source code rejuvenation is carried out through structured, incremental, and human-centered practices, emphasizing control, gradual evolution, and continuous evaluation as key mechanisms to deal with its inherent challenges.

5.2 Hypotheses Derived from the Emergent Theory

After the development, refinement, and integration of the main categories of the emergent theory, this section presents a set of theoretical hypotheses derived from the relationships observed among the Six C's categories. These hypotheses do not aim to establish deterministic or statistically tested causal claims. Instead, they synthesize recurrent patterns identified across the empirical material and express plausible theoretical relationships among contexts, conditions, causes, contingencies, and consequences.

In this thesis, the hypotheses are formulated as theoretical propositions grounded in the empirical evidence and analytical comparisons conducted throughout the study. They help explain how source code rejuvenation tends to emerge, under which circumstances it becomes more likely, what factors shape its execution, and what consequences may result from different rejuvenation practices. Thus, the hypotheses serve as an analytical bridge between the categorical structure of the theory and its broader explanatory contribution.

5.2.1 H1: Rejuvenation as a Response to Obsolescence Pressure

Hypothesis 1: Derived from the relationship between the Context and Causes of the emergent theory, this hypothesis states that source code rejuvenation tends to be initiated when developers begin to face evident pressures arising from system obsolescence,

lack of support, security risks, abandoned or obsolete dependencies, and the evolution of frameworks and libraries. Under these contextual and causal pressures, rejuvenation is no longer perceived merely as an optional improvement, but as a necessary maintenance response to preserve compatibility, security, operability, and long-term maintainability.

The hypothesis suggests that source code rejuvenation frequently emerges as an necessary response to accumulated technical and ecosystem pressures. In such cases, developers and organizations do not rejuvenate the code only because modern language features are available, but because continuing to maintain obsolete constructs, unsupported versions, or outdated dependencies becomes increasingly costly, risky, or impractical.

5.2.2 H2: Rejuvenation as a Driver of Quality and Productivity Improvements

Hypothesis 2: This hypothesis derives from the positive Consequences identified in the emergent theory. It suggests that source code rejuvenation tends to be perceived as valuable when developers and organizations associate the adoption of modern language features with improvements in code readability, expressiveness, reliability, correctness, performance, maintainability, testability, and development productivity. Under these expected or observed benefits, rejuvenation is not only understood as a response to obsolescence, but also as a strategy for improving the internal quality of software systems and supporting more efficient development practices.

This hypothesis indicates that the perceived value of source code rejuvenation is strongly connected to its expected contribution to software quality and development efficiency. In this sense, developers do not adopt modern language features only because a system is obsolete or externally pressured to evolve, but also because these features are perceived as mechanisms for making the codebase clearer, safer, easier to test, and less costly to maintain. Therefore, positive consequences may act both as outcomes of rejuvenation and as anticipatory reasons that justify its adoption.

5.2.3 H3: Rejuvenation Constraints as Socio-Technical Barriers

Hypothesis 3: This hypothesis derives from the restrictive Conditions identified in the emergent theory. It suggests that the constraints that limit or delay source code rejuvenation are not merely technical, but socio-technical in nature. Organizational priorities, managerial approval, governance rules, ecosystem dependencies, domain-specific risks, team knowledge, resistance to change, and lack of shared understanding can restrict rejuvenation as much as, or even more than, purely technical limitations. Therefore, the feasibility of source code rejuvenation depends not only on the availability of modern lan-

guage features or technical migration paths, but also on organizational, ecosystem, and human conditions that shape whether developers are able, allowed, or willing to perform such changes.

This hypothesis highlights that source code rejuvenation is embedded in a broader socio-technical environment. Even when developers recognize technical benefits, rejuvenation may be delayed or restricted by business priorities, regulatory constraints, process rules, lack of testing infrastructure, insufficient tool trust, or uneven knowledge across the team. Thus, restrictive conditions also shape the contingencies adopted by developers, leading to selective, incremental, delayed, or conservative rejuvenation strategies.

5.2.4 H4: Delayed Rejuvenation as a Contributor to System Degradation

Hypothesis 4: This hypothesis derives from the relationship between restrictive Conditions and negative Consequences identified in the emergent theory. It suggests that when organizational, technical, ecosystem, or human constraints delay or prevent source code rejuvenation, software systems tend to experience degradation effects over time. In such cases, the system may accumulate technical debt, become increasingly misaligned with language and ecosystem evolution, and require more complex, costly, and risky changes in the future.

This hypothesis indicates that postponing rejuvenation may transform a gradual maintenance activity into a more disruptive migration effort. As outdated constructs, unsupported versions, obsolete dependencies, and accumulated compatibility gaps remain unresolved, the cost and risk of future rejuvenation tend to increase. Therefore, restrictive conditions do not only affect the timing or feasibility of rejuvenation; they may also contribute to long-term degradation by making future maintenance harder, less predictable, and more expensive.

5.2.5 H5: Rejuvenation as an Adaptive Strategy to System Conditions

Hypothesis 5: Derived from the relationship between Conditions and Contingencies in the emergent theory, this hypothesis states that the strategies adopted to perform source code rejuvenation vary according to the technical, organizational, human, and infrastructural conditions surrounding the system. Under restrictive or uncertain conditions, developers tend to adopt more incremental, selective, risk-aware, and human-controlled strategies. Conversely, when favorable conditions are present, such as shared team knowl-

edge, modular architectures, reliable tool support, automated tests, and CI/CD infrastructure, rejuvenation efforts become more feasible, controlled, and continuous.

This hypothesis suggests that source code rejuvenation is not performed through a single uniform strategy. Instead, developers adapt the timing, scope, automation level, validation mechanisms, and execution approach according to the context in which the system evolves. For example, complex systems, compatibility constraints, limited time, lack of trust in tools, or insufficient testing infrastructure may lead developers to perform localized and manual changes. In contrast, mature validation pipelines, team alignment, prior experience, and continuous technical debt management may support safer and more systematic rejuvenation efforts.

5.2.6 H6: Rejuvenation Tool Support as Controlled and Human-Validated Automation

Hypothesis 6: Derived from the relationship between Conditions and tooling-related Contingencies in the emergent theory, this hypothesis suggest that automated tool support can facilitate source code rejuvenation, but its practical use tends to remain selective, risk-aware, and dependent on human validation. When programming languages evolve, tools such as upgrade kits, refactoring engines, static analyzers, IDE inspections, CI/CD pipelines, and LLM-based assistants may help developers identify source code rejuvenation opportunities, apply simple transformations, detect fragments requiring human assistance, and validate changes through regression testing. However, under conditions involving compatibility constraints, domain risks, low trust in automation, limited team knowledge, confidentiality concerns, or semantic complexity, developers tend to treat automation as decision support rather than as a fully autonomous mechanism for rejuvenation.

This hypothesis suggests that the expectation of fully automated upgrade support, envisioned in earlier work [8] on language evolution and source code transformations, remains only partially realized in practice. The emergent theory explains this limitation by showing that source code rejuvenation is not only a syntactic transformation problem, but also a socio-technical activity shaped by risk, trust, compatibility, domain constraints, team knowledge, and validation practices. Even with more advanced support, including LLM-based tools, developers still perceive automation as something that must be controlled, reviewed, adapted, and validated before being incorporated into actively maintained systems.

Overall, these hypotheses synthesize the explanatory contribution of the emergent theory by showing that source code rejuvenation is not a uniform, purely technical, or automatically triggered activity. Instead, the phenomenon behaves as a socio-technical

process shaped by the interaction between contexts, causes, conditions, contingencies, consequences, and covariances. The hypotheses indicate that rejuvenation tends to emerge under pressures of obsolescence, lack of support, security risks, dependency evolution, and ecosystem change; that it is justified by expected improvements in readability, reliability, performance, maintainability, testability, and productivity; and that its feasibility is constrained by organizational priorities, compatibility demands, human knowledge, risk perception, tool trust, and validation infrastructure. They also show that, when postponed or restricted, rejuvenation may contribute to system degradation, while its actual execution depends on adaptive strategies such as incremental changes, selective automation, human review, continuous validation, and technical debt management. Thus, the theory explains source code rejuvenation as a situated maintenance response through which developers and organizations negotiate source code rejuvenation, risk, quality, and long-term system viability in evolving software ecosystems.

5.3 Implications

The theory presented in this thesis provides a socio-technical explanation of how source code rejuvenation unfolds in practice, highlighting the interplay between context, causes, conditions, contingencies, and consequences. These findings offer actionable implications for software developers and teams who are responsible for evolving long-lived systems under continuous programming language evolution. For the purpose of practical guidance, twelve recommendations are presented below; however, they should be interpreted as supportive guidelines rather than prescriptive or universally applicable rules.

- **Treat rejuvenation as a continuous strategic activity.** The results in Causes indicate that source code rejuvenation should not be treated as an occasional or purely reactive effort. Causes such as system obsolescence, language evolution enforcement, and dependency pressure suggest that delaying rejuvenation may contribute to technical debt accumulation and increasing maintenance complexity. Therefore, developers should proactively monitor signals of obsolescence, such as outdated dependencies, deprecated constructs, and growing maintenance difficulty, and incorporate rejuvenation into regular development workflows.

- **Adopt execution strategies according to risk and context.** The identified Contingencies emphasize the importance of structured execution strategies that fit the system context and its associated risks. In critical or constrained contexts, developers often prefer incremental and controlled changes, applying small and repeated transformations to reduce risk, manage complexity, and preserve system behavior.

However, depending on the situation, both incremental and large-scale rejuvenation efforts may be appropriate. Regardless of the approach, careful planning, risk assessment, validation mechanisms, and strategic decision-making are important to conduct rejuvenation safely and effectively.

- **Evaluate trade-offs before adopting modern constructs.** The findings emphasize the role of strategic decision-making in guiding source code rejuvenation efforts. Developers should continuously evaluate trade-offs among effort, risk, compatibility constraints, and expected benefits. This means that rejuvenation decisions should not rely only on the availability of modern language features, but also on their relevance to the system context, their expected impact on maintainability and performance, and the feasibility of adoption under existing constraints.

- **Integrate rejuvenation into ongoing development work.** The results show that source code rejuvenation benefits from being integrated opportunistically into regular development activities. Instead of treating rejuvenation as a separate phase, developers should align it with maintenance tasks, bug fixes, refactoring activities, and feature development opportunities. This approach supports gradual adoption of modern constructs, reduces disruption to stable systems, and avoids the higher cost of large isolated rejuvenation efforts.

- **Keep human control over tool-supported transformations.** The theory highlights the importance of maintaining human control over tooling support. Tools can help identify rejuvenation opportunities and suggest transformations, but developers should critically evaluate and validate these suggestions before applying them. This selective and risk-aware use of tools helps ensure that transformations remain aligned with system-specific constraints and reduces the likelihood of introducing unintended defects.

- **Promote collaboration and knowledge sharing.** Collaboration and knowledge sharing emerge as important enablers of source code rejuvenation. Teams should disseminate knowledge about modern language features and develop a shared understanding of the constructs adopted in the codebase. This reduces resistance to change, facilitates coordinated transformations, and helps ensure that rejuvenated code remains maintainable by all team members.

- **Align rejuvenation with technical debt management.** The findings suggest that source code rejuvenation should be closely connected to continuous technical debt management practices. By progressively addressing outdated constructs and

reducing architectural complexity, developers can avoid the accumulation of degradation effects and support the long-term sustainability of the system.

- **Manage technical and compatibility constraints explicitly.** Developers should actively manage technical constraints and compatibility requirements when performing source code rejuvenation. This includes considering dependency chains, runtime environments, supported language versions, and backward compatibility policies. Rejuvenation should be planned according to these constraints to ensure that changes are feasible and do not compromise system stability.

- **Use CI/CD and automated validation as safeguards.** The use of CI/CD infrastructure and automated validation mechanisms is critical to support source code rejuvenation. Continuous integration pipelines and automated tests help developers verify that changes preserve system behavior and reduce the risk of regressions. Investing in testing and validation infrastructure increases confidence in applying rejuvenation efforts and supports safer system evolution.

- **Recognize mixed codebases as natural contexts for rejuvenation.** The identified contexts in Section 4.1 indicate that source code rejuvenation predominantly occurs in environments where legacy and modern language constructs coexist within the same system, leading to heterogeneous codebases that evolve unevenly over time. Developers should therefore recognize that rejuvenation is more likely to emerge in mixed and transitional environments, where the accumulation of different language paradigms increases cognitive load and maintenance complexity. Understanding these contexts helps teams see source code rejuvenation as part of the natural evolution of software systems rather than as an isolated transformation effort.

- **Assess feasibility before initiating rejuvenation.** The Conditions highlight that source code rejuvenation is not always feasible and is often constrained or enabled by technical, organizational, and environmental factors. Developers should assess conditions such as effort and cost constraints, compatibility requirements, testing infrastructure, organizational priorities, and team knowledge before initiating rejuvenation efforts.

- **Align rejuvenation with contractual, product, and governance constraints.** The findings indicate that contractual obligations, product requirements, and governance constraints may significantly restrict when and how developers can perform rejuvenation. These constraints include limitations imposed by client agreements, service-level expectations, regulatory requirements, release schedules, and the need

to maintain system stability in production environments. Such constraints may delay, limit, or even prevent rejuvenation efforts, even when technical benefits are clear. Understanding these conditions helps teams identify when rejuvenation is viable, mitigate risks, and design strategies aligned with existing constraints, obligations, and system capabilities.

Overall, these implications position source code rejuvenation as a disciplined, incremental, and socio-technical practice, where successful adoption depends not only on technical decisions but also on planning, collaboration, and continuous evaluation within evolving development environments.

5.4 Positioning the Emergent Theory in the Literature

This study is situated within a growing body of research that examines the evolution of programming languages and the adoption of modern language features across legacy systems developed in programming languages such as C++, JavaScript, and Python.

Prior work has investigated different aspects that envolve this phenomenon, including the extent to which modern language features are adopted; the evolving patterns of their usage over time; the impacts perceived by developers when incorporating these features into existing systems; practices and automated tools to support these efforts; and motivations, challenges, and trade-offs developers encounter throughout migration and modernization efforts.

Rather than treating these studies as prescriptive foundations, this thesis use them to contextualize and position the emergent socio-technical theory developed with existing literature. In doing so, the related work was organized into three complementary areas:

(i) **Tools and Techniques for Source Code Rejuvenation** — Studies that propose techniques, frameworks, or automated or semi-automated approaches to support the integration of modern language features into existing codebases.

(ii) **Feature Adoption and Usage Analysis** — Research that examines how developers adopt and use language features, patterns of their adoption, as well as the impacts on systems.

(iii) **Software Modernization and Migration Studies** — Works that investigate broader processes of system evolution, including language migrations, large-scale code transformations, and modernization efforts, often focusing on the technical and organizational dynamics involved in transitioning legacy systems to newer paradigms.

The following subsections discuss these three areas, using them to contextualize and position the emergent socio-technical theory developed in this thesis.

5.4.1 Tools and Techniques for Source Code Rejuvenation

As programming languages change, they add new features to make codes easier to understand, shorter, and faster to write. Due to these changes, developers are actively seeking practices and techniques to update old codebases and integrate new features. Prior studies in literature proposed that automated refactoring tools should be used to eliminate old constructs and idioms from source code, thereby improving code readability and maintainability while incorporating modern programming practices.

Pirkelbauer et al. [9] introduced the concept of source code rejuvenation as a source-to-source transformation process aimed at migrating legacy code to modern language constructs driven by programming language evolution. Using examples from the transition between C++03 and C++0x, this study demonstrates how rejuvenation replaces low-level patterns with higher-level abstractions, improving maintainability, security, and performance. The study distinguishes rejuvenation from traditional refactoring by emphasizing its reliance on language and library evolution and its allowance for behavior-improving transformations. While the approach highlights the potential of automated tooling, it also acknowledges limitations such as the need for human intervention, challenges in ensuring behavioral equivalence, and constraints in fully automating certain transformations.

Still regarding C++, Kumar et al. [24] analyzed how to rejuvenate old C++ programs by demacrofication, which means replacing preprocessor macros with modern C++11 constructs to make the code easier to read, maintain, and trust. The authors propose a semi-automated, multi-pass transformation approach supported by tooling that classifies macros and maps them to equivalent C++11 declarations, combining automated suggestions with iterative validation and developer intervention. Empirical evaluation across multiple large-scale C++ libraries demonstrates that a substantial proportion of macros can be transformed, although full automation is limited by challenges such as macro dependencies, ordering constraints, and the need for deeper compiler integration.

Regarding Java, Overbey and Johnson [8] suggested that refactoring tools become an integral part of language implementation, enabling proactive development of rejuvenation efforts alongside language changes. This study addresses the challenges of programming language evolution, particularly the accumulation of outdated constructs and increasing complexity that hinder maintainability and comprehension. The findings highlight that such tools can reduce the burden on developers by eliminating outdated constructs and facilitating code evolution, although challenges remain related to tool robustness, reliance on heuristics, handling of preprocessors, and developers' difficulty trusting the tools.

Franklin et al. [10] presented *LAMBDAFICATOR*, an automated refactoring tool designed to support the migration of imperative Java code to functional programming constructs introduced in Java 8, such as lambda expressions and stream operations. Empirical evaluation on multiple open-source projects demonstrates substantial applicability, code reduction, and improvements in readability, while also enabling opportunities for parallelism. However, limitations arise from the complexity of ensuring semantic equivalence and handling eager versus lazy execution semantics.

Similarly, Dantas et al. [11] investigated the practical viability of rejuvenating Java legacy systems by introducing modern language constructs through automated transformations, evaluating their acceptance via pull requests submitted to open-source projects. The findings reveal that simpler transformations with clear benefits are more readily accepted, while more complex ones—particularly lambda-based transformations—face resistance due to compatibility constraints and perceived lack of improvement, often resulting in false positives.

In the same way, Khatchadouriana and Masuhara [26] conducted empirical approach to assess the adoption of new programming language features by automatically introducing them into real-world projects and analyzing developer reactions. Focusing on Java 8 default methods, the authors employ a conservative refactoring tool to generate semantics-preserving transformations and submit them as pull requests to open-source repositories, enabling direct observation of acceptance and rejection decisions. The findings reveal that adoption is strongly influenced by contextual factors such as backward compatibility requirements, architectural constraints, and perceived risk, while acceptance is more likely when transformations are simple, localized, and align with existing design practices. The study highlights that even correct and minimally invasive transformations may be rejected if they do not provide clear benefits or violate project-specific constraints.

Regarding JavaScript, Gokhale et al. [74] presented *Desynchronizer*, a semi-automatic refactoring tool designed to support the migration from synchronous to asynchronous JavaScript APIs, addressing the trade-off between ease of use and performance in modern applications. The results shows that a large proportion of synchronous calls can be successfully transformed with minimal impact on program behavior and no observed degradation in test execution performance. However, the study highlights inherent limitations related to unsound static analysis in dynamic languages, incomplete test coverage, and the need for developer validation of suggested transformations.

In another context, Zhang et al. [75] investigated the prevalence of non-idiomatic Python code and proposes an automated refactoring approach to transform such code into idiomatic forms, aiming to improve readability, conciseness, and performance. The findings reveal that non-idiomatic patterns are widespread and that automated refac-

toring can achieve high accuracy while yielding measurable performance improvements. Empirical validation through pull requests demonstrates moderate developer acceptance, though some resistance arises due to concerns about readability, semantic nuances, and preferences for consistent, project-wide changes.

Finally, Rózsa et al. [76] proposed an automated framework to transform legacy Python conditional branching into structural pattern matching, a feature introduced in Python 3.10 to improve code readability and maintainability. Evaluation on multiple large-scale open-source Python projects demonstrates that the framework can successfully perform hundreds of transformations, with empirical evidence suggesting improvements in code readability, albeit subject to developer perception. The study also highlights limitations related to the subjectivity of readability gains, potential code duplication in certain transformations, and the need for configuration to avoid undesirable outcomes, such as introducing bugs or reducing maintainability in the codebase.

More recently, advances in large language models (LLMs) have introduced a complementary paradigm for automating code transformations. In recent years, LLMs have attracted substantial attention and have been widely adopted across various areas of software engineering [77, 78]. For example, May et al. [65] introduced *FreshBrew*, a benchmark designed to evaluate AI agents in the context of large-scale Java code migration, addressing the growing use of LLM-based systems for software modernization. The results reveal that, while modern AI agents can successfully perform a significant portion of migrations, their effectiveness decreases with project complexity and is constrained by issues such as dependency management, API incompatibilities, and behavioral failures. The study highlights both the potential and limitations of AI-driven code transformation, emphasizing the need for robust evaluation frameworks and human oversight.

Ala-Salmi et al. [79] proposed *VAPU*, a multi-agent system designed to autonomously modernize legacy codebases by orchestrating a pipeline of LLM-based agents that simulate collaborative software development roles. The approach decomposes modernization tasks into sequential steps involving task planning, execution, verification, and refinement, enabling iterative updates of outdated code while attempting to ensure correctness through feedback loops. Empirical evaluation on a legacy web application and multiple open-source projects shows that VAPU can achieve competitive or improved results compared to traditional prompt-based approaches, particularly for more complex tasks, although its effectiveness is influenced by factors such as model choice, temperature settings, and task complexity. The study also highlights limitations related to error propagation across agent chains, dependence on LLM capabilities, and challenges in verification accuracy.

Chen et al. [66] presented *DIFFPLOIT*, an LLM-based framework designed to automate the migration of vulnerability exploits across different versions of open-source

libraries, addressing the challenge of reproducing exploits in evolving software environments. The approach leverages a diff-driven, iterative process that combines failure analysis, contextual extraction, and guided code adaptation to incrementally refine exploits until successful reproduction is achieved. Empirical evaluation on a large dataset of real-world vulnerabilities demonstrates high effectiveness, significantly outperforming baseline approaches and reducing manual effort, while also uncovering previously undocumented affected versions. However, limitations arise in handling substantial API changes, specialized payload requirements, and variations in vulnerability behavior across versions.

Almeida et al. [62] evaluated the effectiveness of an agentic LLM-based approach, specifically GitHub Copilot Agent Mode, for automating library migration in Python applications, focusing on the upgrade of SQLAlchemy from version 1.x to 2.x. The results show that while the agent achieves high migration coverage and improvements in static code quality, it struggles to preserve functional correctness, as evidenced by relatively low test-pass rates and occasional regressions. These findings highlight the capability of LLM agents to identify and apply syntactic and structural updates, but also expose limitations in handling behavioral preservation and complex dependencies.

Amin et al. [64] introduced *JMigBench*, a benchmark designed to evaluate the capability of Large Language Models (LLMs) in performing fine-grained source code migration tasks, specifically from Java 8 to Java 11. The approach focuses on function-level transformations and assesses LLM performance using lexical and semantic similarity metrics, as well as correctness in removing deprecated constructs. Empirical results using the Mistral Codestral model indicate that LLMs can effectively handle simple, one-to-one API substitutions but struggle with more complex migrations involving multiple steps or deeper semantic changes, showing limited consistency and partial success in eliminating deprecated features. The study also highlights limitations related to the use of synthetic datasets and the absence of runtime validation, which restricts conclusions about real-world applicability.

Finally, Kebede et al. [63] presented *MigMate*, an LLM-based library migration tool integrated into the Visual Studio Code environment, aiming to automate and streamline the process of migrating Python projects across third-party libraries. The approach combines automated API mapping and code transformation with an interactive, human-in-the-loop interface that allows developers to review, validate, and selectively apply suggested changes, thereby addressing trust and usability concerns associated with fully automated tools. Empirical evaluation through a user study indicates substantial reductions in migration time and high usability scores, suggesting that integrating LLM-based transformations directly into developer workflows can improve efficiency and adoption. However, limitations include the lack of correctness evaluation, small sample size, and restricted

exposure to tool features.

The prior studies consistently highlighted that tool support plays an important role in assisting source code modernization and rejuvenation tasks. Automated and semi-automated approaches are commonly used to identify outdated constructs, suggest transformations, and facilitate the adoption of modern language features, leading to improvements in readability, maintainability, performance, and development efficiency. These tools are particularly effective in reducing manual effort and enabling large-scale transformations across codebases. However, the literature also reports consistent limitations. Tool support is usually not fully automated, requiring human validation and intervention to ensure correctness and behavioral preservation. Additional challenges include limited contextual understanding, dependency and compatibility constraints, incomplete transformations, and varying levels of developer acceptance, especially when perceived benefits are unclear or risks are high.

The emergent theory in this thesis supports these findings by confirming that tool usage is not fully automated and is instead applied in a selective, risk-aware, and human-controlled manner. It further expands prior work by situating tool support within a broader socio-technical context, where its use is shaped by organizational constraints, risk perception, effort considerations, and compatibility requirements. Rather than treating tools as standalone solutions, the theory frames them as one of several strategies developers adopt when performing rejuvenation, often in combination with planning, incremental execution, and technical debt management. In this view, perceived productivity improvements are associated not only with the capabilities of the tools themselves but also with how developers evaluate, control, and integrate tool-supported transformations into their workflows.

The theory also demonstrates that specific conditions may hinder or prevent the use of automated tool support. In particular, organizational and managerial constraints, such as prioritization of short-term delivery over internal improvements and governance or approval processes, may limit the adoption of automated transformations. Similarly, in critical or domain-sensitive systems, where stability, safety, and reliability are central requirements, developers tend to avoid or restrict the use of automation due to concerns about unintended side effects and the difficulty of validating behavioral equivalence. Risk, uncertainty, and fear factors also influence this decision, as developers may perceive automated changes as potentially introducing defects into otherwise stable systems.

Also, limitations on effort, cost, compatibility, and outside dependencies may make it even harder to use automated transformations on a large scale. Recent LLM-based approaches also exhibit similar limitations, such as concerns about correctness, dependency handling, and the necessity for human validation. In this sense, the results shown in the

theory suggest that the adoption of both traditional tools and LLM-based solutions is shaped by the same set of socio-technical conditions, which may constrain or discourage their use depending on the context.

5.4.2 Feature Adoption and Usage Analysis

A complementary line of research focuses on understanding how developers adopt and use programming language features, examining patterns of adoption over time and their impacts on software systems. For example, Parnin et al. [25] presented an empirical investigation of Java generics adoption, analyzing how this language feature has been integrated into open-source projects over time. The findings reveal that generics adoption is limited and uneven, often driven by a small group of developers rather than widespread community uptake, with most usage concentrated in specific contexts such as collections. Additionally, the study shows that large-scale migration of legacy code to generics is uncommon and that expected benefits, such as reduction in type casts, are not consistently observed. These results highlight that the introduction of new language features does not guarantee their adoption or systematic integration into existing systems.

Similarly, Dyer et al. [80] examined a large-scale empirical analysis of Java language feature adoption, examining how developers use, adopt, and transition to new language constructs over time. The findings reveal that developers often anticipate and adopt features before official release, but adoption is uneven, with some features widely used while others remain underutilized, and millions of missed opportunities where newer constructs could improve code quality. The study also shows that developers do update legacy code to incorporate modern features, although this process is selective and influenced by factors such as tooling support and developer familiarity.

Mazinanian et al. [81] investigated the adoption, usage patterns, and motivations behind the use of lambda expressions in Java through a mixed-methods approach combining large-scale repository mining and developer surveys. The findings reveal an increasing adoption trend over time, primarily driven by motivations such as improved readability and conciseness, while also showing that usage is often concentrated among a subset of developers and frequently introduced manually rather than through automated tools. The study also highlights inefficiencies in usage, limited awareness of performance implications, and variability in how developers leverage functional features.

In previous works of our research group, a study [19] examined whether the introduction of lambda expressions improves the comprehension of Java programs by combining quantitative analysis of code metrics with qualitative evaluations from developers and students. Using pairs of code snippets before and after lambda introduction, the authors assess readability and complexity through established models and human judgment. The

results reveal a discrepancy between quantitative and qualitative findings: while metric-based models show little or no improvement in readability or complexity, participants generally perceive lambda expressions as beneficial for comprehension, particularly in cases such as replacing anonymous inner classes. The study highlights that the impact of such transformations is context-dependent and not uniformly captured by existing metrics. Next, the work is expanded [20]. The following study highlighted that the effectiveness of such transformations is highly context-dependent, with certain uses (e.g., chaining stream operations) potentially harming readability and increasing debugging difficulty.

Regarding Python, Malloy and Power [22] presented an empirical analysis of the transition from Python 2 to Python 3, investigating how developers adopt new language features in the presence of backward incompatibility between versions. The findings reveal that developers tend to prioritize backward compatibility, often confining their code to a subset of features shared between Python 2 and Python 3, thereby limiting the adoption of newer constructs. While developers show willingness to adopt back-ported features within Python 2, the broader migration to Python 3 remains slow and incomplete due to compatibility constraints. These results highlight structural barriers to language evolution and the challenges of migrating legacy systems.

Peng et al. [28] examined a large-scale empirical analysis of Python language feature usage, examining how developers apply different constructs across real-world projects and domains. The findings reveal that a small subset of features dominates practical usage, while many advanced or recently introduced features, such as gradual typing remain underutilized. The study also shows that feature adoption varies significantly across domains, reflecting differing requirements and development practices. Additionally, the results highlight recurring usage patterns and missed opportunities to leverage certain features for improving code quality.

Vándor et al. [82] investigated the usefulness and developer preference of Python's structural pattern matching feature compared to traditional conditional branching through a controlled code review experiment. By analyzing developer evaluations of functionally equivalent code snippets generated via automated transformations, the authors assess readability, modifiability, and overall preference. The findings indicate a general preference for structural pattern matching, particularly in scenarios where it simplifies complex branching logic, although this preference is influenced by factors such as code length and individual developer judgment. The study also highlights that adoption decisions are not solely driven by technical benefits but are strongly shaped by subjective preferences and contextual factors.

Regarding Javascript, Alves et al. [83] conducted a comprehensive empirical analysis of functional programming concepts in JavaScript, investigating their prevalence, evolu-

tion, and correlation with code quality in practical projects. The authors examine the utilization of constructs such as recursion, immutability, lazy evaluation, and higher-order functions by analyzing millions of lines of code from various repositories, along with their correlation to bug-fixing endeavors. The results show that functional programming concepts are used a lot and tend to become more common over time. Most constructs, except for those related to immutability, are less likely to be removed in bug-fixing commits, which means they are less likely to cause bugs. The study also finds no proof that these kinds of structures make the code harder to understand. These results show that functional paradigms are becoming more important in modern JavaScript development and that they could help improve code quality.

Scarsbrook et al. [15] presented an empirical analysis of TypeScript feature adoption, examining how developers incorporate newly introduced language constructs over time in large-scale open-source projects. By mining hundreds of repositories and analyzing millions of commits, the authors investigate both the adoption speed of language versions and the usage patterns of specific syntactic features. The findings reveal that while TypeScript versions are rapidly adopted shortly after release, individual features exhibit highly variable adoption rates, with only a subset becoming widely used. This suggests that developers selectively adopt features based on perceived utility rather than uniformly embracing language evolution. The study also indicates that supporting a reduced subset of features may be sufficient for many tools and applications.

Nicolini et al. [84] investigated the adoption of new JavaScript language features through the use of transpilers, such as the Babel transpiler, to understand how developers leverage modern constructs while maintaining browser compatibility. By combining repository mining, Stack Overflow analysis, and compatibility data, the authors examine feature usage, developer challenges, and runtime constraints. The findings reveal that a significant portion of projects rely on Babel to enable early adoption of new features, although direct usage of proposal-stage features remains limited. Developers frequently encounter difficulties related to plug-in configuration and usage, particularly during development and setup phases. Additionally, while many modern features exhibit high browser compatibility, transpilers remain essential to avoid runtime issues in less supported environments.

Regarding C++, Uesbeck et al. [85] conducted a randomized controlled experiment evaluating the impact of C++ lambda expressions on developer performance and experience compared to traditional iterators. The findings indicate that lambda expressions do not provide performance benefits in the studied context and may negatively affect less experienced developers, leading to increased errors and longer completion times. While experienced developers perform consistently regardless of the construct used, the results

highlight that the adoption of new language features can introduce usability challenges and learning overhead.

Finally, Chen et al. [86] presented a large-scale empirical analysis of C++ template usage, examining how developers adopt and apply generic programming constructs in real-world systems. The findings show that templates are effectively used to reduce code duplication, although their benefits are often concentrated in a small number of widely reused abstractions. Additionally, adoption is uneven and follows a power-law distribution, with a limited subset of developers contributing most template usage, while legacy constructs such as C-style generics and virtual functions remain prevalent. The study also highlights that large-scale refactoring of existing code to adopt templates is uncommon, indicating limited migration of legacy constructs.

Prior studies on language feature adoption consistently reported that the adoption process is selective, uneven, and highly context-dependent, rather than being an automatic consequence of feature release. The emergent theory supports these findings by showing that source code rejuvenation is shaped by interacting conditions such as compatibility constraints, effort and cost limitations, risk and uncertainty, human and knowledge factors, and perceived relevance, all of which help explain why some features remain underused, why migration opportunities are missed, and why expected benefits are not always realized.

At the same time, the theory expands prior work by moving beyond adoption patterns to explain how developers respond to these barriers in practice. In particular, the contingencies identified in Chapter 4 suggest that some of the limitations emphasized in earlier studies may be reduced through strategies such as careful planning, incremental execution, strategic evaluation of risks and benefits, selective and human-controlled use of tool support, knowledge sharing within teams, management of technical constraints and compatibility, and validation through testing and CI/CD processes. Rather than assuming that these strategies eliminate the barriers, the theory shows how they can help developers operationalize rejuvenation more safely and deliberately under restrictive conditions, while also clarifying why adoption remains heterogeneous across projects and features.

In addition, the three mining repository large-scale studies in C++, JavaScript, and Python that compose the empirical basis of the theory reinforce and extend previous adoption research. Like earlier work, they identify heterogeneous adoption trajectories, measurable delays between feature release and broader use, and selective adoption of modern constructs. However, they go further by providing large-scale empirical evidence of efforts to rejuvenate source code in practice and by integrating these temporal patterns

with qualitative evidence from C++ survey responses and JavaScript and Python pull-request discussions.

This combination makes it possible to move from describing adoption trends to explaining the phenomenon itself: not only that feature adoption is uneven, but also how, when, and why developers rejuvenate legacy code, what challenges hinder them, and which practices they use to address those challenges. In this sense, the theory in this thesis expands prior studies by grounding adoption heterogeneity, motivations, constraints, and strategies in a single socio-technical explanation derived from mixed-method empirical evidence.

5.4.3 Software Modernization and Migration Studies

Prior research on software modernization and migration has primarily focused on understanding system-level transformations, encompassing architectural evolution, platform migration, and organizational drivers. These studies investigated how legacy systems are adapted through approaches such as reengineering, incremental migration, and large-scale system restructuring, while also examining the expected benefits, challenges, and outcomes of such efforts.

Khadka et al. [87] conducted a study investigates how industry practitioners perceive legacy systems and their modernization, combining grounded theory interviews with a large-scale survey to capture both qualitative insights and quantitative validation. By analyzing perspectives from professionals across diverse domains, the authors identify key benefits, drivers, challenges, and practices associated with legacy systems and modernization efforts. The findings reveal that legacy systems are widely regarded as business-critical, reliable, and stable systems that support core organizational processes, often benefiting from proven technology and long-term operational maturity. Modernization is primarily driven by the need for agility, reduced maintenance costs, faster time-to-market, and the scarcity of expertise required to maintain legacy technologies. At the same time, significant challenges emerge, including complex and monolithic architectures, data migration difficulties, lack of documentation and expertise, organizational resistance to change, and constraints related to time, cost, and business alignment.

In terms of practices and strategies, the study highlights that modernization is commonly performed through a combination of approaches such as incremental migration, reverse engineering, reengineering, and the use of middleware and wrapping techniques, often targeting architectural transformations such as migration to service-oriented architectures (SOA). These approaches emphasize gradual system evolution, coexistence between legacy and modern components, and the need to balance technical decisions with organizational and business considerations. Overall, the study shows that modern-

ization is not purely a technical process but a socio-technical endeavor shaped by both engineering constraints and organizational dynamics.

Next, in another study, Khadka et al. [29] presented a retrospective analysis of software modernization through multiple industrial case studies, focusing on the alignment between expected and actual post-modernization outcomes. By examining project documentation and conducting interviews across diverse domains, the authors assess whether modernization efforts achieve their intended technical and business goals. The findings indicate that most anticipated benefits—such as improved maintainability, increased flexibility, enhanced usability, reduced operational and maintenance costs, and better system integration—are generally achieved, although some objectives, particularly the complete elimination of legacy technologies, are only partially fulfilled.

Beyond these expected outcomes, the study also reports several unintended consequences. Positive effects include increased organizational transparency, improved collaboration, higher system availability, and enhanced business flexibility. However, negative effects are also observed, such as user resistance to new systems, performance degradation compared to legacy implementations, and adaptation challenges during early adoption phases. These findings highlight that modernization outcomes extend beyond purely technical improvements and involve significant organizational and human factors.

In terms of modernization practices, the cases reveal a combination of strategies centered on architectural transformation and system restructuring. These include user interface modernization (e.g., migration from character-based interfaces to graphical interfaces), system consolidation, decomposition of monolithic systems into modular or layered architectures, re-hosting to modern infrastructures (e.g., virtualization environments), migration of legacy platforms (e.g., COBOL or COTS systems to modern platforms such as Oracle-based or service-oriented architectures), and enablement of new delivery models such as Software-as-a-Service (SaaS). These modernization efforts often involve incremental evolution, reuse of existing components, and separation of concerns through layered designs. Overall, the study reinforces that modernization is a socio-technical process, where technical transformations are tightly coupled with organizational dynamics, user adoption, and business strategy.

Finally, Bragagnolo et al. [30] proposed a theoretical framework for software modernization, aiming to clarify the conceptual ambiguity surrounding migration, reengineering, and related processes through a systematic literature review combined with grounded theory analysis. By synthesizing evidence from 30 selected studies, the authors construct a set of bottom-up taxonomies that define key concepts, relationships, and dimensions structuring modernization efforts, including distinctions between reengineering, migra-

tion, adaptation, and replacement, as well as classifications of approaches such as black-box, white-box, and grey-box techniques.

The framework identifies that modernization is driven by the progressive decline of systems, characterized by decadence (e.g., increasing complexity, lack of documentation, and reduced maintainability) and obsolescence (e.g., technological shifts and environmental changes), which together create pressure for system evolution. In this context, modernization aims to restore system quality, improve maintainability, enable compatibility with modern technologies, and support continued system evolution.

The study also highlights several barriers and risks associated with migration processes. These include the heterogeneity of systems and contexts, ambiguity in terminology across studies, lack of standardized definitions, dependency on specific project circumstances (e.g., architecture, technologies, and business constraints), and the inherent risks of failure in large-scale migration efforts. Additionally, knowledge availability, system complexity, and unclear process definitions are identified as key factors that complicate modernization initiatives.

In terms of practices, the framework emphasizes that successful modernization relies on iterative and incremental processes, allowing gradual transformation, validation, and risk mitigation. It also stresses the importance of validation and verification activities throughout the process to ensure correctness and alignment with objectives. The framework further highlights that modernization involves multiple interacting elements, including legacy system characteristics, drivers, objectives, processes, and architectural transformations.

The study categorizes different types of migration and modernization approaches, including architectural migration (e.g., monolithic to microservices), user interface migration (e.g., desktop to web-based systems), data migration, and language or platform migration. It also distinguishes between different strategic approaches such as black-box (preserving internal structure), white-box (deep transformation and restructuring), and grey-box (hybrid approaches combining both strategies). These approaches reflect different levels of knowledge requirements and intervention in the system.

Overall, the framework provides a structured conceptual foundation for understanding software modernization, emphasizing the need for unified terminology, systematic classification of approaches, and alignment between migration processes and core software engineering principles such as iterativity, incrementality, and validation.

While prior studies on software modernization and migration give useful insights into system-level transformations, they fundamentally differ from the present thesis in both scope and analytical depth. First, none of the analyzed studies explicitly address source code rejuvenation. Instead, they focus on architectural migration, platform evolution,

and organizational drivers, leaving the fine-grained evolution of source code largely unexplored. In contrast, this thesis introduces a novel and complementary perspective by conceptualizing source code rejuvenation as a specific form of software migration, grounded in empirical evidence and explained through a socio-technical theory that captures how, when, and why developers replace legacy language constructs with modern alternatives.

Second, although these studies identify benefits such as improved maintainability, flexibility, cost reduction, and system longevity, as well as barriers including system complexity, lack of expertise, organizational resistance, and migration risks, their analyses remain primarily at the system and architectural levels. The emergent theory corroborates many of these factors but significantly extends their interpretation by revealing how similar benefits, challenges, and practices manifest at the code level during source code rejuvenation efforts. In particular, this thesis shows that practices such as incremental evolution, risk-aware decision-making, and continuous validation, commonly reported in modernization, also govern source code rejuvenation, albeit under different constraints related to language evolution, developer cognition, and tooling limitations. By incorporating rejuvenation as an additional and previously underexplored form of migration, this work broadens the understanding of software evolution, bridging the gap between high-level modernization strategies and the micro-level transformations that sustain long-term system adaptability.

5.5 Limitations

This thesis adopts the Total Quality Framework (TQF) [88] as the primary lens to reason about research quality and limitations. Unlike traditional approaches in software engineering that structure limitations in terms of internal, external, and construct validity, the TQF provides a paradigm-neutral and qualitative-oriented perspective that is better aligned with the socio-technical grounded theory methodology employed in this thesis.

In qualitative research, concepts such as reliability and validity require reinterpretation to account for the interpretive, contextual, and non-deterministic nature of the data [89]. As such, rather than framing limitations as threats to predefined forms of validity, this study discusses them in terms of credibility, analyzability, transparency, and usefulness, which collectively reflect the degree of confidence that can be placed in the findings. This perspective allows for a more appropriate and nuanced assessment of limitations in studies that aim to generate theory from empirical and context-dependent evidence.

The following subsections discuss the study limitations with respect to *credibility* (Section 5.5.1), *analyzability* (Section 5.5.2), *transparency* (Section 5.5.3), and *usefulness* (Section 5.5.4).

5.5.1 Credibility

Regarding *credibility*, this study relies on a combination of semi-structured interviews and repository mining studies to construct the emergent theory of source code rejuvenation. While this mixed-methods design strengthens the robustness of the findings through triangulation, it also introduces potential biases associated with each data source.

With respect to data gathering, interview data reflects participants' perceptions and experiences, which may be influenced by recall bias or subjective interpretation. To mitigate this limitation, participants were selected based on direct experience with legacy systems and language evolution, ensuring that their accounts were grounded in practical scenarios. In addition, interview questions were designed to elicit concrete examples of rejuvenation practices, encouraging participants to describe specific situations rather than abstract opinions.

Similarly, mining-based analyses capture observable patterns in code repositories but may not fully represent the underlying intentions or contextual reasoning behind developers' actions. To address this limitation, mining results were not interpreted in isolation but complemented with qualitative evidence from interviews, pull request discussions, and survey responses. This triangulation allowed us to relate observable code changes to developers' motivations and challenges, strengthening the completeness and interpretability of the data.

Furthermore, the integration of multiple data sources (interviews, mining studies, and discussion artifacts) supported the expansion and refinement of emerging concepts through constant comparison and iterative analysis, reducing the reliance on a single perspective. This integration enabled the strengthening and densification of categories by relating empirical evidence across different contexts and data types. Although these strategies enhance the credibility of the study, the empirical evidence remains subject to the inherent limitations of each data source, and some aspects of developers' reasoning may not be fully captured.

5.5.2 Analyzability

In terms of *analyzability*, the construction of the theory through socio-technical grounded theory involves interpretive analysis, which inherently introduces subjectivity in how data is processed and interpreted. Although systematic procedures such as constant comparison, memoing, and theoretical saturation were applied to enhance analytical rigor, the abstraction of codes into concepts and categories depends on the researchers' analytical judgment.

Another limitation concerns the preparation and processing of the interview data. Some interviews were conducted in Brazilian Portuguese and were later translated into English, while interviews conducted directly in English were only transcribed. This decision supported the use of a unified analytical language throughout the coding process, but it may have introduced translation-related limitations, such as the loss of nuance, meaning, or emphasis present in the original accounts.

The coding and analysis were supported by ATLAS.ti, a professional qualitative data analysis tool that facilitated the organization of data, collaborative analysis, traceability of codes, and systematic comparison across interviews and other empirical materials. Initial coding was conducted by the author of this thesis. These initial codes were then subjected to multiple rounds of comparison, revision, and discussion with researchers experienced in software engineering, source code transformation, and software evolution. The codes and concepts were reviewed, analyzed, and discussed in dedicated analytical meetings, some of which were recorded to support traceability and later verification of interpretive decisions. This collaborative process helped refine interpretations, challenge assumptions, and improve the consistency of the emerging concepts.

Another analyzability limitation concerns the interpretation of covariances among the categories of the emergent theory. In this thesis, covariances were used to describe recurring relationships in which variation in one category was associated with variation in another, such as relationships between causes and consequences, conditions and contingencies, or contingencies and consequences. However, these covariances should not be interpreted as causal, deterministic, or statistically validated relationships. Their identification depended on theoretical comparison, memoing, and the researchers' interpretive judgment over the empirical material. To mitigate this limitation, covariances were included only when they were repeatedly grounded in the data and useful for explaining the socio-technical dynamics of source code rejuvenation.

A further limitation concerns the timing of the analysis in the first round of interviews. In socio-technical grounded theory, data collection and analysis are expected to proceed iteratively, with emerging concepts guiding subsequent data collection through theoretical sampling. However, for the first 23 interviews in Round 1, the decision to begin the analysis only after the transcription and translation of the interviews may have limited the extent to which early analytical insights could immediately inform subsequent interviews. As a result, some opportunities for probing emerging concepts, refining questions, or exploring unexpected issues may have been missed during this first round.

To mitigate this limitation, the gaps identified during the first round were explicitly considered in the design of subsequent empirical steps. Round 2, composed of mining studies, expanded the empirical basis of the theory by examining how modern language

features are adopted across different programming languages and software ecosystems. Round 3 introduced additional interviews as interview-based theoretical sampling, including a new question on the role of LLM support in source code rejuvenation and migration activities. These later rounds helped revisit, complement, and refine issues that remained insufficiently explored after the first round of interviews. Nevertheless, it is still possible that some analytical opportunities were lost because the first round did not fully follow a batch-based iterative analysis process from the beginning.

Throughout the overall analytical process, data from multiple sources were continuously compared and integrated, allowing emerging concepts to be refined and adjusted across different contexts. Multiple rounds of coding and analysis were supported by memoing to document analytical decisions and ensure traceability of interpretations. In addition, the integration of diverse data sources, including interviews, mining studies, surveys, and discussion artifacts, supported the refinement of categories by relating evidence across different types of data.

Despite these mitigation strategies, alternative interpretations of the data may still be possible, which may influence the consistency and verification of the analytical outcomes. As with most grounded theory studies, the resulting theory reflects a plausible and well-supported interpretation of the data, rather than a single definitive representation of the phenomenon.

5.5.3 Transparency

Concerning *transparency*, this study provides detailed documentation of the research design, data collection, and analysis procedures to enable readers to assess the rigor and applicability of the findings. Following the principles of transparency in qualitative research, thick descriptions are provided throughout the thesis to support the interpretation and evaluation of the results.

Specifically, the research methodology applied is fully documented in Chapter 3, where the multi-study design and the application of Socio-Technical Grounded Theory (STGT) are detailed. This includes the description of data collection procedures (semi-structured interviews, repository mining, survey responses, and pull request discussions), as well as the iterative processes of open and focused coding, constant comparison, theoretical sampling, and memoing that guided theory construction.

The findings and the resulting theoretical constructs are presented in detail in Chapter 4, where concepts, categories, and their relationships are grounded in empirical evidence collected across multiple data sources. The theory is structured using the Six C's framework, explicitly articulating contexts, conditions, causes, contingencies, covariances,

and consequences, supported by qualitative excerpts and patterns observed in large-scale mining analyses.

In addition, Section 5.2 presents hypotheses derived from the emergent theory, making explicit how the relationships among contexts, causes, conditions, contingencies, consequences, and covariances. These hypotheses are not treated as predefined propositions tested in this thesis, but as theoretically grounded outcomes of the analytical process that increase the transparency of the theory's explanatory scope and support future theory refinement. Then, the implications of the proposed theory are discussed in Section 5.3, where the findings are translated into practical and theoretical insights for software developers and organizations. These implications provide guidance on how source code rejuvenation can be understood, prioritized, and operationalized in evolving software systems.

To further enhance transparency and reproducibility, the datasets generated and analyzed in this thesis are publicly available in a dedicated replication repository.[1] This repository includes raw and processed data, as well as supporting materials used in the analyses, enabling other researchers to inspect, replicate, or extend the study.

Finally, while this study aims to provide a comprehensive socio-technical explanation of source code rejuvenation, it does not seek to establish deterministic causal relationships. Instead, the proposed theory captures patterns of association and interaction among categories within specific contexts. As such, the findings should be interpreted as an explanatory and context-sensitive framework that supports understanding and reasoning about the phenomenon, rather than as universally predictive rules.

5.5.4 Usefulness

Regarding *usefulness*, the sampling strategy, although theoretically grounded and iterative, may limit the generalizability and transferability of the findings. Participants were selected based on their experience with long-lived systems and language evolution, which provides depth but may not fully represent all developer profiles or organizational contexts. In addition, the mining studies focused on specific programming languages (e.g., C++, JavaScript, and Python), which may influence the observed practices and patterns. While these languages were selected to ensure diversity and relevance, the applicability of the findings to other environments with different evolution dynamics may be limited.

Despite these limitations, the proposed theory is not intended to provide universal generalizations, but rather a context-sensitive and extensible framework that can be applied, examined, and refined in different settings. This theory can and should be further

[1] `https://github.com/waltim/socio-technical-grounded-theory-scr`

evolved through its application in other organizational contexts, across different programming languages, and with diverse developer populations worldwide. Such extensions may enrich, refine, or expand the identified categories and their relationships, strengthening the explanatory power of the theory over time.

Finally, still in terms of *usefulness*, the identification of contexts, conditions, causes, contingencies, and consequences reflects the data available at the time of the study. As programming languages, tools, and development practices continue to evolve, new factors may emerge that extend or reshape the proposed theory. In particular, the increasing adoption of automated tooling and large language models may influence how source code rejuvenation is performed in practice, potentially altering some of the observed strategies and dynamics over time.

Nevertheless, the findings provide actionable insights that can support developers and organizations in managing software aging. By recognizing early signals of system degradation, evaluating risks, costs, and effort, and understanding the potential benefits of rejuvenation, practitioners can make more informed and proactive decisions. In this sense, while not generalizable in a strict sense, the theory offers practical guidance to improve software quality and longevity in systems that continuously evolve alongside rapidly changing programming languages.

Despite these limitations, the use of a mixed-methods design, cross-language analysis, and systematic grounded theory procedures provided a strong foundation for the emerged theory, offering both analytical depth and practical relevance for understanding source code rejuvenation in evolving software systems.

Chapter 6

Conclusions

This thesis investigated the phenomenon of source code rejuvenation in contemporary software engineering practice through a socio-technical grounded theory approach supported by a multi-method design. The study integrated qualitative and quantitative data sources. First, semi-structured interviews with software practitioners were conducted from diverse organizational contexts, as summarized in Table D.1. These software contexts reflect environments where long-lived systems and legacy codebases are common, providing fertile ground for source code rejuvenation practices.

Second, as part of the advanced stage of the STGT process, theoretical sampling was conducted to address gaps, expand the empirical basis of the theory, and contrast the categories and concepts that emerged from the first round of interviews. In Round 2, large-scale mining studies of open-source repositories in three programming languages, C++, JavaScript, and Python, were conducted. These studies involved a quantitative examination of commit histories to identify the adoption of modern language features over time, enabling the characterization of longitudinal patterns of source code rejuvenation across projects. This quantitative analysis was complemented by evidence from a survey with C++ developers, which captured perceptions, challenges, and practices related to the adoption of modern language features.

In addition, qualitative analyses of developer discussions in pull requests from JavaScript and Python systems were performed to examine the reasoning, trade-offs, and social dynamics involved in adopting modern language features in collaborative settings. Round 3 then extended the theoretical sampling process through additional semi-structured interviews with practitioners, allowing the study to revisit unresolved issues from the first round, further examine practical constraints and decision-making processes, and explore the emerging role of LLM-based support in source code rejuvenation and migration activities.

These data sources were not used in isolation but analyzed and constantly compared,

allowing the triangulation and progressive refinement of the emerging theory. This integration enabled a comprehensive empirical grounding of the theory, capturing the phenomenon from multiple complementary socio-technical perspectives. This thesis resulted in a socio-technical grounded theory that explains how, when, and why source code rejuvenation occurs in practice. The theory characterizes source code rejuvenation as a situated maintenance response rather than a uniform, purely technical, or automatically triggered activity. It shows that rejuvenation emerges from the interaction of technical, organizational, and human factors, structured through the relationships among contexts, causes, conditions, contingencies, and consequences.

The theory explains that source code rejuvenation tends to occur under pressures related to system obsolescence, lack of support, security risks, dependency evolution, language and ecosystem changes, and increasing maintenance complexity. At the same time, it shows that rejuvenation is often justified by expected or observed improvements in code readability, expressiveness, reliability, correctness, performance, maintainability, testability, and development productivity. However, its feasibility is not determined only by technical opportunity. It is also constrained or enabled by organizational priorities, compatibility demands, available testing and validation infrastructure, developers' knowledge, risk perception, trust in tool support, and the perceived relevance of rejuvenation efforts.

In this sense, the theory explains source code rejuvenation as an adaptive and negotiated process. Developers and organizations decide whether, when, and how to rejuvenate code by balancing rejuvenation benefits against risks, costs, and practical constraints. When rejuvenation is postponed or restricted, legacy constructs, outdated dependencies, and unsupported technologies may contribute to system degradation. When it is performed, its execution depends on strategies such as incremental changes, selective automation, human review, continuous validation, and technical debt management. Thus, the theory explains source code rejuvenation as a socio-technical process through which developers and organizations negotiate rejuvenation, risk, quality, and long-term system viability in evolving software ecosystems.

Regarding recommendations for future works, the researchers may focus on extending and refining the theory across additional programming language environments not covered in the current mining studies, enabling a broader expansion of the findings. Moreover, as a grounded theory, the proposed model can continue to evolve through the incorporation of new empirical evidence, contexts, and technological advancements.

Furthermore, applying the theory in different organizational settings, development teams, and software domains may better understand how socio-technical factors vary across contexts. This includes investigating which developer profiles, programming lan-

guages, ecosystems, and types of systems are more likely to be impacted by emerging tool support in source code rejuvenation. For example, the impact of LLM-based support may vary depending on the ecosystem and system criticality. In more dynamic ecosystems, such as JavaScript and Python, where language features, frameworks, and dependencies evolve rapidly, LLMs may have a stronger influence on rejuvenation practices. In contrast, in critical systems or highly regulated environments, the impact of LLMs may be more limited due to stricter requirements for validation, safety, traceability, and human oversight.

An important research direction also involves validating which socio-technical aspects and implications of the proposed theory in this thesis remain stable or need refinement in light of the increasing adoption of large language models. For instance, future studies may examine whether LLMs reduce, reinforce, or reshape existing challenges related to knowledge gaps among team members, trust in automated support, code review practices, risk perception, and decision-making processes. Such investigations may help determine whether LLMs primarily support existing rejuvenation strategies, such as selective automation and human review, or whether they introduce new practices, constraints, and organizational dynamics.

Reference

[1] Scott Meyers. *Effective modern C++: 42 specific ways to improve your use of C++ 11 and C++ 14.* " O'Reilly Media, Inc.", 2014. xiv, 14, 15

[2] Rashina Hoda. Qualitative research with socio-technical grounded theory a practical guide to qualitative data analysis and theory development in the digital world. *Innovations*, 2025. xiv, 19, 21, 23, 24, 34, 36, 99

[3] Walter Lucas, Fausto Carvalho, Rafael Campos Nunes, Rodrigo Bonifácio, João Saraiva, and Paola R. G. Accioly. Embracing modern C++ features: An empirical assessment on the KDE community. *J. Softw. Evol. Process.*, 36(5), 2024. xiv, 2, 4, 14, 15, 21, 22, 23, 31, 33, 73, 74, 84

[4] Walter Lucas, Rafael Campos Nunes, Rodrigo Bonifácio, Fausto Carvalho, Ricardo Lima, Michael Silva, Adriano Torres, Paola R. G. Accioly, Eduardo Monteiro, and João Saraiva. Understanding the adoption of modern javascript features: An empirical study on open-source systems. *Empir. Softw. Eng.*, 30(3):107, 2025. xiv, 4, 13, 21, 22, 23, 31, 33, 74, 84

[5] Walter Mendonça, Marcos Leite, Oseias Romeiro, Fausto Carvalho, Rodrigo Bonifácio, Eduardo Monteiro, Gustavo Pinto, Paola Accioly, and João Saraiva. "it makes the code clearer": Why developers adopt modern python features in open source projects, 2025. Preprint, not peer reviewed. Available at: https://ssrn.com/abstract=6300278. xiv, 16, 21, 22, 23, 31, 33, 68, 73, 74, 84

[6] Barney G Glaser. *Advances in the methodology of grounded theory: Theoretical sensitivity.* University of California, 1978. xiv, 23, 36, 37, 99

[7] Walter Lucas. A socio-technical grounded theory about source code rejuvenation: Data, codebook, and supporting artifacts. `https://github.com/waltim/socio-technical-grounded-theory-scr`, 2026. Accessed: 2026-05-13. xv, 25, 27, 39, 40, 240

[8] Jeffrey L. Overbey and Ralph E. Johnson. Regrowing a language: refactoring tools allow programming languages to evolve. In Shail Arora and Gary T. Leavens, editors, *Proceedings of the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2009, October 25-29, 2009, Orlando, Florida, USA*, pages 493–502. ACM, 2009. 1, 2, 114, 119

[9] Peter Pirkelbauer, Damian Dechev, and Bjarne Stroustrup. Source code rejuvenation is not refactoring. In *International Conference on Current Trends in Theory and Practice of Computer Science*, pages 639–650. Springer, 2010. 1, 2, 11, 12, 119

[10] Lyle Franklin, Alex Gyori, Jan Lahoda, and Danny Dig. LAMBDAFICATOR: from imperative to functional programming through automated refactoring. In David Notkin, Betty H. C. Cheng, and Klaus Pohl, editors, *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, pages 1287–1290. IEEE Computer Society, 2013. 1, 2, 120

[11] Reno Dantas, Antonio Carvalho, Diego Marcilio, Luisa Fantin, Uriel Silva, Walter Lucas, and Rodrigo Bonifácio. Reconciling the past and the present: An empirical study on the application of source code transformations to automatically rejuvenate java programs. In Rocco Oliveto, Massimiliano Di Penta, and David C. Shepherd, editors, *25th International Conference on Software Analysis, Evolution and Reengineering, SANER 2018, Campobasso, Italy, March 20-23, 2018*, pages 497–501. IEEE Computer Society, 2018. 1, 2, 12, 32, 120

[12] Pedro V. R. de Carvalho, Rodrigo Bonifácio, Walter Lucas, and Alana Paula Barbosa Mota. Mining discussions on software migration: A study of the boost mailing list regarding C++ code evolution. In Ivan Machado, José Carlos Maldonado, Tayana Conte, Edna Dias Canedo, Johnny Marques, Breno Bernard Nicolau de França, Patrícia Matsubara, Davi Viana, Sérgio Soares, Gleison Santos, Larissa Rocha, Bruno Gadelha, Rodrigo Pereira dos Santos, Ana Carolina Oran, and Adolfo Gustavo Serra Seca Neto, editors, *Proceedings of the XXIII Brazilian Symposium on Software Quality, SBQS 2024, Salvador, Bahia, Brazil, November 5-8, 2024*, pages 264–274. ACM, 2024. 1

[13] Raoul-Gabriel Urma, Mario Fusco, and Alan Mycroft. *Java 8 in Action: Lambdas, Streams, and Functional-Style Programming*. Manning Publications Co., USA, 1st edition, 2014. 1, 32

[14] Brian A. Malloy and James F. Power. Quantifying the transition from python 2 to 3: An empirical study of python applications. In Ayse Bener, Burak Turhan, and Stefan Biffl, editors, *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM 2017, Toronto, ON, Canada, November 9-10, 2017*, pages 314–323. IEEE Computer Society, 2017. 1, 2

[15] Joshua D. Scarsbrook, Mark Utting, and Ryan K. L. Ko. Typescript's evolution: An analysis of feature adoption over time. In *20th IEEE/ACM International Conference on Mining Software Repositories, MSR 2023, Melbourne, Australia, May 15-16, 2023*, pages 109–114. IEEE, 2023. 1, 126

[16] Bjarne Stroustrup. Thriving in a crowded and changing world: C++ 2006-2020. *Proc. ACM Program. Lang.*, 4(HOPL):70:1–70:168, 2020. 1, 14

[17] Allen Wirfs-Brock and Brendan Eich. Javascript: The first 20 years. *Proc. ACM Program. Lang.*, 4(HOPL), jun 2020. 1

[18] Timothy L Staples. Expansion and evolution of the r programming language. *Royal Society Open Science*, 10(4):221550, 2023. 1

[19] Walter Lucas, Rodrigo Bonifácio, Edna Dias Canedo, Diego Marcilio, and Fernanda Lima. Does the introduction of lambda expressions improve the comprehension of java programs? In Ivan do Carmo Machado, Rodrigo Souza, Rita Suzana Pitangueira Maciel, and Cláudio Sant'Anna, editors, *Proceedings of the XXXIII Brazilian Symposium on Software Engineering, SBES 2019, Salvador, Brazil, September 23-27, 2019*, pages 187–196. ACM, 2019. 1, 2, 32, 124

[20] Walter Lucas Monteiro de Mendonça, José Fortes Neto, Francisco Vitor Lopes, Diego Marcilio, Rodrigo Bonifácio, Edna Dias Canedo, Fernanda Lima, and João Saraiva. Understanding the impact of introducing lambda expressions in java programs. *J. Softw. Eng. Res. Dev.*, 8, 2020. 1, 125

[21] J-M Favre. Languages evolve too! changing the software time scale. In *Eighth International Workshop on Principles of Software Evolution (IWPSE'05)*, pages 33–42. IEEE, 2005. 1

[22] Brian A. Malloy and James F. Power. An empirical analysis of the transition from python 2 to python 3. *Empir. Softw. Eng.*, 24(2):751–778, 2019. 1, 2, 125

[23] Luca Di Grazia and Michael Pradel. The evolution of type annotations in python: an empirical study. In Abhik Roychoudhury, Cristian Cadar, and Miryung Kim, editors, *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022*, pages 209–220. ACM, 2022. 1, 2

[24] Aditya Kumar, Andrew Sutton, and Bjarne Stroustrup. Rejuvenating C++ programs through demacrofication. In *28th IEEE International Conference on Software Maintenance, ICSM 2012, Trento, Italy, September 23-28, 2012*, pages 98–107. IEEE Computer Society, 2012. 1, 2, 119

[25] Chris Parnin, Christian Bird, and Emerson R. Murphy-Hill. Java generics adoption: how new features are introduced, championed, or ignored. In Arie van Deursen, Tao Xie, and Thomas Zimmermann, editors, *Proceedings of the 8th International Working Conference on Mining Software Repositories, MSR 2011 (Co-located with ICSE), Waikiki, Honolulu, HI, USA, May 21-28, 2011, Proceedings*, pages 3–12. ACM, 2011. 2, 32, 124

[26] Raffi Khatchadourian and Hidehiko Masuhara. Proactive empirical assessment of new language feature adoption via automated refactoring: The case of java 8 default methods. *Art Sci. Eng. Program.*, 2(3):6, 2018. 2, 120

[27] Oscar Callaú, Romain Robbes, Éric Tanter, and David Röthlisberger. How developers use the dynamic features of programming languages: the case of smalltalk. In Arie van Deursen, Tao Xie, and Thomas Zimmermann, editors, *Proceedings of the 8th International Working Conference on Mining Software Repositories, MSR 2011 (Co-located with ICSE), Waikiki, Honolulu, HI, USA, May 21-28, 2011, Proceedings*, pages 23–32. ACM, 2011. 2

[28] Yun Peng, Yu Zhang, and Mingzhe Hu. An empirical study for common language features used in python projects. In *28th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2021, Honolulu, HI, USA, March 9-12, 2021*, pages 24–35. IEEE, 2021. 2, 125

[29] Ravi Khadka, Prajan Shrestha, Bart Klein, Amir Saeidi, Jurriaan Hage, Slinger Jansen, Edwin van Dis, and Magiel Bruntink. Does software modernization deliver what it aimed for? A post modernization analysis of five software modernization case studies. In Rainer Koschke, Jens Krinke, and Martin P. Robillard, editors, *2015 IEEE International Conference on Software Maintenance and Evolution, ICSME 2015, Bremen, Germany, September 29 - October 1, 2015*, pages 477–486. IEEE Computer Society, 2015. 2, 129

[30] Santiago Bragagnolo, Nicolas Anquetil, Stéphane Ducasse, Abderrahmane Seriai, and Mustapha Derras. Software Migration: A Theoretical Framework. Research report, Inria Lille Nord Europe - Laboratoire CRIStAL - Université de Lille, 2021. 2, 10, 129

[31] David Lorge Parnas. Software aging. In Bruno Fadini, Leon J. Osterweil, and Axel van Lamsweerde, editors, *Proceedings of the 16th International Conference on Software Engineering, Sorrento, Italy, May 16-21, 1994*, pages 279–287. IEEE Computer Society / ACM Press, 1994. 7

[32] Zahir Irani, Raul-Mario Abril-Jimenez, Vishanth Weerakkody, Amizan Omar, and Uthayasankar Sivarajah. The impact of legacy systems on digital transformation in european public administration: Lesson learned from a multi case analysis. *Gov. Inf. Q.*, 40(1):101784, 2023. 7

[33] Hans Christian Benestad, Bente Anda, and Erik Arisholm. Understanding software maintenance and evolution by analyzing individual changes: a literature review. *J. Softw. Maintenance Res. Pract.*, 21(6):349–378, 2009. 8

[34] International Organization for Standardization (ISO), International Electrotechnical Commission (IEC), and Institute of Electrical and Electronics Engineers (IEEE). ISO/IEC/IEEE 14764:2022 — Software Engineering — Software Life Cycle Processes — Maintenance, 2022. International Standard, Third edition, Geneva, Switzerland. 8

[35] Keith H. Bennett and Václav Rajlich. Software maintenance and evolution: a roadmap. In Anthony Finkelstein, editor, *22nd International Conference on on Software Engineering, Future of Software Engineering Track, ICSE 2000, Limerick Ireland, June 4-11, 2000*, pages 73–87. ACM, 2000. 8

[36] Bennet P. Lientz. Issues in software maintenance. *ACM Comput. Surv.*, 15(3):271–278, 1983. 8

[37] Thomas M. Pigoski. *Practical Software Maintenance: Best Practices for Managing Your Software Investment*. Wiley Publishing, 1st edition, 1996. 8, 9

[38] Uttamjit Kaur and Gagandeep Singh. A review on software maintenance issues and how to reduce maintenance efforts. *International Journal of Computer Applications*, 118(1):6–11, 2015. 8

[39] Meir M Lehman. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9):1060–1076, 2005. 8

[40] Sasa Dekleva. Delphi study of software maintenance problems. In *Software Maintenance, 1992. Proceerdings., Conference on*, pages 10–17. IEEE, 1992. 9

[41] J Daniel Couger. Effect of cultural differences on motivation of analysts and programmers: Singapore vs. the united states. *MIS Quarterly*, pages 189–196, 1986. 9

[42] Bennet P Lientz. Issues in software maintenance. *ACM Computing Surveys (CSUR)*, 15(3):271–278, 1983. 9

[43] Paul J Layzell and Linda A Macaulay. An investigation into software maintenance—perception and practices. *Journal of Software Maintenance: research and practice*, 6(3):105–120, 1994. 9

[44] Hans Van Vliet, Hans Van Vliet, and JC Van Vliet. *Software engineering: principles and practice*, volume 13. John Wiley & Sons Hoboken, NJ, 2008. 9

[45] Priyadarshi Tripathy and Kshirasagar Naik. *Software evolution and maintenance: a practitioner's approach*. John Wiley & Sons, 2014. 9, 10

[46] Israel Herraiz, Daniel Rodríguez, Gregorio Robles, and Jesús M. González-Barahona. The evolution of the laws of software evolution: A discussion based on a systematic literature review. *ACM Comput. Surv.*, 46(2):28:1–28:28, 2013. 9, 10

[47] Meir M Lehman. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9):1060–1076, 1980. 10

[48] Maryam Razavian and Patricia Lago. A lean and mean strategy for migration to services. In *Proceedings of the WICSA/ECSA 2012 Companion Volume*, WICSA/ECSA '12, page 61–68, New York, NY, USA, 2012. Association for Computing Machinery. 10

[49] Hong Zhou, Jian Kang, Feng Chen, and Hongji Yang. Optima: An ontology-based platform-specific software migration approach. In *Seventh International Conference on Quality Software (QSIC 2007)*, pages 143–152, 2007. 10

[50] J. Martin and H.A. Muller. C to java migration experiences. In *Proceedings of the Sixth European Conference on Software Maintenance and Reengineering*, pages 143–153, 2002. 10

[51] Ying Zou and K. Kontogiannis. A framework for migrating procedural code to object-oriented platforms. In *Proceedings Eighth Asia-Pacific Software Engineering Conference*, pages 390–399, 2001. 11

[52] Eelco Visser. A survey of rewriting strategies in program transformation systems. *Electronic Notes in Theoretical Computer Science*, 57:109–143, 2001. 11

[53] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Signature Series (Fowler). Pearson Education, 2018. 11, 12, 17

[54] Axel Rauschmayer. *Exploring ES6: Upgrade to the Next Version of JavaScript*. Leanpub, 2015. 12, 13, 14

[55] Brett Slatkin. *Effective python: 90 specific ways to write better python*. Addison-Wesley Professional, 2019. 15

[56] Rashina Hoda. Socio-technical grounded theory for software engineering. *arXiv preprint arXiv:2103.14235*, 2021. 21

[57] Barney G Glaser and Anselm Strauss. L.(1967). the discovery of grounded theory: Strategies for qualitative research. *Chi cago: Aldine*, 1967. 21

[58] Anselm Strauss and Juliet Corbin. *Basics of qualitative research*. Sage publications, 1990. 21

[59] Kathy Charmaz. *Constructing grounded theory: A practical guide through qualitative analysis*. sage, 2006. 21

[60] Kathy Charmaz. Grounded theory as an emergent method. In Sharlene Nagy Hesse-Biber and Patricia Leavy, editors, *Handbook of emergent methods*, pages 155–170. Guilford Press, New York, NY, 2008. 21

[61] K. Charmaz. *Constructing Grounded Theory*. Introducing Qualitative Methods series. SAGE Publications, 2014. 21

[62] Aylton Almeida, Laerte Xavier, and Marco Tulio Valente. Using copilot agent mode to automate library migration: A quantitative assessment. *arXiv preprint arXiv:2510.26699*, 2025. 30, 122

[63] Matthias Kebede, May Mahmoud, Mohayeminul Islam, and Sarah Nadi. Migmate: A vs code extension for llm-based library migration of python projects. *arXiv preprint arXiv:2603.01596*, 2026. 30, 122

[64] Nishil Amin, Zhiwei Fei, Xiang Li, Justyna Petke, and He Ye. Jmigbench: A benchmark for evaluating llms on source code migration (java 8 to java 11). *arXiv preprint arXiv:2602.09930*, 2026. 30, 122

[65] Victor May, Diganta Misra, Yanqi Luo, Anjali Sridhar, Justine Gehring, and Silvio Soares Ribeiro Junior. Freshbrew: A benchmark for evaluating ai agents on java code migration. *arXiv preprint arXiv:2510.04852*, 2025. 30, 121

[66] Zirui Chen, Zhipeng Xue, Jiayuan Zhou, Xing Hu, Xin Xia, and Xiaohu Yang. Diffploit: Facilitating cross-version exploit migration for open source library vulnerabilities. *arXiv preprint arXiv:2511.12950*, 2025. 30, 121

[67] Chris Parnin, Christian Bird, and Emerson R. Murphy-Hill. Adoption and use of java generics. *Empir. Softw. Eng.*, 18(6):1047–1089, 2013. 32

[68] Robert Dyer, Hridesh Rajan, Hoan Anh Nguyen, and Tien N. Nguyen. Mining billions of AST nodes to study actual and potential usage of java language features. In Pankaj Jalote, Lionel C. Briand, and André van der Hoek, editors, *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*, pages 779–790. ACM, 2014. 32

[69] Rashina Hoda, James Noble, and Stuart Marshall. The impact of inadequate customer collaboration on self-organizing agile teams. *Inf. Softw. Technol.*, 53(5):521–534, 2011. 36

[70] Hashini Gunatilake, John C. Grundy, Rashina Hoda, and Ingo Mueller. The role of empathy in software engineering - A socio-technical grounded theory. *CoRR*, abs/2504.13002, 2025. 36, 99

[71] Tim Klinger, Peri L. Tarr, Patrick Wagstrom, and Clay Williams. An enterprise perspective on technical debt. In Ipek Ozkaya, Philippe Kruchten, Robert L. Nord, and Nanette Brown, editors, *Proceedings of the 2nd Workshop on Managing Technical Debt, MTD 2011, Waikiki, Honolulu, HI, USA, May 23, 2011*, pages 35–38. ACM, 2011. 81

[72] Yuepu Guo, Rodrigo Oliveira Spínola, and Carolyn B. Seaman. Exploring the costs of technical debt management - a case study. *Empir. Softw. Eng.*, 21(1):159–182, 2016. 81

[73] Ulrike Maria Graetsch, Hourieh Khalajzadeh, Mojtaba Shahin, Rashina Hoda, and John C. Grundy. Dealing with data challenges when delivering data-intensive software solutions. *IEEE Trans. Software Eng.*, 49(9):4349–4370, 2023. 99

[74] Satyajit Gokhale, Alexi Turcotte, and Frank Tip. Automatic migration from synchronous to asynchronous javascript apis. *Proc. ACM Program. Lang.*, 5(OOPSLA), oct 2021. 120

[75] Zejun Zhang, Zhenchang Xing, Xin Xia, Xiwei Xu, and Liming Zhu. Making python code idiomatic by automatic refactoring non-idiomatic python code with pythonic idioms. In Abhik Roychoudhury, Cristian Cadar, and Miryung Kim, editors, *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022*, pages 696–708. ACM, 2022. 120

[76] Balázs Rózsa, Gábor Antal, and Rudolf Ferenc. Don't DIY: automatically transform legacy python code to support structural pattern matching. In *22nd IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2021, Limassol, Cyprus, October 3, 2022*, pages 164–169. IEEE, 2022. 121

[77] Negar Alizadeh, Boris Belchev, Nishant Saurabh, Patricia Kelbert, and Fernando Castor. Analyzing the energy and accuracy of llms in software development. *CoRR*, abs/2412.00329, 2024. 121

[78] Julian Oertel and Regina Hebig. The impact of generative AI on developer practices, behavior, and software quality. In *IEEE International Conference on Software Maintenance and Evolution, ICSME 2025, Auckland, New Zealand, September 7-12, 2025*, pages 878–880. IEEE, 2025. 121

[79] Valtteri Ala-Salmi, Zeeshan Rasheed, Abdul Malik Sami, Muhammad Waseem, Kai-Kristian Kemell, Jussi Rasku, Mika Saari, and Pekka Abrahamsson. Vapu: System for autonomous legacy code modernization. *arXiv preprint arXiv:2510.18509*, 2025. 121

[80] Robert Dyer, Hridesh Rajan, Hoan Anh Nguyen, and Tien N. Nguyen. Mining billions of ast nodes to study actual and potential usage of java language features. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, page 779–790, New York, NY, USA, 2014. Association for Computing Machinery. 124

[81] Davood Mazinanian, Ameya Ketkar, Nikolaos Tsantalis, and Danny Dig. Understanding the use of lambda expressions in java. *Proc. ACM Program. Lang.*, 1(OOPSLA):85:1–85:31, 2017. 124

[82] Norbert Vándor, Gábor Antal, Péter Hegedüs, and Rudolf Ferenc. On the usefulness of python structural pattern matching: An empirical study. In *IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2024, Rovaniemi, Finland, March 12-15, 2024*, pages 501–511. IEEE, 2024. 125

[83] Fernando Alves, Delano Oliveira, Fernanda Madeiral, and Fernando Castor. On the bug-proneness of structures inspired by functional programming in javascript projects, 2022. 125

[84] Thiago Nicolini, André C. Hora, and Eduardo Figueiredo. On the usage of new javascript features through transpilers: The babel case. *IEEE Softw.*, 41(1):105–112, 2024. 126

[85] Phillip Merlin Uesbeck, Andreas Stefik, Stefan Hanenberg, Jan Pedersen, and Patrick Daleiden. An empirical study on the impact of C++ lambdas and programmer experience. In Laura K. Dillon, Willem Visser, and Laurie A. Williams, editors, *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, pages 760–771. ACM, 2016. 126

[86] Lin Chen, Di Wu, Wanwangying Ma, Yuming Zhou, Baowen Xu, and Hareton Leung. How C++ templates are used for generic programming: An empirical study on 50 open source systems. *ACM Trans. Softw. Eng. Methodol.*, 29(1):3:1–3:49, 2020. 127

[87] Ravi Khadka, Belfrit V. Batlajery, Amir Saeidi, Slinger Jansen, and Jurriaan Hage. How do professionals perceive legacy systems and software modernization? In Pankaj Jalote, Lionel C. Briand, and André van der Hoek, editors, *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*, pages 36–47. ACM, 2014. 128

[88] Margaret R Roller and Paul J Lavrakas. *Applied qualitative research design: A total quality framework approach*. Guilford Publications, 2015. 131

[89] Nahid Golafshani et al. Understanding reliability and validity in qualitative research. *The qualitative report*, 8(4):597–607, 2003. 131

Appendix A

Appendix A — C++ Study

Received: 14 October 2022 | Revised: 25 April 2023 | Accepted: 18 July 2023

DOI: 10.1002/smr.2605

RESEARCH ARTICLE - EMPIRICAL

Journal of Software: Evolution and Process WILEY

Embracing modern C++ features: An empirical assessment on the KDE community

Walter Lucas[1] | Fausto Carvalho[1] | Rafael Campos Nunes[1] | Rodrigo Bonifácio[1] | João Saraiva[2] | Paola Accioly[3]

[1]Computer Science Department, University of Brasília, Brasília, Brazil

[2]Universidade do Minho, Braga, Portugal

[3]Federal University of Cariri, Juazeiro do Norte, Brazil

Correspondence
Walter Lucas, Computer Science Department, University of Brasília, Brasília, Brazil.
Email: walter.mendonca@aluno.unb.br

Funding information
FAP-DF; CAPES, Grant/Award Number: 07/2019; Foundation for Science and Technology (FCT), Grant/Award Number: LA/P/0063/2020

Abstract

Similar to software systems, programming languages evolve substantially over time. Indeed, the community has more recently seen the release of new versions of mainstream languages in shorter and shorter time frames. For instance, the C++ working group has begun to release a new version of the language every 3 years, which now has a greater number of modern C++ features and improvements in modern standards (C++11, C++14, C++17, and C++ 20). Nonetheless, there is little empirical evidence on how developers are transitioning to use modern C++ constructs in legacy systems, and not understanding the trends and reasons for adopting these new modern C++ features might hinder software developers in conducting rejuvenation efforts. In this paper, we conduct an in-depth study to understand the development practices of KDE contributors to evolve their projects toward the use of modern C++ features. Our results show a trend in the widespread adoption of some modern C++ features (*lambda expressions*, *auto-typed variables*, and *range-based for*) in KDE community projects. We also found that developers in the KDE community are making large efforts to modernize their programs using automated tools, and we present some modernization scenarios and the benefits of adopting modern C++ features of the C++ programming language. Our results might help C++ software developers, in general, to evolve C++ legacy systems and tools builders to implement more effective tools that could help in rejuvenation efforts.

KEYWORDS
C++ programming language, language evolution, software rejuvenation

1 | INTRODUCTION

The C++ programming language was designed at the beginning of the 1980s with the main purpose of extending the C language with object-oriented constructs. The first C++ standard was released in 1998 (C++98), with contributions from a language committee* formed in 1990. The language has adhered to the C++11 standard in 2011 and is currently receiving updates more frequently (an estimated time of 3 years difference between releases of the new standards). These updates, in turn, made the language more expressive and secure, adopting certain idioms or completely new constructs such as type inference, lambda expressions, resource transfer, smart pointers, and concurrent support within the standard library.[1]

*Committee ISO/IEC JTC1 (Joint Technical Committee 1)/SC22 (Subcommittee 22)/WG21 (Working Group 21).

kde-devel mailing list (May, 2022)
Hello, KDE team,
I would like to create the independent fork of Krita that would allow to use all modern features of C++, up till that is supported by the latest GCC release, 12.1 at the time of writing.
The current Krita development rules are limited by C++11 that is now the ten years old standard. Even that is limited by the lengthy list of restrictions. The rules also require using deprecated features of the Qt framework like Q_FOREACH. I understand the care of the community not to spoil anything and to preserve the beauty of the existing code. This means radical changes should be done in a form.

OpenJDK JEP 347: Enable C++14 Language Features (July, 2018)
Summary:

- Allow the use of C++14 language features in JDK C++ source code, and give specific guidance about which of those features may be used in HotSpot code.

[…]

Risk and Assumptions:

- There may be other platforms with toolchains that do not yet support the C++14 language standard.
- There may be bugs in the support for some new features by some compilers.

FIGURE 1 Open-source forum messages discussing the need for updating the C++ version.

These language modifications change the way developers write their programs. Nonetheless, keeping legacy systems up to date with new versions of a language is a common concern related to language evolution.[2] On the one hand, developers wish to use modern C++ features in their programs, to improve some quality concerns (e.g., software readability, maintenance, and security). On the other hand, migrating a code base to support new versions of a language is everything but trivial. Developers have to conciliate dependencies with external libraries and also consider the implications of changing the configuration or versions of compilers to the users of their programs or libraries.[3] Indeed, we have observed in the developer's mailing lists of open-source communities some extensive, even heated discussions about whether or not to conduct a maintenance effort on a program or library toward language evolution. For instance, Figure 1 presents two snippets of message threads from the open-source community, revealing not only the interest in updating the code to support C++11 but also some challenges that might hinder software developers from modernizing legacy code bases.

Given the importance of C++ and the challenges related to evolving legacy systems to adopt new versions of programming languages, it is worth characterizing how software developers embrace the modern features of C++ and which practices developers use during rejuvenation efforts. Indeed, previous work[4–6] reports efforts to *rejuvenate programs*, a particular kind of software maintenance whose goal is to replace legacy code features and idioms with modern constructs available in recent versions of a programming language.[7] Efforts to rejuvenate software systems have also been investigated and reported in the literature.

For instance, Mazinanian et al[5] report a large-scale study to understand how professional developers use lambda expressions in Java programs, after its adoption on Java 8. The study evaluated 100,540 lambda expressions in 241 open-source projects, revealing an increasing trend in lambda expression adoption. Similarly, Alqaimi et al[4] investigate the use of lambda expressions in Java programs on GitHub and report that 11% of the repositories use lambda expressions and that only 6% of lambda expressions are accompanied by source code comments—which might suggest that lambda do not make the programs harder to understand. The authors also present an automated tool to generate complete documentation for lambda expressions in Java. Kumar et al[6] present a tool for rejuvenating C++ code that implements source code transformations of C++ macros into C++11 feature usages. They report that 68% to 98% of macros can be translated into modern C++11 features. In addition, the authors also present techniques to assist developers to automate the rejuvenation process.

Despite all these research efforts to discuss modern C++ features adoption, there is still a lack of literature about the practices C++ developers use to conduct rejuvenation efforts. Our research aims to fill this gap by conducting a large-scale empirical study where we analyze 272 open-source projects from the KDE community,[†] in order to understand the trends in the adoption of modern C++ features and the practices developers use while modernizing C++ code. Our findings characterize maintenance efforts in the KDE community to introduce modern C++ features released in the recent versions of C++: C++11, C++14, and C++17. We discuss the trends in the adoption of modern C++ features such as *lambda expressions*, *auto-typed variables*, and *range-based for*, and our results bring evidence of the use of automated tools for C++ code

[†]KDE is an international cross-platform project community designed for Linux systems.

modernization. In addition, we present a catalog of commits (Table A.1) of rejuvenation efforts conducted by KDE developers that could help other developers rejuvenate their programs and tool builders identify opportunities to implement source code transformations.

This paper is organized as follows: Section 2.1 presents some background information about the evolution of C++ language. Section 2.2 compares our study with previous work about software rejuvenation in the literature. Section 3 presents our research questions and the procedures that we follow during our investigation. Section 4 shows the results of our quantitative assessments and its implications. Section 5 presents some examples of rejuvenation efforts and summarizes the perspective of KDE developers rejuvenation efforts in C++ project. In Section 6, we summarize the implications of our study and the main threats to the validity of our work. Finally, Section 7 presents our final considerations.

1.1 | Artifacts availability

Our study is fully reproducible. All developed tools, collected data, and R scripts are available online for statistical analysis in a Git repository: https://github.com/PAMunb/cppEvolution.

2 | BACKGROUND AND RELATED WORK

Some programming languages are constantly evolving and follow a rigorous and organized evolution process. The C++ community, for instance, is responsible for steering the language's evolution in accordance with the recommendations of the International C++ Standard Committee. The C++ community is composed of organizations such as Apple, Facebook, Google, IBM, Intel, Microsoft, and Nvidia, as well as interested parties from multiple nations. In addition, universities and hardware vendors support the C++ committee, along with representatives from diverse fields such as finance, game development, C++ library communities (e.g., Boost), and platform vendors. The community organizes meetings into working groups (WGs) and study groups (SGs).[1] It is the responsibility of committee members to submit proposals, which may include modifications to the language or standard libraries. Additionally, the committee functions as a filter to prevent bad proposals from entering the standard.[1] The majority of the committee's efforts are devoted to addressing simple issues such as naming conventions, grammatical details, the addition of new constructs, and backwards compatibility with previous versions of C++. In the following Section 2.1, we provide an overview of the C++ evolution process and its modifications. Next, in Section 2.2, we discuss related works and how our own compares to them.

2.1 | C++ Language evolution

The C++ programming language is a multi-paradigm and general-purpose language, whose first official version, known informally as C++98, has been released as an ISO/IEC standard (ISO/IEC 14882:1998).[8] The C++98 standard includes numerous language features such as templates, exceptions, dynamic_cast, namespaces, declarations in conditions, named casts, and bool_type. C++ also includes the Standard Template Library (STL),[9] which provides features such as the general and efficient framework of containers, iterators, and algorithms. In addition, other resources such as traits, string, bitset, locales, auto_ptr, and shared_ptr were added to the standard language library.[10]

In 2003, the WG21 committee issued a new version, which became known informally as C++03. This version of the language was primarily a bug fix release that included several library revisions (C++ Standard Library Issues List TC1[‡]). In 2006, the committee issued a new version known as C++0x. The C++0x version was to contain a large number of features that were frozen and not released until the next version in 2011.[1] Starting in 2011, the International C++ Committee agreed to publish a new C++ release every 3 years.

In 2011, the committee released the standardized version known as C++11, which brought significant changes to the C++[1] programming language and paved the way for what is now called *modern C++*.[11] C++11 introduced various modern language features such as auto and decltype, range-for, nullptr, constexpr functions, user-defined literals, lambda expressions (unnamed function objects), variadic templates, and the noexcept keyword, among many others.[12] The C++11 version also included several built-in modern C++ features from the Boost Library,[13] which had a significant impact on the new release of the C++[14] Standard. Some of these modern C++ features are the thread library, exception_ptr, error_code, error_condition, and iterator improvements.

The C++14 standard was issued by the WG21 committee as an extension of the C++11.[1] The changes consisted mainly of bug fixes that had been noticed during the initial use of the C++11 standard, and also included minor improvements to existing language features (two-range overloads for some algorithms, type alias versions of type traits, user-defined literals for basic_string, duration, and complex).[12] The C++14 version included language features such as variable templates, generic lambdas, lambda init capture, new/delete elision, and aggregate classes with

[‡]https://open-std.org/JTC1/SC22/WG21/docs/lwg-status.html#TC1.

default non-static member initializers.[12] Some library resources such as std::make_unique, std::shared_timed_mutex, std::shared_lock, std::integer_sequence, std::exchange, and std::quoted have also been included.

The C++17 standard has not undergone major changes like the previous standards C++11 and C++14, but it includes modifications and features to improve the previous standards. C++17 included changes to remove language and library features that were deprecated during the C++ standard evolution.[11] C++17 also added modern language features like u8 character literal, made noexcept part of the type system, new order of evaluation rules, and lambda capture of `*this`. Improvements in the library were also part of this standard, such as file system, scoped_lock, shared_mutex (reader-writer locks), any, variant, optional, string_view, parallel algorithms, and others.[11,15]

C++20 introduced a programming concept that differs from previous standards, with language features aimed at concurrency, generic lambda expressions, metaprogramming, stricter type safety, and others.[16] The new standard has language features such as feature test macros, the three-way comparison operator `<=>` and the operator `==()=` default, designated initializers, init statements, and initializers in range-for. C++20 also included new library features such as concepts, ranges, dates and time zones, span, formats, improved concurrency, parallelism support, and others.[1]

2.2 | Research on software rejuvenation

Software maintenance can be motivated by various factors, such as business adaptations, changing requirements, architectural transitions (e.g., the use of cloud environments), the need for quality improvements (e.g., refactoring), and also technical aspects, such as program evolution scenarios that are driven by the evolution of the programming languages used.[17] Software rejuvenation is a particular type of software maintenance whose goal is to replace obsolete features with modern programming language features that can coexist with evolved software.[2] Research works that investigate the evolution of software through rejuvenation is increasingly common in software engineering.[5,6,18,19]

Regarding previous studies proposing tools to automate software rejuvenation, Kumar et al[6] presented a tool for rejuvenating C++11 code. The methodology proposed in the paper involves three main steps: identifying preprocessor macros in the codebase using a combination of lexical and syntactic analysis, transforming the macros into equivalent C++ code using a set of predefined rules, and analyzing the transformed code to detect any issues that the macro transformation process may have introduced. They reported that modern C++11 features can replace 68% to 98% of macros. Their results suggest that it is possible to replace a large part of legacy code with modern C++ features, but some transformations require manual adjustments. Dantas et al[18] presented a library of Java transformations developed in the Rascal metaprogramming language to rejuvenate legacy systems to support newer programming language constructs. The authors ran a total of 2462 source code transformations in 40 open-source projects. A small sample of these transformations was submitted to projects via GitHub's pull-requests mechanism. The study identified that simple transformations, such as adopting *diamond operators*, are more likely to be accepted than transformations that substantially change the code, such as replacing *for loops* by the new functional style. Unlike our study, these works[6,18] do not investigate what motivations drive developers to adopt modern language features or the benefits of adopting modern language features.

Mazinanian et al[5] conducted a large-scale study with a mixed-methods approach to understanding how developers use lambda expressions in Java programs. The authors evaluated 100,540 lambda expressions across 241 open-source projects to identify the usage patterns of lambda expressions. They also surveyed Java developers to collect information on their experience with using lambda expressions. The survey questions focused on developers' experiences with the language feature, how often they use it, and what benefits and limitations they perceive when using it. The study revealed an increasing trend in adopting lambda expressions in 2016 (the proportion of lambdas introduced per line of code doubled compared with 2015). In addition, they present that leading developers introduce more lambda expressions than external collaborators and that Java developers tend to write them manually. We also found a trend (Section 4) of adoption of some features (*auto-typed variables*, *lambda expressions*, and *range-based for*) since 2016 for C++ projects in the KDE community. In addition, our study shows evidence that the top developers were responsible for 63.2% of the rejuvenation commits found. Finally, the survey results indicate some benefits of using lambda expressions, like improved code readability and reduced code verbosity. However, developers also reported challenges when using lambda expressions, including debugging issues. Additionally, the findings suggest that the Java community should continue to improve tooling to support developers' use of lambda expressions.

Lucas et al[19] conducted an empirical study to investigate the impact on code comprehension after introducing *lambda expressions* in Java programs. The study evaluated 66 pairs of code transformations (one pair consists of a source code snippet in the version before the transformation and the same snippet after the introduction of *lambda expressions*). First, some metrics were extracted for each pair of transformations: two metrics related to source code complexity (number of lines of code and cyclomatic complexity[20]) and two metrics that estimate source code readability.[21,22] Then, the authors surveyed professionals to collect their perceptions about the benefits of understanding the program by adopting *lambda expressions*. Such results presented a contradiction. Based on the quantitative assessment, the researchers found no evidence that the introduction of *lambda expressions* leads to improvements in the readability of software, which differs from the qualitative evaluation that suggests improvements in program comprehension. Additionally, the study revealed that the use of *lambda expressions* in some scenarios can improve code comprehension (like the replacement of anonymous inner classes by *lambda expressions*). Differently, just replacing a simple `for statement`

over a collection statement by a `collections.forEach()` does not bring any benefits, according to the participants of the survey. Our work differs from those studies[5,19] in two aspects: (a) We study the adoption of modern features introduced by the evolution of the C++ language introduced between three versions (C++11, C++14, and C++17); and (b) our study is not limited only to features that introduce the functional style (such as lambda expressions).

Contrary to Mazananian et al. and Lucas et al., Uesbeck et al[23] and Zheng et al[24] conducted studies showing harmful effects of adopting *lambda expressions* in C++ and Java projects, respectively. Uesbeck et al. conducted controlled experiments to understand the impact on developers' productivity while implementing different tasks with and without *lambda expressions*. Their results show that using *lambda expressions* negatively affects productivity for less experienced developers regarding how quickly they can write correct programs. However, it had no positive or negative effect on more experienced developers' productivity. The results suggest that learning how to use *lambda expressions* properly might take some time. Moreover, Zheng et al. show data confirming a trend of developers removing *lambda expressions* from their code in Java starting in 2016. They conduct a study to understand the reasons behind such phenomena. As a result, the authors describe developers often misusing *lambda expressions*, causing unwanted side effects, such as memory leakage or inducing new bugs. They provide seven main reasons for removing *lambda expressions* and seven common migration patterns. Such results are beneficial for developers that are still learning how to use lambda expressions in Java. These previous studies[23,24] complement our results because we also show an increasing use trend in lambda expressions since 2016 and the factors that might explain it, but we do not analyze the harmful effects of this trend.

Finally, Chen et al[25] conduct a study to understand how developers have used C++ templates since their first release. They find out that the main reason for using C++ templates is to avoid code duplication. Moreover, they also measure the proportion of explicit type declarations versus implicit type declarations (the *auto-typed variables*), concluding that developers prefer to keep explicit type declarations instead of changing to the auto-typed variables. This result complements our study because, while we describe many examples where developers chose to rejuvenate code by using *auto*-typed variables, we do not measure how much of the code still uses explicit type declarations.

3 | STUDY SETTINGS

This study aims to gain a comprehensive knowledge of how C++ developers utilize *modern* C++ features included in the C++11, C++14, and C++17 standards. This study focuses on a few of C++11 and C++14 innovations, including *lambda expressions*, type inference algorithms based on *auto-typed variables*, and the brand-new *range-based for* statement. According to Stroustrup, these features would be subject to more adoption expectations.[1] In addition, we incorporate the C++17 *if-statement with initializer* and the new C++ support for concurrent programming in our research.

Also related to the scope of our research, we focus on a relevant organization of C++ developers: the KDE open community, an international free software community responsible for developing hundreds of applications targeting different domains, such as Games, Education, and Frameworks, including the Plasma desktop that runs in Linux distributions (e.g., OpenSuse) and mobile phones. Similar to other open-source organizations, KDE makes available development policies[§] and leverages static analysis procedures for conformance checking (including the Krazy tool[¶]).

We started analyzing 1061 C++ repositories that we checked out from the GitHub KDE community (the official read-only mirror of the KDE projects). We removed 11 repositories that do not contain C++ files. From the remaining 1050 repositories, we filtered out projects with a small percentage of C++ code (below 50%) and projects that started after 2010 or that did not have recent updates—that is, we only consider projects that have at least one commit in 2022. Our curated dataset contains 272 KDE programs and libraries written in C++.

3.1 | Research questions

We investigate the following research questions:

RQ1. To what extent do KDE systems rely on modern C++ features?
RQ2. When did KDE developers start using modern C++ features?
RQ3. Is there any trend in the adoption of modern C++ features in KDE applications?
RQ4. Do KDE developers conduct maintenance efforts having the sole goal of rejuvenating C++ code?
RQ5. Which tools do KDE developers use to support maintenance efforts for code rejuvenation?
RQ6. What are the reasons that motivate KDE developers to conduct maintenance efforts for code rejuvenation?
RQ7. Are the core developers of the projects responsible for conducting rejuvenation efforts in KDE projects?

[§] https://community.kde.org/Policies.
[¶] https://github.com/Krazy-collection/krazy.

Answering the first research question allows us to understand how popular modern C++ features such as *lambda expressions*, *auto-typed variables*, *range-based for*, and *if-with-initializer statements* are in the KDE applications' code base. Answering the second research question enables us to know how long it takes for fully fledged KDE projects to start migration efforts of the codebase to use modern C++ features. Answering the third research question allows us to understand if there is a trend toward rejuvenating the code of KDE applications. Answering the fourth and fifth research questions allows us to identify whether or not KDE developers conduct rejuvenation efforts, and, if so, what tools they use. Answering the sixth research question allow us to identify what motivations lead KDE developers to conduct rejuvenation efforts in their programs. Finally, answering the last research question enables us to identify which developers conduct rejuvenation efforts and whether they are the main developers of KDE projects.

3.2 | Research procedures

To answer our research questions, we mined the source code repository of 272 KDE applications and collected facts about the usage of modern C++ features using a static analysis tool we implemented using Java and the Eclipse CDT infrastructure for parsing and traversing C++ code. For the research questions RQ5 and RQ6, we also surveyed KDE developers via e-mail messages.

In more detail, to answer the first research question, we first run our static analysis infrastructure on the latest revision of the applications. We then (i) calculate the absolute number of occurrences of the modern C++ features we are interested in (e.g., *lambda expressions*, *auto-typed variables*, *range-based for*, and *if-with-initializer statements*) and (ii) carry out descriptive statistics on that data. To answer the second and third research questions, we collect the same statistics considering a weekly based interval of all revisions of the applications' code base, starting from January 1, 2010, until June 1, 2022. Our primary motivation for establishing this range is due to the considerable number of commits that some projects have, which can lead to a time-consuming analysis. As a means to mitigate this issue, we defined the range to ensure the feasibility of our study. Algorithm 1 clarifies this approach. We use Time Series to investigate the trend in the adoption of modern C++ features.

Algorithm 1 Reverse walking of the code history (RWCH).

Require: A valid source code Git repository (*repository*)
Ensure: a set (*observations*) whose entries contain the metrics for a given revision in the source code

```
previous_date = null
observations = {}
for Revision r ∈ repository.history() do
    current_date = r.date
    if previous_date == null or diffInDays(current_date, previous_date) >= 7 then
        repository.checkout(r)
        metrics = traverse(repository)
        observations = observations ∪ (r, metrics)
        previous_date = current_date
    end if
end for
```

We use a conservative heuristic to search for commits that characterize rejuvenation efforts in KDE projects and then answer the fourth and fifth research questions. Our heuristic iterates over consecutive commits in our dataset (i.e., with a distance of at least seven days) and computes the increase in the adoption of each modern C++ feature. Whenever we find an increase of at least 50% in the adoption of modern C++ features and the amount of statements increases by less than 5%, we assume that a rejuvenation effort might have taken place in that period.

Using this conservative heuristic, we automatically identified 207 intervals that could contain a rejuvenation effort. We then retrieve all commits within the period of each interval and again perform a search for patches that might contain a rejuvenation commit. Using our approach, we then iterate over the commits several times, reducing the number of commits that must be manually evaluated. Within all intervals, we found the 207 that contain between 1000 and 24,000 commits. Because this approach would demand a huge amount of time to iterate over large commits to collect the metrics, we use a keyword search in the commit messages in cases where an interval contains more than 100 commits. The keywords we consider in the search include `lambda`, `auto`, `range-based`, or `modern for` combined with `modern`, `modernize`, `port away`, `migrate`, `migration`, `c++11`, `c++14`, and `c++17`. We select these keywords from commits we found during a manual search of commit messages. Otherwise, if the interval contains fewer than 100 commits, we use the iterative approach to identify smaller intervals that might contain a rejuvenation effort. Our heuristic produces two independent text files, containing the results of the commit message keyword-based search and

the results of the iterative search. We manually analyze the results to validate whether or not a given commit corresponds to a rejuvenation effort.

Subsequently, we mine the authors of the commits that we validate as *rejuvenation efforts*. To answer the sixth research question, we contacted these developers via email to understand their motivations for carrying out maintenance efforts aimed at rejuvenating the projects' source code. The developers we contacted are the same contributors who conducted the large rejuvenation efforts identified in earlier stages of our research. Finally, to answer the last research question, we used the notion of Truck Factor (TF) to verify if these contributors are the core developers of the projects in which they conducted rejuvenation efforts. The notion of TF aims to identify a minimum set of developers that, if they abandon the project, the maintenance and evolution of that project might be discontinued. Furthermore, TF is suitable for assessing the distribution of knowledge among the developers of a project as shown by Ricca et al.[26] As reported by Bosu and Sultana,[27] the notion of TF can also be used to identify developers who have made significant contributions to guide the development and evolution of projects. There are several approaches to calculating TF in the literature, but here, we used the approach proposed by Avelino et al[28] for the following reasons: (i) The approach proved to be superior to other strategies,[29] and (ii) there is an easy-to-use implementation available, which (iii) has been used in another study to identify core developers.[30]

In Section 4, we present the results of a descriptive statistic analysis of our dataset, considering all projects. In this way, we give high-level answers to our research questions. Next, in Section 5.2, we present more detailed qualitative information regarding the KDE developer's adoption of modern C++ features.

4 | RESULTS OF THE QUANTITATIVE ASSESSMENT

In this section, we present the findings of a descriptive analysis that we conducted on our dataset, which includes all projects. We also address the research questions RQ1, RQ2, and RQ3. In the following section, we will discuss qualitatively the modern C++ features used by KDE developers. Table 1 shows the general adoption of our modern C++ features of interest in KDE projects, as well as when a feature first appeared in our dataset. Notably, *auto-typed variables*, *lambda expressions*, and *range-based for* are present in over sixty percent of the KDE projects in our dataset. This confirms Bjarne Stroustrup's prediction that certain modern C++ features would be extensively adopted.[1] Moreover, a few months before the formal release of C++11, *auto-typed variables*, *lambda expressions*, and *decltype specifier* began to appear. Although C++11 added C++ multithreading capability, it is hardly used in our dataset. Conversely, we did not find evidence of the use for four other modern C++ features (*async function*, *future declarations*, *promise declarations*, and *shared future declarations*) that were in our initial subset. Therefore, we have focused our research on these three modern C++ features (*auto-typed variables*, *lambda expressions*, and *range-based for*).

The modern C++ features *lambda expressions*, *auto-typed variables*, and *range-based for* are widely used across the projects, even though their adoption distribution is not uniform (see Figure 2). The median distribution of *lambda expressions*, *auto-typed variables*, and *range-based for* is 2, 9, and 8.5, respectively. However, the KDevelop project alone contains 4441 occurrences of *auto-typed variables* (13.36% of the total). We observe the same lack of uniformity for *lambda expressions* and *range-based for*. We run a test to estimate the Spearman correlation[31,32] between the modern features adoption and the size of the projects (in terms of the total number of statements and the total number of C++ files). First, we found just a moderate correlation ($cor \leq 0.62$) between the adoption of language features and the size of the projects, indicating that we cannot explain a large use of language features using only project size metrics. Next, we run a test to estimate a correlation between the usage of modern C++ features (*lambda expressions*, *auto-typed variables*, and *range-based for*). We found a positive and strong correlation ($cor \geq 0.71$) between *lambda expressions*, *auto-typed variables*, and *range-based for* usage and a very strong correlation ($cor \geq 0.85$) between *auto-typed variables* and *range-based for*. Although LLVM introduced the support for modern C++ features before 2011 (e.g., LLVM version of 2008 already supports *lambda expressions*[33]), our results give evidence that, within the KDE projects in our dataset, the widespread adoption of modern C++ feature took place years after the release of the C++11 standard.[34]

TABLE 1 Adoption of modern C++ features in KDE projects.

Feature	Projects adoption (%)	Occurrences (#)	First occurrence
auto-typed variables	80.51	33,225	March 2010
constant expressions	14.70	777	November 2012
if-with-initializer statements	4.4	71	May 2020
lambda expressions	63.60	8918	March 2010
range-based for	78.30	16,485	December 2011
thread declaration	0.73	6	May 2018
decltype specifier	2.2	27	April 2010

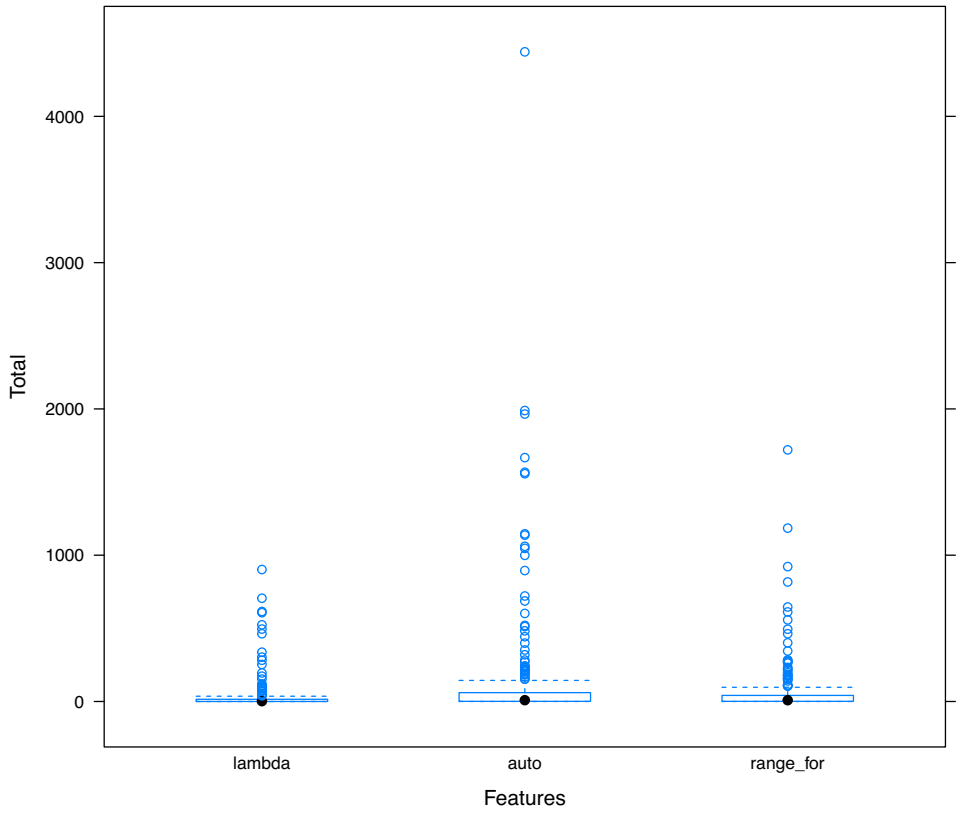


FIGURE 2 Distribution of modern C++ features usage across the KDE projects in our dataset. Each of the points on this graph represents the respective amount of that modern feature (*auto-typed variables*, *lambda expressions*, and *range-based for*) usage per project.

***RQ1*: To what extent do KDE systems rely on modern C++ features?** Our findings suggest that KDE developers extensively use *auto-typed variables*, *lambda expressions*, and *range-based for*—more than 80% of the projects use *auto-typed variables* in our dataset, while more than 60% of the projects use *lambda expressions* and 78% of the projects use *range-based for*. Other modern C++ features, such as *constant expressions*, *decltype specifier*, *if-with-initializer statements*, and *thread declaration* are rarely used in KDE projects.

Although KDE developers started using *auto-typed variables*, *lambda expressions*, and *range-based for* before 2012, we observed a more significant growth in their usage after 2016 (5 years after the release of the C++11 specification); see Figure 3. After this moment, we found rejuvenation efforts that changed many files to introduce these language features. We present more detailed information about these efforts in the next section.

***RQ2*: When did KDE developers start using modern C++ features?** Our findings suggest that the widespread adoption of modern features happened 5 years after the release of the C++11 specification—for those modern features frequently used. Accordingly, our results suggest that the general and widespread adoption of new language is not immediate—even for long waited features such as C++ *lambda expressions*.

Besides an observable trend in adopting new C++ features, Figure 3 presents some interesting situations, in which the total number of a given language feature usage declines. We manually investigated this issue and found two main reasons for that (a) there are specific commits that revert contributions that aim to rejuvenate the code base and (b) commits that merge branches often lead to this disruptive language feature

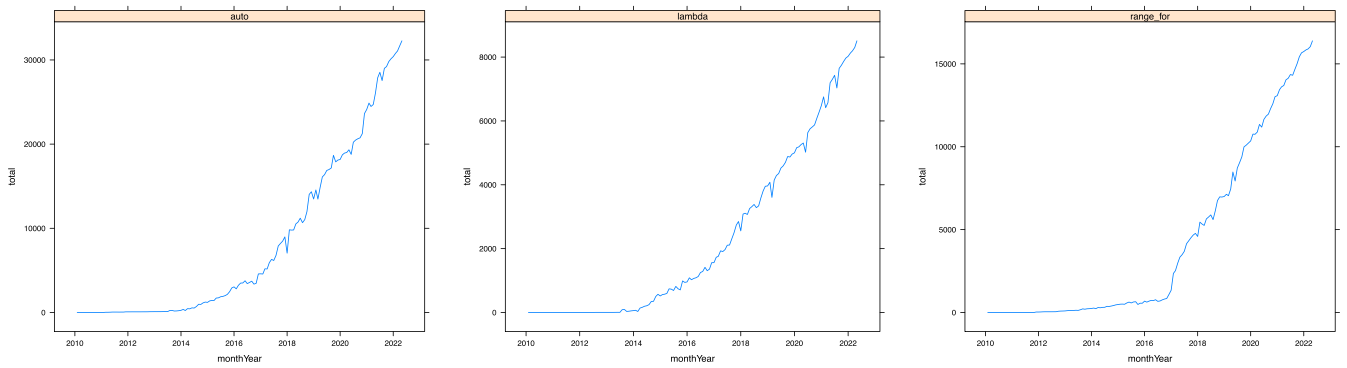


FIGURE 3 Distribution of *auto-typed variables*, *lambda expressions*, and *range-based for* usage across the KDE projects in our dataset.

adoption pattern, which contains contributions that reduce the number of feature adoption followed by contributions that rejuvenate the code again.

For instance, commit 1a5e7ab4 of `konversation` (commit date on October 1, 2020, and message *Port away from Qt's* `foreach`*: loops over method-local containers*) converts 110 *foreach* statements into the modern *range-based for* statement. This commit changed a total of 41 files.[#] Subsequently, commit ef184b7d reverts the changes in the aforementioned commit. The reversion goal is explicit in the commit message: "Revert Port away from Qt's `foreach`: loops over method-local containers." The author of this commit further details his/her motivation: "This patch reverts commit 1a5e7ab, which was accidentally … pushed as squashed commit of multiple commits which should be separate." For this particular case, the motivation to revert the commit was a procedure error when integrating and pushing a set of commits. We also observed the exclusion of many modern feature usages after merge commits.

We model our dataset using *time series* and conduct a regression analysis study using the `tslm` function available in the `Forecast` `R` `package`. To this end, we consolidate a monthly snapshot of the total usage of the modern C++ features *auto-typed variables*, *range-based for*, and *lambda expressions*. The `tslm` function implements a conventional regression analysis, though it includes additional predictors such as *trending* and *seasoning*. Because our dataset does not present seasonal data, we just estimate the *trend* component. The regression analysis results suggest a statistically significant trending increase for the adoption of the three features in our study (*auto-typed variables*, *range-based for*, and *lambda expressions*), with trending values of 210.78, 112.48, and 57.19 for *auto-typed variables*, *range-based for*, and *lambda expressions*, respectively. This means that every month, there is a trend of including 210.78 new *auto-typed variables* declarations. Note that if we constraint the same analysis for the period from 2010 until 2016, the same analysis will lead to trend values of 40.02, 10.29, and 14.31 *auto-typed variables*, *range-based for*, and *lambda expressions*. This result suggests a reasonable increase in adopting these modern C++ language features.

> ***RQ3*: Is there any trend in the adoption of modern C++ features in KDE applications?** Our findings also suggest a consistent trend toward increasing adoption of modern C++ language features in KDE projects, as well as a leaning toward the migration of legacy code to adopt *auto-typed variables*, *lambda expressions*, and *range-based for*. Nonetheless, occasionally, it was possible to identify commits that regress some migration efforts. This might explain points in the change history of the systems where we observed a reduction in adopting new language features followed by new commits that revert the reduction.

We find this trend interesting, and we investigated the motivations that can lead to increased adoption of these modern C++ features (*auto-typed variables*, *lambda expressions*, and *range-based for*). We present quotes from the answers of KDE developers who participated in our survey and some results of our descriptive analysis that can explain this trend. We detailed it in Section 5.3. In the next section, we present some examples of rejuvenation efforts to introduce *auto-typed variables*, *lambda expressions*, and *range-based for* statements. In particular, we highlight the KDE developers' perspectives on this particular type of software maintenance.

[#]Qt is an application development framework widely used by KDE projects. The KDE Free Qt Foundation https://kde.org/community/whatiskde/kdefreeqtfoundation/ ensures the availability of the Qt tools for KDE software development.

5 | RESULTS OF THE QUALITATIVE ASSESSMENT

We use two strategies in our qualitative assessment. First, we mine the source code history searching for possible rejuvenation scenarios that we further validate, via a manual analysis, focusing on commits that introduce many occurrences of *auto-typed variables*, *lambda expressions*, and *range-based for*. Second, we contacted KDE developers that implemented these rejuvenation efforts in their programs to answer the following questions:

SQ1. *What are the main motivations for adopting modern C++ features, such as lambda expressions, range-for, and declarations using the auto operator?*
SQ2. *How often do you conduct software rejuvenation efforts? Do you use any tool to support this kind of software maintenance?*
SQ3. *Is there any direction from the KDE community about when features of a new version of the C++ language should be used more widely in KDE projects, or is this an individual or small team decision in each project?*
SQ4. *What are the main reasons for the adoption rate of modern C++ features having increased substantially between 2015 and 2016?*
SQ5. *We have found that some features for concurrent programming in modern C++ (like thread declaration, futures, and async) are rarely used within the KDE projects. Is there an explanation for this?*

The first strategy (mine the source code history) allowed us to answer RQ7. In the second strategy (survey KDE developers), we use the answers from question SQ1 of the questionnaire to address our research question RQ6, while question SQ2 allows us to tackle question RQ5. Furthermore, questions SQ1, SQ3, and SQ4 give us insights into the motivations behind the trends, allowing us to explore RQ3 in more detail. Finally, SQ5 helps us understand the motivations that lead KDE developers not to use some modern C++ features like async and concurrence. We surveyed KDE developers primarily through email correspondence and conducted one interview with a contributor through an online meeting to collect data. We consolidated the survey results and presented a summarized overview of the findings using quoted text.

5.1 | Rejuvenation efforts

In Sections 3, we detail our conservative approach for mining rejuvenation efforts: source code patches that increase by at least 50% the amount of a modern C++ language feature (such as *auto-typed variables*, *lambda expressions*, and *range-based for*) and do not increase the total number of project statements in more than 5%. This conservative approach might lead to both false positives and false negatives. The results of our conservative approach to identifying rejuvenation efforts reveal a total of 81 commit candidates, from which we were able to manually confirm that 57 of them (70.37%) correspond to true positives—that is, commits that update the code with the sole goal of rejuvenating a program. More interesting, part of large rejuvenation efforts (8.7%) were conducted with the support of automatic refactoring tools.

These 57 rejuvenation efforts we confirm relate to 43 distinct projects in our dataset (15.80% of 272 projects). Also, of the 57 rejuvenation commits found, seven commits correspond to the introduction of *lambda expressions*, 27 commits correspond to the introduction of *auto-typed variables*, and 27 commits correspond to the introduction of *range-based for*. We found commits that introduce more than one feature, though. For instance, commit ea924b146 of `plasma-framework` (commit date on May 7, 2015, and message *port libplasma away from sycoca as much as possible*) introduces *lambda expressions*, *auto-typed variables*, and *range-based for*. Other examples include:

- Commit 2e508ac of `kdenlive` (commit date on April 20, 2017, and message *Use modern* `for`) replaces more than 200 occurrences of the `foreach` Qt statement[‖] by the new *range-based for* statement in 59 files. Figure 4 shows one example of the changes in this particular commit. Similar to several other contributions, the sole purpose of this commit was to rejuvenate the source code to use modern C++ features.
- Commit 5c28a3e5 of `labplot` (commit date on December 3, 2017, and message *Replace foreach macros with range-based for loops in many places*) is another example of a large transformation that replaces more than 300 occurrences of the `foreach` macro by the modern *range-based for* statement. This particular commit changes a total of 36 files. Listing 2 in Figure 4 shows an example of transformation in this particular commit—which also introduces more than 250 new declarations using the *auto-typed variables* feature for type deduction.
- Commit f519ce07 of `kid3` (commit date on September 29, 2018, and message *Use auto where it improves the readability*) corresponds to another example of code rejuvenation, introducing more than 400 *auto-typed variables* in 88 files. This commit is also representative w.r.t the KDE developer's adoption of tools to support code rejuvenation efforts. In the full message, the author says *Refactored automatically using clang-tidy*. See an example in Listing 3 (Figure 5). Listing 4 in Figure 5 shows an example of transformation in this particular commit. Another commit 7e6ff7d7 of `akregator` (commit date on November 2, 2020, and message *Modernize code*) introducing more than 94 *auto-typed variables* in 41 files.

[‖]https://doc.qt.io/qt-6/foreach-keyword.html.

```
if (!pattern.isEmpty()) {
  QVariantList mapped;
- foreach (const QModelIndex &ix, m_model->getChildrenIndexes()) {
+ for (const QModelIndex &ix : m_model->getChildrenIndexes()) {
    mapped << m_proxyModel->mapFromSource(ix);
  }
  // ...
}
```

Listing 1 Example of introducing *range-based for* in a large commit from `kdenlive`

```
- foreach (PluginLoader *loader, m_pluginsWithErrors) {
+ for (auto* loader : m_pluginsWithErrors) {
      if (loader->fileName() == absolutePath) {
         m_pluginsWithErrors.removeAll(loader);
         delete loader;
         break;
      }
  }
```

FIGURE 4 Example of introducing *range-based for*.

```
while (listIndex < completions.size()) {
  const QByteArray& name = completions.at(listIndex++);
  if (name.left(textLen) == text) {
- char* r = reinterpret_cast<char*>(::malloc(name.length() + 1));
+ auto r = reinterpret_cast<char*>(::malloc(name.length() + 1));
   qstrcpy(r, name.constData());
   return r;
  }
}
```

Listing 3 Example of introducing *auto-typed variables* in a large commit from `kid3`

```
static DeleteSubscriptionJob *reallyCreateJob(TreeNode *node)
{
- DeleteSubscriptionJob *job = new DeleteSubscriptionJob;
+ auto *job = new DeleteSubscriptionJob;
   job->setSubscriptionId(node->id());
   return job;
}
```

Listing 4 Example of introducing *auto-typed variables* in a large commit from `akregator`

FIGURE 5 Example of introducing *auto-typed variables*

Although we found several commits that introduce many occurrence of *auto-typed variables* and *range-based for*, this kind of rejuvenation effort is more scarse for *lambda expressions*. Commit 40a2aee94 of `kbibtex` (commit date on July 20, 2019, and message *Refactoring usage of QSignalMapper by using lambda functions in QObject::connect calls*) introduces new 40 *lambda expressions* distributed in 16 files. Listing 5 in Figure 6 shows one example of the changes in this particular commit. Finally, commit 034e0b7b of `kwidgetsaddons` (commit date on May 7, 2021, and message *Modernise code base ... Less SLOT() usage where PMF or lambda works*) introduce more than 50 new *lambda expressions* distributed in 65 files. Listing 6 in Figure 6 shows one example of the changes in this particular commit.

***RQ4*: Do KDE developers conduct maintenance efforts having the sole goal of rejuvenating C++ code?** We found evidence that KDE developers conduct rejuvenation efforts to adopt *auto-typed variables*, *lambda expressions*, and *range-based for*—maintenance tasks whose goal was mainly to refactor the code to replace legacy features with new ones. Even considering an overly conservative approach, we identified more than 50 rejuvenation efforts, changing several hundreds of lines of code to modernize C++ code.

5.2 | Responses from KDE developers

We identified thirty-two developers from the KDE community who were responsible for driving the 57 commits that characterize the rejuvenation efforts according to our heuristics. We contacted these developers to understand the motivations that led them to rejuvenate the project's

```
for (int i = 0; i < ordinals.length(); ++i) {
- QAction *editionAction = editionMenu->addAction(ordinals[i], sm, SLOT(map()));
- sm->setMapping(editionAction, i + 1);
+ QAction *editionAction = editionMenu->addAction(ordinals[i]);
+ connect(editionAction, &QAction::triggered, p, [this, i]() {
+ setEdition(i + 1);
+ });
}
```

Listing 5 Example of introducing *lambda expressions* in a large commit from `kbibtex`

```
@@ -157,8 +157,12 @@ void KAnimatedButtonPrivate::updateIcons()
      newMovie = new QMovie(icon_path);
      frames = 0;
      newMovie->setCacheMode(QMovie::CacheAll);
- QObject::connect(newMovie, SIGNAL(frameChanged(int)), q, SLOT(_k_movieFrameChanged(int)));
- QObject::connect(newMovie, SIGNAL(finished()), q, SLOT(_k_movieFinished()));
+ QObject::connect(newMovie, &QMovie::frameChanged, q, [this](int value) {
+ movieFrameChanged(value);
+ });
+ QObject::connect(newMovie, &QMovie::finished, q, [this] {
+ movieFinished();
+ });
```

Listing 6 Example of introducing *lambda expressions* in a large commit from `kwidgetsaddons`

FIGURE 6 Example of introducing *lambda expressions*.

source code and received responses from 34.37% of the developers contacted. We used the TF metaphor (Section 3.2) and found that 7 (63.63%) of the 11 developers who responded to our contact are in the list of top developers of the KDE projects in which they conducted rejuvenation efforts. Also, top developers were responsible for 36 (63.2%) of the rejuvenation commits found by our heuristic. Table 2 presents the characteristics of the developers participating in our survey. The first column presents the identification of the participant in our study. The second column contains the project name on which this developer performed a modernization effort. The third column presents the number of contributions of the developer to the project. The fourth and last columns inform whether this developer is a core developer according to the TF metaphor.

***RQ7*: Are the core developers of the projects responsible for conducting rejuvenation efforts in KDE projects?** We used the TF metaphor and found that 36 (63.2%) of the 57 commits marking the largest rejuvenation efforts were made by the core developers of the project. Our results suggest that software rejuvenation is a high-level concern that requires the involvement of technical leaders to conduct large maintenance efforts.

Our data analysis consists of a process involving annotations extracted from the developers' responses following a two-step coding approach: the first consisted of the open coding process followed by a second step to categorize the codes according to the recommendations described by Saldana.[35] The first author conducted the coding under the supervision of the fourth author, who supported code validation and categorization. This process resulted in seven main categories and 35 codes. The followings are the categories and the codes associated with them.

- *Motivations to rejuvenate*: Code quality improvements, Avoid writing boilerplate code, Performance improvements, Modern features can reduce error-prone, Code rejuvenation can attrack new contributors, Modern features allow faster writing of code, Delays Software aging, Prevent the software from dying.
- *When rejuvenations occur*: When the benefits of modern features are considerable, When the code really needs to be rewritten, Incremental code rejuvenation for small changes, When the compiler supports the modern features, When the frameworks evolves, Rejuvenation efforts are conducted during bug fixing, During the implementation of modern C++ features, Developers conduct rejuvenation efforts when learning about the modern C++ features, Rejuvenation efforts are made when the quality of the code has already degraded.
- *Tooling support*: Tooling support to identify rejuvenation opportunities, Tooling support for large rejuvenation efforts, Automated code rejuvenation can introduce bugs, Tooling support can reduce rejuvenation costs.
- *Challenges*: High cost of making changes, Incompatibility of modern features with old versions of frameworks, Habit of using existing features makes it difficult to adopt modern features, Modern features may worsen readability,modern features that require a change in logic are harder to port automatically.

TABLE 2 Group of KDE developers that participated in the survey and conducted rejuvenation efforts.

Id (#)	Project name	Commits realized (#)	Is core developer?
P1	kphotoalbum	1602	Yes
P2	kwin	2473	Yes
P3	calligra	271	No
P4	partitionmanager	495	Yes
P5	akonadi	1688	Yes
P6	discover	6	No
P7	labplot	3801	Yes
P8	plasma-framework	2	Yes
P9	kdevelop	2133	Yes
P10	amarok	14	No
P11	okular	47	No

- *Existence of third-party libraries*: Existing features facilitate the work, Existing features have better integration with APIs from the same library, The features made available by third-party libraries meet expectations.
- *Decision to rejuvenate*: The project team decides when to rejuvenate, Individual members can decide to perform rejuvenation.
- *Direction for rejuvenation*: Framework libraries influence rejuvenation, The compiler influences the language version to use, Discussions of developers community influences the adoption of modern features, the community decides which compiler version to use.

5.2.1 | Motivations to rejuvenate

This category summarizes the motivations and factors found in this research that drive developers in the KDE community to rejuvenate their programs. The most emerging motivations pointed out by the participants are related to the benefits acquired by adopting the modern features of the C++ language in their software. All participants report on code quality improvements (increased code understanding, making code more concise, readability improvements, ease of maintenance activities, security improvements, and machine code quality improvements).

> My personal motivations for adopting new C++ features has been to write code which is easier to understand and more concise, as well as the convenience that these new features bring.
>
> Kphotoalbum—P1

> Lambda expressions come very handy in Qt's signal and slot connections, the code is more compact in such places … range based for-loops also allow for a more compact code. Same for the auto-keyword, especially when dealing with enums inside of class namespaces or so—this is where the auto-keyword can save a lot of space and a lot of typing work.
>
> Labplot—P7

Furthermore, participants (P1, P3, and P6–P10) report that the adoption of modern features like *lambda expressions*, *range-based for*, and *auto-typed variables* reduces the amount of boilerplate code and makes the code leaner. Another benefit reported by participant P3 was performance gain when adopting *range-based for*. Participants P5 and P10 report that they performed rejuvenation to attract new collaborators to the project. Participant P5 reports that the use of modern features makes programs less error-prone.

> A big factor is to reduce boilerplate code, to get the code much more readable (for instance range for instead of plain iterators.. at least where possible) speaking about iterators, also something like **auto it** = **myLisy**.**begin**(); is much more readable than **QList < QString > :: iterator ii** = **myList**.**begin**();.
>
> Plasma-framework—P8

> Range-for was already kind of present through some Qt macro kung-fu, so it's more getting rid of this magic to switch to the proper way of doing things, and getting some performance improvements by removing useless copy constructions …
>
> Calligra—P3

> My main motivation is better legibility and more concise and expressive code. I believe all three lead to code that is easier to maintain and less error-prone.
>
> Akonadi—P5

Finally, participant P10 reports that software rejuvenation must be performed periodically to slow down software aging and increase its lifespan; otherwise, the software may die.

> In the case of the linux client here at the company, I have been doing it weekly, because like, here I have been using GTK instead of Qt, but it is the same. Because GTK has evolved, the language has evolved and I want the client to be more readable, I do it weekly. In the end, if you don't do this, nobody will be interested anymore, the software will get old and will die … so this work has to be done periodically or it will die.
>
> Amarok—P10

***RQ6*: What are the reasons that motivate KDE developers to conduct maintenance efforts for code rejuvenation?** Our results show that KDE developers perceive that modern C++ features can improve code readability, reduce the error-prones, and simplify software maintenance. They use *lambda expressions*, *range-based for*, and *auto-typed variables* to eliminate boilerplate code by avoiding creating unnecessary classes, rewriting variables with long names, and making the code cleaner. In addition, developers rejuvenate the code base of their programs in an attempt to attract new contributors to the repository, as well as delay aging and prevent software from reaching the end of its useful life. Such reported benefits show some advantages in using modern C++ features. Also, KDE developers rejuvenate their programs' source code with the goal of attracting new contributors to the project.

5.2.2 | When rejuvenations occur

This category summarizes the situations, moments, and ways in which some of the rejuvenation efforts reported by developers in the KDE community are taking place. Developers P3, P8, and P10 reported making rejuvenation efforts as the framework evolves. Other rejuvenation efforts take place when the compiler supports modern features. According to developers P1 and P7, rejuvenation occurs gradually as small changes are made to the C++.

> It depends on the project mainly. Most of the time, a major Qt upgrade (Qt4=>Qt5, Qt5=>Qt6) will require a more modern C++, thus allowing the full feature set. But there are some projects that will simply use these features when they consider most distributions will have a modern compiler handling them.
>
> Calligra—P3

Some developers report some conditions that need to be evaluated to decide whether rejuvenation should be performed. Participants P1 and P8 perform rejuvenation only when the program code needs to be rewritten. Participant P1 reports that rejuvenation is always performed when the benefits of introducing a modern C++ feature are significant. Participant P9 reports that rejuvenation is performed when code quality has already deteriorated.

> it's more continuous and low intensity, new code tends to use more modern C++ features, old code is ported more when it really needs to be rewritten.
>
> Plasma-framework—P8

KDE developers report that rejuvenation efforts occur during code creation and review. Participant P5 reports that he takes rejuvenation actions when he fixes bugs in the program. On the other hand, rejuvenation efforts may also take place after a new feature is implemented, according to participants P4 and P5. Finally, participant P6 reports that some rejuvenation efforts are made when he learns the modern features of the C++ language.

> Often code modernization just happens by writing new modern code. Occasionally when doing major porting, e.g., Qt4 to Qt5, or Qt5 to Qt6.
>
> Partitionmanager—P4

> I usually end up modernizing a particular piece of code when I'm fixing some bug or implementing a feature there and I notice that the particular area of code base could benefit from a little bit of modernization.
>
> Akonadi—P5

KDE developers report various situations in which they perform rejuvenation efforts, such as when enhancing frameworks, performing code reviews to fix bugs, and implementing new features into the program. There are some criteria they use to decide whether or not to perform rejuvenation efforts, such as whether the compiler now supports modern language features. In addition, developers report that program rejuvenation is performed incrementally for small changes to the language.

5.2.3 | Tooling support

This category summarizes KDE developers' perceptions of tooling support to advance rejuvenation efforts in their programs. Most of the developers (81.8%) surveyed report using some tool to support rejuvenation efforts. Participants P3, P7, and P10 report using automated tools to identify opportunities for rejuvenation. Participant P1 reports that automated tools can be useful when major rejuvenation efforts are needed. In addition, participant P9 reports that tool support can reduce the cost of rejuvenation. The participants P3, P6, and P9 report that automated tools (like Clazy and Clang-tidy) have adequate tooling support for process rejuvenation efforts automatically. Finally, the automated tools to support program rejuvenation used by developers in the KDE community are Clazy,[**] Clang-tidy,[††] KDevelop,[‡‡] and Clion.[§§]

> We had some phases were we spent dedicated time on our side to modernize the code a bit. We used clang-tidy to identify and to adjust the code … more was done later manually in multiple further iterations from time to time. Right now these activities are more or less done, we follow our code style using the C++11 features and there is no need anymore for a "modernization effort."
>
> Labplot—P7

> Depending on the feature. C++11 was a huge improvement over C++98, which meant that a dedicated modernization effort was warranted. Tools that we used at this time were refactoring scripts … written by other KDE developers.
>
> Kphotoalbum—P1

> On the tooling side, we are pretty well equipped, with clang-tidy and clazy. I use both regularly, and I think the same is true for many other KDE community members. They are able to modernize some patterns automatically. clang-tidy is useful for general C++ modernization, while clazy handles Qt specific things.
>
> Discover—P6

> Clazy introduced checks that suggested rewriting code to use several C++11 features. If you look at the history of the *foreach* check (it suggests replacing an old *Q_FOREACH* macro with range-based for), you can see it was created back in 2015. […] Rewriting code to use auto or range-based for is simple and can be helped by tools (and I don't know about the other devs, but most of my for loops use auto variables, the two go together nicely). Using lambda, on the other hand, is something you will do when writing of fixing code for other purpose, not because a tool told you could use one there, but because you see a pattern where lambdas will be helpful.
>
> Calligra—P3

[**] Clazy is a static code analyzer based on the Clang framework.
[††] Clang-tidy is a clang-based C++ linter tool.
[‡‡] KDevelop is an extensible IDE plugin for C/C++ and other programming languages.
[§§] Clion CLion is a C/C++ development environment with many built-in features.

***RQ5*: Which tools do KDE developers use to support maintenance efforts for code rejuvenation?** Our findings show that KDE developers use tools (Clazy, Clang-tidy, KDevelop, and Clion) to support rejuvenation efforts. According to their reports, they mainly use the tools to identify scenarios that can be rejuvenated. In addition, automated tools can support major rejuvenation efforts and reduce the costs of these changes according to reports from developers in the KDE community. Finally, developers reported that features such as *range-based for* and *auto-typed variables* have adequate tooling support for process automation, while *lambda expressions* require additional cognitive effort from developers to be included in a code snippet of a program.

5.2.4 | Challenges

This category summarizes some factors that KDE developers believe can prevent programs from being rejuvenated. Most developers report at least one factor that can prevent or delay program rejuvenation. Participants P6, P8, and P10 report that some of the modern features can affect code readability. Developers P3 and P6 report that some features are not adopted due to incompatibility with older versions of the framework. In addition, participants P1 and P9 report that the cost of performing rejuvenation is high. Finally, participants P3 and P4 report that the habit of using legacy features of the C++ language can delay the adoption of modern C++ features. Finally, participant P5 reports that features that require logic changes are difficult to port automatically.

> So in a business context it's relatively difficult to get time/budget for doing these kind of things, usually it's being done when the code quality has degraded quite a bit already, i.e. when it's too late or people finally get too annoyed working with the legacy code base.
>
> Kdevelop—P9

> But some features have been integrated in Qt for a very long time, and the newcomers have a hard time rising above and replacing years of habits. For instance, std::thread. It's a welcome addition to the C++ standard library, and it was badly needed. But the need for an abstraction to system threading APIs is not new, and for a long time Qt had a QThread class, deeply rooted in its object and event models.
>
> Calligra—P3

> Mostly, Qt already has facilities for concurrent programming. Moving from these to different ones has a far higher cost as with the features mentioned in the first question. I would therefore expect the uptake to be slightly higher in newly written applications.
>
> Kphotoalbum—P1

> With auto, especially when using templated classes, writing new code is much more comfortable, since long type signatures don't need to be spelled out every time. Also, when changing the type of a variable, with auto the amount of code that needs to be changed becomes a lot less. However, it might even sacrifice some of the readability to speed of development. At least while using IDEs, this isn't a problem in practice.
>
> Discover—P6

> When KDE adopted C++11 we used some scripts to automatically port from the old connect() syntax to the new syntax [0], although regarding latest C++17 features (e.g. optional), those are hard to port automatically as usually change in code logic is needed.
>
> Akonadi—P5

KDE developers face some challenges in rejuvenating their programs, such as the incompatibility of legacy versions of the framework, and projects often need to maintain compatibility with older platforms that do not yet support the modern C++ features. Some features such as *auto-typed variables* can affect the readability of the code. Moreover, the habit of using legacy features may delay the adoption of modern features by developers. Another problem for developers is the high cost of program changes. Finally, some features require cognitive efforts to be introduced in the code and are difficult to port automatically.

5.2.5 | Existence of third-party libraries

This category summarizes the motivations that lead KDE developers to use existing features instead of modern features of the C++ language. In our quantitative analysis, we found that modern features for working with multithreading are rarely used by KDE developers. Some features that are not included in the standard library are available in third-party libraries (Qt) that facilitate developers' work. Participants P2, P3, P5, and P6 report that C++ language features provided by third-party libraries integrate better with developed APIs that support the same library. Participants P1, P7, and P10 report that these features meet the current needs of developers and that they do not need to make changes to adopt modern features immediately.

> This is available in Qt already for very long time and many projects, including LabPlot and Cantor, are using this. Right now there is no need and no plans on our side to move away from what Qt is offering. This can change in future, of course, but right now our spare time is better invested in other areas.
>
> LabPlot—P7

> Such preference is motivated by the fact that QThread is integrated with other Qt APIs better than say std::thread.
>
> Kwin—P2

> Multi-threading is mostly done by the means of QThread and QObjects. Methods of QObjects can be executed indirectly by the Qt metaobject system, and that is clever enough to call functions from the QThread which the object belongs to. For futures and async, there is a Qt framework (QtConcurrent) which provides similar APIs that are easier to integrate into existing Qt projects.
>
> Discover—P6

Our results show that KDE developers use the modern C++ features provided by the Qt library to work with threads and that these language features meet the current needs of developers. Moreover, the feature provided by the Qt library is easier to integrate into the APIs of their projects. According to these developers, there is no need for a migration to use the new library currently available in the standard library.

5.2.6 | Decision to rejuvenate

In this category, you will find those responsible for performing rejuvenation, that is, those who make the decision to rejuvenate a program by the developers of the KDE community. Most participants (P1–P6 and P9) report that the decision to rejuvenate is made by the development team. Participants P5–P7 and P10 report that this decision can also be made by an individual member.

> This is a team decision in each project. Of course, one looks at other projects (especially the frameworks libraries) as a reference, but ultimately every team can make their own rules.
>
> Kphotoalbum—P1

> The KDE Community is working on multiple bigger project like the Frameworks, Plasma but also such big applications like Krita, Kdenlive, LabPlot, Cantor, etc. There are code styles and minimal compiler requirements within every single project and there is no "global" prescription for what to use. The project maintainers decide on their own what they need and when. I'm mainly working on LabPlot and Cantor and we have C++11 as the minimum supported version now.
>
> Labplot—P7

The results presented show that the decision to rejuvenate a program in the KDE community is made by the development team and can also come from a single team member.

5.2.7 | Direction for rejuvenation

This category summarizes the factors that influence the use of modern features of the C++ language. Developers report that there is no direction from the community, but some factors may influence the decision of developers in the KDE community developers. Participants P1, P3, P4, and P5 report that framework libraries, when using modern C++ features, influence rejuvenation. Participants P6, P9, and P10 report that they can be influenced to use modern C++ features by following mailing list discussions, talks, and community events and C++ language. In addition, the

community decides which compiler version to use, which in turn influences which language version to use and which features are supported by the compiler.

> But the KDE Frameworks will have a bigger influence, and they are much more carefully driven than the applications. If one of the major framework was to require, say, coroutines (in the future obviously), it sure would influence the applications into using this feature even for things not related to the said framework.
>
> Calligra—P3

> The standard set in KDECompilerSettings.cmake is updated every now and then, after some discussion on some KDE mailing list (kde-devel, kde-core-devel) weighting the pros and cons.
>
> Kdevelop—P9

> Usually there is no organized modernization effort in KDE. People tend to start to modernize their own projects when they learn about the new possibilities, and if I remember correctly there were modern C++ talks at Akademy, which certainly helped adoption.
>
> Discover—P6

> Nowdays the modernization effor on C++ go a bit hand in hand with another effort as we are in the middle of the transition between Qt5 and Qt6, which will also bring a major, source and binary incompatible major release of the KDE frameworks, that leads to a general modernization effort.
>
> Plasma-framework—P8

Our findings suggest that KDE developers report that the community does not provide direction, but their participation in discussion lists, talks, and events can influence the decision to rejuvenate a program. In addition, the evolution of the frameworks influences the decision of developers in the KDE community to use modern C++ features.

5.3 | The primary reasons for the increased adoption of modern C++ features

As presented in Section 4, we found a trend (see Figure 3) of adoption for three modern C++ features (*auto-typed variables*, *lambda expressions*, and *range-based for*) that had increased substantially between 2015 and 2016. This trend motivated us to investigate and additional question: *What are the main reasons for the adoption rate of modern C++ features having increased substantially in this period?* Based on the responses of the interviewed developers and our findings (Sections 4, 5.1, and 5.2), three main reasons might explain that trend:

R 1. C++ projects within the KDE community are closely linked to the Qt framework. In 2016, version 5.7[¶¶] of the framework was released, which requires compilers to support the C++11 version (like Clang and GCC). According to our interviewees (P3, P4, and P8), during this period, developers began migrating to the Qt5 version of the framework and occasionally started adopting the modern C++ features of the language made available by the C++11 standard. Additionally, 55 (96.49%) commits that characterizes rejuvenation efforts identified by our heuristic, were made between 2016-2022. See quotes below with the developers' responses.

> It depends on the project mainly. Most of the time, a major Qt upgrade (Qt4=>Qt5, Qt5=>Qt6) will require a more modern C++, thus allowing the full feature set. But there are some projects that will simply use these features when they consider most distributions will have a modern compiler handling them. A lot of KDE projects will use tools like Clazy and Clang-tidy to help identify parts of code that need modernizing. Clazy is really Qt oriented while, as you surely know, Clang-tidy is a generic tool.
>
> Calligra—P3

> It's mostly individual decisions. However, there is also baseline determined by Qt requirements. E.g. Qt5 needs at least C++11 and Qt6 needs C++17, so all KDE software can depend at least on C++11 or C++17 if it targets Qt6.
>
> Partitionmanager—P4

[¶¶] https://wiki.qt.io/New_Features_in_Qt_5.7.

R 2. In 2015, the Clazy tool introduced changes[##] that suggest the use of modern features in the C++11 standard. Additionally, KDE developers (participants P3, P6, and P9) reported that features such as range-based for and auto have adequate tooling support for process automation (e.g., replacing a deprecated Q_FOREACH macro), while lambda expressions require additional cognitive effort from developers to be included in a code snippet of a program. See quote below.

> Clazy introduced checks that suggested rewriting code to use several C++11 features. If you look at the history of the *foreach* check (it suggests replacing an old *Q_FOREACH* macro with range-based for), you can see it was created back in 2015. [...] Rewriting code to use auto or range-based for is simple and can be helped by tools (and I don't know about the other devs, but most of my for loops use auto variables, the two go together nicely). Using lambda, on the other hand, is something you will do when writing of fixing code for other purpose, not because a tool told you could use one there, but because you see a pattern where lambdas will be helpful.
>
> Calligra—P3

R 3. According to P3 and P8, the KDE projects often need to maintain compatibility with older platforms that do not yet support the new C++ language constructions. This could explain why adoption slowed between 2012 and 2014. See quotes below with the developers' responses.

> One first big factor is that some projects can't adopt modern C++ that fast perhaps because they need to support old platforms where that is not possible (think about a commercial Windows app that perhaps still supported XP until a couple of years ago, or very old embedded systems..) We don't have this requirement as our target platform is primary linux/bsd distributions in their current release (and secondary android, windows and mac, but also for those, only recent releases).
>
> Plasma-framework—P8

> There is no major direction given. Each project does as it wants. For instance, calligra chose for a long time to keep compatibility with some very old libraries because one of our known users, Jolla, is still using Qt 5.6, prohibiting some new constructions from being used. But the KDE Frameworks will have a bigger influence, and they are much more carefully driven than the applications. If one of the major framework was to require, say, coroutines (in the future obviously), it sure would influence the applications into using this feature even for things not related to the said framework.
>
> Calligra—P3

6 | DISCUSSION

In this section, we discuss the implications of our results (Section 6.1) and present some limitations that might threat the validity of our work (Section 6.2).

6.1 | Implications of the results

We conducted a large-scale study of C++ projects in the KDE community. We evaluated the use of modern features such as *auto-typed variables*, *lambda expressions*, and *range-based for* in community projects and contacted developers who have made efforts to rejuvenate their programs. In this section, we summarize the implications of our study on some topics:

- We present a list of benefits and rejuvenation scenarios that developers can adopt in their programs. Our results for RQ1, RQ3, RQ4, and RQ6 show that using *lambda expressions*, *range-based for*, and *auto-typed variables* provides benefits such as readability, makes code cleaner, and simplifies software maintenance. KDE developers also report that using *lambda expressions* reduces the amount of boilerplate. Lucas et al[19] and Mendonça et al[36] show that developers also realize these benefits of using *lambda expressions* in Java programs. Our findings for RQ2 show that the use of *lambda expressions* is recommended by KDE developers in conjunction with resources for working with Qt's signals and slot connections in C++ programs.

[##] https://github.com/KDE/clazy/commits/master/src/checks/level1/foreach.cpp.

- Our findings for RQ5 demonstrate the use of automated tools (Clazy, Clang-tidy, KDevelop, and Clion) to help KDE developers identify rejuvenation scenarios. In addition, the tools can reduce the cost of rejuvenation efforts, which benefits developers in terms of the time it takes to complete these efforts. A broader audience of C++ developers can also benefit from these findings.
- Our results for RQ4 present a catalog of commits (Table A.1) that characterizes rejuvenation efforts. We hope the examples in the catalog could help software developers to rejuvenate their programs and tool builders to identify opportunities to implement source code transformations.
- Our findings for RQ7 revealed that core developers were responsible for most of the rejuvenation commits found to adopt modern C++ features in our dataset. It is similar to the occurrences reported by Mazinanian et al[5] for Java developers.
- KDE developers report that *lambda expressions* requires additional cognitive effort from developers to be included in a code snippet of a program. According to Uesbeck et al,[23] using *lambda expressions* negatively affects productivity for less experienced developers regarding how quickly they can write correct programs. Based on these results, we recommend that rejuvenation efforts to include *lambda expressions* be undertaken by experienced developers with knowledge of the codebase of programs.

6.2 | Threats to validity

Regarding *construct validity*, we have employed the TF metaphor[28] to identify the core developers of KDE projects. Our research findings indicate that the core developers contributed to 63.15% of the rejuvenation efforts identified through our conservative approach. Critics may question our decision to use TF as a proxy for identifying the core developers, but previous studies[30] have demonstrated the effectiveness of this metric. Furthermore, we assert that other metrics[29] could have yielded the same result, which is that technical leaders are responsible for significant rejuvenation efforts. Additionally, this result is consistent with our qualitative assessment, as seven out of 11 survey respondents were core developers in the projects.

The accuracy of our conservative approach to identifying rejuvenation efforts represents an *internal threat* for our research. In Section 5.1, we show that our conservative approach resulted in 57 (70.37%) of the rejuvenation efforts between 81 intervals we collect. We stressed that our initial research goal was to evaluate whether the KDE community developers conducted large rejuvenation efforts. To achieve this goal, we would not necessarily have to find all the contributions that characterize rejuvenation efforts, but some that showed the occurrence of this phenomenon in the KDE community projects. That is, here, we privilege precision over recall, and many other rejuvenation efforts might not have been captured using our conservative approach.

In terms of *external validity*, we analyzed 272 out of 1050 C++ projects from the KDE community. The projects we selected satisfy the criteria of having more than 10 years of development and at least one recent contribution (after January 1, 2021). This represents a wide range of application domains so that we can generalize our results to the KDE organization as a whole. However, studies in source code repositories from other organizations might reveal different results. Furthermore, we cannot generalize our findings to closed-source projects because we only look at open-source repositories. Besides that, we believe that developers in general can benefit from the rejuvenation practices we present in our study. Also related to *external validity*, we contacted 32 KDE developers that led the rejuvenation efforts found using our heuristic, to better understand their motivations to use modern C++ features. Eleven developers (34.37% of the initial population) answered our survey, which does not allow us to generalize our findings.

7 | CONCLUSIONS

During this research, we investigate the practices C++ developers use to rejuvenate open-source programs in the KDE organization. Our findings bring evidence about a trend toward the widespread adoption of modern C++ features, including *lambda expressions*, *range-based for*, and *auto-typed variables*. Nonetheless, to our surprise, the new concurrent support for C++ multithreading is rarely used in KDE projects. The main reason for this observation is the use of the Qt support for multithreading in KDE projects, which is not being replaced by the new standard multithreading features of C++.

Our research also revealed large modernization efforts, some of them replacing hundreds of occurrences of legacy constructs with the *range-based for* and *auto-typed variables* statements, for instance. Some of these efforts have even been conducted using the support of tools, such as Clazy and Clang-Tidy. After surveying KDE developers responsible for these large modernization efforts, we identified a couple of reasons that motivate this kind of software maintenance, which aims at improving the readability, conciseness, and expressiveness of the code, slowing software aging, and even attracting new contributors to the project repositories. We believe our findings could help a broader range of C++ developers in taking a decision on whether or not to rejuvenate their programs.

Our research findings indicate that developers perceive some benefits to adopting modern features of the C++ programming language and how they adopt them in their programs. Programming language designers need to know how developers embrace the new features of the

language and the impact on their programming productivity. We presume that this is an important aspect that merits a thorough investigation. For future work, we advise conducting exploratory research focused on the interests of software language designers. We also intend to look into the technical aspects to suggest improvements for automated tools and help developers to rejuvenate their software.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their valuable comments, which helped us improve the quality of this paper. This work was partially supported by FAP-DF, CAPES research grant 07/2019, and national funds through the Portuguese funding agency, Foundation for Science and Technology (FCT), within project LA/P/0063/2020 (INESC TEC INTERNATIONAL VISITING RESEARCHER PROGRAMME 2022 EDITION).

DATA AVAILABILITY STATEMENT

The data that support the findings of this study are openly available in cppEvolution at https://github.com/PAMunb/cppEvolution.

ORCID

Walter Lucas https://orcid.org/0000-0001-7391-9622
Fausto Carvalho https://orcid.org/0000-0002-6622-2330
Rafael Campos Nunes https://orcid.org/0000-0003-3769-6171
Rodrigo Bonifácio https://orcid.org/0000-0002-2380-2829
João Saraiva https://orcid.org/0000-0002-5686-7151
Paola Accioly https://orcid.org/0000-0002-4428-2543

REFERENCES

1. Stroustrup, B. Thriving in a crowded and changing world: C++ 2006-2020. *Proc ACM Program Lang*. 2020;4(HOPL):70:1-70:168. https://doi.org/10.1145/3386320
2. Overbey, JL, Johnson, RE. Regrowing a language: refactoring tools allow programming languages to evolve. In: Arora, S, Leavens, GT, eds. *Proceedings of the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2009, October 25–29, 2009, Orlando, Florida, USA*: ACM; 2009:493-502, https://doi.org/10.1145/1640089.1640127
3. Bragagnolo, S, Anquetil, N, Ducasse, S, Seriai, A, Derras, M. Software Migration: A Theoretical Framework (A Grounded Theory Approach on Systematic Literature Review), Inria Lille Nord Europe—Laboratoire CRIStAL—Université de Lille; 2021. Research Report. https://hal.inria.fr/hal-03171124
4. Alqaimi, A, Thongtanunam, P, Treude, C. Automatically generating documentation for lambda expressions in Java. In: Storey, M-AD, Adams, B, Haiduc, S, eds. *Proceedings of the 16th International Conference on Mining Software Repositories, MSR 2019, 26–27 May 2019, Montreal, Canada*: IEEE/ACM; 2019:310-320, https://doi.org/10.1109/MSR.2019.00057
5. Mazinanian, D, Ketkar, A, Tsantalis, N, Dig, D. Understanding the use of lambda expressions in Java. *Proc ACM Program Lang*. 2017;1(OOPSLA):85:1-85:31. https://doi.org/10.1145/3133909
6. Kumar, A, Sutton, A, Stroustrup, B. Rejuvenating C++ programs through demacrofication. *28th IEEE International Conference on Software Maintenance, ICSM 2012, Trento, Italy, September 23–28, 2012*: IEEE Computer Society; 2012:98-107, https://doi.org/10.1109/ICSM.2012.6405259
7. Pirkelbauer, P, Dechev, D, Stroustrup, B. Source code rejuvenation is not refactoring. In: van Leeuwen, J, Muscholl, A, Peleg, D, Pokorný, J, Rumpe, B, eds. *SOFSEM 2010: Theory and Practice of Computer Science, 36th Conference On Current Trends in Theory and Practice of Computer Science, Špindlerův Mlýn, Czech Republic, January 23–29, 2010. Proceedings*, Lecture Notes in Computer Science, vol. 5901: Springer; 2010:639-650, https://doi.org/10.1007/978-3-642-11266-9_53
8. Stroustrup, B. Evolving a language in and for the real world: C++ 1991-2006. *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, HOPL III: Association for Computing Machinery; 2007:4-1–4-59, https://doi.org/10.1145/1238844.1238848
9. Stepanov, A, Lee, M. *The Standard Template Library*, Vol. 1501: Hewlett Packard Laboratories; 1995.
10. Stroustrup, B. *The C++ Programming Language*. 3rd ed.: Addison-Wesley Longman Publishing Co., Inc.; 1997. ISBN 0201889544.
11. Josuttis, NM. *C++ 17: The Complete Guide*: Nicolai Josuttis; 2019. ISBN 9783967300178. https://books.google.com.br/books?id%3DUAmQzQEACAAJ
12. Meyers, S. *Effective Modern C++: 42 Specific Ways to Improve Your Use of C++11 and C++14*. 1st ed.: O'Reilly Media, Inc.; 2014. ISBN 1491903996.
13. Schäling, B. *The Boost C++ Libraries*: XML Press; 2014. ISBN 9781937434366 1937434362.
14. Stroustrup, B. *The C++ Programming Language*. 4th ed.: Addison-Wesley Professional; 2013. ISBN 0321563840.
15. O'Dwyer, A. *Mastering the C++17 STL: Make Full Use of the Standard Library Components in C++17*: Packt Publishing; 2017. ISBN 178712682X.
16. Deitel, P, Deitel, H. *C++20 for Programmers: An Objects-Natural Aproach*, Deitel Developer Series: Pearson Education Canada; 2022. ISBN 9780136905691. https://books.google.com.br/books?id%3Dcih_zQEACAAJ
17. Rajlich, V. Software evolution and maintenance. In: Herbsleb, JD, Dwyer, MB, eds. *Proceedings of the Future of Software Engineering, FOSE 2014, Hyderabad, India, May 31–June 7, 2014*: ACM; 2014:133-144, https://doi.org/10.1145/2593882.2593893
18. Dantas, R, Carvalho, A, Marcilio, D, et al. Reconciling the past and the present: an empirical study on the application of source code transformations to automatically rejuvenate Java programs. In: Oliveto, R, Penta, MD, Shepherd DC, eds. *25th International Conference on Software Analysis, Evolution and Reengineering, SANER 2018, Campobasso, Italy, March 20–23, 2018*: IEEE Computer Society; 2018:497-501, https://doi.org/10.1109/SANER.2018.8330247
19. Lucas, W, Bonifácio, R, Canedo, ED, Marcilio, D, Lima, F. Does the introduction of lambda expressions improve the comprehension of Java programs? In: do Carmo Machado, I, Souza, R, Maciel, RSP, Sant'Anna, C, eds. *Proceedings of the XXXIII Brazilian Symposium on Software Engineering, SBES 2019, Salvador, Brazil, September 23–27, 2019*: ACM; 2019:187-196, https://doi.org/10.1145/3350768.3350791

20. McCabe, TJ. A complexity measure. *IEEE Trans Software Eng*. 1976;2(4):308-320. https://doi.org/10.1109/TSE.1976.233837
21. Buse, RPL, Weimer, W. A metric for software readability. In: Ryder, BG, Zeller, A, eds. *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2008, Seattle, WA, USA, July 20–24, 2008*: ACM; 2008:121-130, https://doi.org/10.1145/1390630.1390647
22. Posnett, D, Hindle, A, Devanbu, PT. A simpler model of software readability. In: van Deursen, A, Xie, T, Zimmermann, T, eds. *Proceedings of the 8th International Working Conference on Mining Software Repositories, MSR 2011 (co-located with ICSE), Waikiki, Honolulu, HI, USA, May 21–28, 2011, Proceedings*: ACM; 2011:73-82, https://doi.org/10.1145/1985441.1985454
23. Uesbeck, PM, Stefik, A, Hanenberg, S, Pedersen, J, Daleiden, P. An empirical study on the impact of C++ lambdas and programmer experience. *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16: Association for Computing Machinery; 2016:760-771, https://doi.org/10.1145/2884781.2884849
24. Zheng, M, Yang, J, Wen, M, Zhu, H, Liu, Y, Jin, H. Why do developers remove lambda expressions in Java? *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering*, ASE '21: IEEE Press; 2022:67-78, https://doi.org/10.1109/ASE51524.2021.9678600
25. Chen, L, Wu, D, Ma, W, Zhou, Y, Xu, B, Leung, H. How C++ templates are used for generic programming: an empirical study on 50 open source systems. *ACM Trans Softw Eng Methodol*. 2020;29(1):1-49. https://doi.org/10.1145/3356579
26. Ricca, F, Marchetto, A, Torchiano, M. On the difficulty of computing the truck factor. In: Caivano, D, Oivo, M, Baldassarre, MT, Visaggio, G, eds. *Product-Focused Software Process Improvement—12th International Conference, PROFES 2011, Torre Canne, Italy, June 20–22, 2011. Proceedings*, Lecture Notes in Business Information Processing, vol. 6759: Springer; 2011:337-351, https://doi.org/10.1007/978-3-642-21843-9_26
27. Bosu, A, Sultana, KZ. Diversity and inclusion in Open Source Software (OSS) projects: where do we stand? *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM 2019, Porto de Galinhas, Recife, Brazil, September 19–20, 2019*: IEEE; 2019:1-11, https://doi.org/10.1109/ESEM.2019.8870179
28. Avelino, G, Passos, LT, Hora, AC, Valente, MT. A novel approach for estimating truck factors. *24th IEEE International Conference on Program Comprehension, ICPC 2016, Austin, TX, USA, May 16–17, 2016*: IEEE Computer Society; 2016:1-10, https://doi.org/10.1109/ICPC.2016.7503718
29. Ferreira, MM, Valente, MT, Ferreira, KAM. A comparison of three algorithms for computing truck factors. In: Scanniello, G, Lo, D, Serebrenik, A, eds. *Proceedings of the 25th International Conference on Program Comprehension, ICPC 2017, Buenos Aires, Argentina, May 22–23, 2017*: IEEE Computer Society; 2017:207-217, https://doi.org/10.1109/ICPC.2017.35
30. Canedo, ED, Bonifácio, R, Okimoto, MV, Serebrenik, A, Pinto, G, Monteiro, E. Work practices and perceptions from women core developers in OSS communities. In: Baldassarre, MT, Lanubile, F, Kalinowski, M, Sarro, F, eds. *ESEM '20: ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, Bari, Italy, October 5–7, 2020*: ACM; 2020:26:1-26:11, https://doi.org/10.1145/3382494.3410682
31. Myers, L, Sirois, MJ. *Spearman Correlation Coefficients, Differences between*: John Wiley & Sons, Ltd; 2006. https://onlinelibrary.wiley.com/doi/abs/10.1002/0471667196.ess5050.pub2
32. De Winter, JCF, Gosling, SD, Potter, J. Comparing the Pearson and Spearman correlation coefficients across distributions and sample sizes: a tutorial using simulations and empirical data. *Psychol Methods*. 2016;21(3):273.
33. Järvi, J, Freeman, J. Lambda functions for C++0x. In: Wainwright, RL, Haddad, H, eds. *Proceedings of the 2008 ACM Symposium on Applied Computing (SAC), Fortaleza, Ceara, Brazil, March 16–20, 2008*: ACM; 2008:178-183, https://doi.org/10.1145/1363686.1363735
34. Becker, P. ISO/IEC 14882:2011 Information technology—Programming languages—C++. https://www.iso.org/standard/50372.html. [Online; accessed 03-May-2022]; 2011.
35. Saldana, J. *The Coding Manual for Qualitative Researchers*, English Short Title Catalogue Eighteenth Century Collection, publisher=SAGE Publications; 2012. ISBN 9781446247372. https://books.google.com.br/books?id%3DkUms8QrE_SAC
36. de Mendonça, WLM, Fortes, J, Lopes, FV, et al. Understanding the impact of introducing lambda expressions in Java programs. *J Softw Eng Res Dev*. 2020;8:7:1-7:22. https://doi.org/10.5753/jserd.2020.744

AUTHOR BIOGRAPHIES

Walter Lucas is a PhD student in Computer Science at the University of Brasília, Brazil, since 2019. He earned a master's degree in Computer Science from the University of Brasília, Brazil, in 2019. He obtained a degree in Information Systems at the University Center of Patos de Minas in 2015, working in Software Engineering, in the areas of Code Comprehension, Software Evolution and Program Transformations.

Fausto Carvalho is a software engineer at the Brazilian Public Prosecutor's Office, with a degree in Computer Science from the Catholic University of Brasília, Brazil. His areas of interest are software security, programming languages, software maintenance, and evolution.

Rafael Campos Nunes is a Computer Science BSc undergraduate student at the University of Brasília. Currently, he is a software engineer working in the Communications Industry building the next generation of streaming CDNs and is interested in studying software engineering and programming languages.

Rodrigo Bonifácio is an associate professor at the University of Brasília, Brazil. His main research interests focus on program analysis and manipulation, with particular emphasis on software maintenance and evolution and software security.

João Saraiva is an associate professor in the Department of Informatics at the University of Minho, Braga, Portugal, and a senior researcher at HASLab/INESC TEC. He obtained a master's degree from the University of Minho in 1993 and a PhD degree in Computer Science at the University of Utrecht in 1999. His areas of interest are programming language design and implementation, program analysis and transformation, and functional programming.

Paola Accioly holds a PhD in Computer Science from the Federal University of Pernambuco and is an assistant professor at the Federal University of Cariri. She obtained a master's degree in Computer Science from the Federal University of Pernambuco (2012) and a degree in Computer Science from the Federal University of Pernambuco (2009). Her thesis focuses on software engineering, specifically on collaborative software development support and empirical software engineering.

How to cite this article: Lucas W, Carvalho F, Nunes RC, Bonifácio R, Saraiva J, Accioly P. Embracing modern C++ features: An empirical assessment on the KDE community. *J Softw Evol Proc*. 2023;e2605. doi:10.1002/smr.2605

APPENDIX A

TABLE A1 Catalog of commits that characterize rejuvenation efforts.

Project name	Commit date	Commit hash	Modern feature adopted	Files modified	Additions	Deletions
akonadi	02/04/21	ddd7758e07917d7968177c0cfdaa82d620ca8d4b	Auto-typed variables	143	709	709
akonadi	04/23/17	5535d6e5604d3bc800868ba5ea820fd6ffeb3d24	Auto-typed variables	166	5943	13,311
akonadi-contacts	11/02/20	d7bea9b44aece787d7faf42b419f6dffbd468ca1	Auto-typed variables	78	237	237
akregator	02/03/21	b65b4e407a80324f528efa1a8bc3a6abd9082d33	Auto-typed variables	37	73	73
akregator	11/02/20	7e6ff7d7b14c47db21b71518edbdec44367574bd	Auto-typed variables	41	94	94
baloo-widgets	06/01/22	93b3e120c96469a28b423ddf646e397d30e3a232	Auto-typed variables	18	55	73
gwenview	02/09/22	02ce8d7907048beb1c8aebd3d5a4b485c2c0c349	Auto-typed variables	83	312	312
kdenlive	04/14/17	387199e1a602cd1101916ed2a96b0e8b3e8e86c5	Auto-typed variables	178	1401	1420
kdepim-runtime	02/04/21	fdc00bcca3ad00362dc310238aae1761f6966f19	Auto-typed variables	5	9	9
kdevelop	01/22/20	4626d69f68ea9e3af81f5324167d5608a5505c45	Auto-typed variables	114	280	282
kid3	09/30/18	4fa1900eed4091f4100b244df07311e62208a1ab	Auto-typed variables	65	283	557
kid3	09/29/18	f519ce07581980b95599227307eb46c8bc7f9abe	Auto-typed variables	88	495	495
kmail	11/01/20	3055a93326cc6b90cba8c50d72b8ab0d1d33c2ac	Auto-typed variables	130	524	524
knotes	02/04/21	0c6207800da3a16b8c0b4dda171c4cc7e4628e80	Auto-typed variables	37	133	133
knotes	05/02/21	4d1e8e4d60e95b63286ac0103fed2e0310a3cd92	Auto-typed variables	35	131	130
konversation	06/15/21	7c6ed0b7658a5c2ea66d6032c569012258847b91	Auto-typed variables	63	252	252
korganizer	02/03/21	003270471b1314abf46226a92a1d32d808a83219	Auto-typed variables	9	13	13
kpimtextedit	11/01/20	c6fa90c5c0bf719439fd9e2d2678eb0e329ff624	Auto-typed variables	47	125	125
krfb	10/23/20	a109e3d6c9cf5e312567ec5d5d98534457ec14b9	Auto-typed variables	7	18	19
krusader	02/24/19	59ad209a23d28ea6e745a097bf06e889b8dfb88b	Auto-typed variables	96	455	455
okteta	02/22/19	c4c70a1e758b3fb319c0e41d3dd6ce787bf6194a	Auto-typed variables	173	456	496
systemsettings	10/07/21	68273cc95d6760f0e4289538e19985b041bcaa5f	Auto-typed variables	15	288	404
zanshin	02/11/15	3dfeb5fed3ceaead480444cb2be8d4e6e879ca69	Auto-typed variables	12	18	18
calligra	03/13/21	33973e1295b	Lambda expressions	9	33	29
kbibtex	07/20/19	40a2aee94fdc14b66b6806d552b599de3b4e64c1	Lambda expressions	16	367	435
kwidgetsaddons	05/07/21	034e0b7b1d5af08136e38a9eb7df4ee514fe385b	Lambda expressions	65	446	426
gwenview	07/13/21	222ed1fa0c40fb036b19bd5ecdd09a0faeb6c016	Auto-typed variables, lambda expressions	6	155	138

(Continues)

TABLE A1 (Continued)

Project name	Commit date	Commit hash	Modern feature adopted	Files modified	Additions	Deletions
plasma-framework	05/07/15	ea924b14691da3819ca17d876f63ffc686fcf840	Auto-typed variables, lambda expressions, range-based for	22	385	366
akonadi	12/17/16	46badf63d00f46cdb8ffcb2ff45791d4a6d63173	Range-based for	54	115	115
discover	05/21/21	13891987fbeba55d2ecc6f6ec4cf948f7c34d460	Range-based for	35	111	92
dolphin	10/23/20	a24327cd50ef17b953ecb908d260b73460158107	Range-based for	38	140	121
granatier	05/15/18	8a03ba70008f7f2c73cdd3f3d03b9ed7e922a37e	Range-based for	9	108	120
granatier	04/07/16	1535c053127e485fd6d6851f9938c677763eb00b	Range-based for	4	59	59
kalarm	08/11/19	61590fae7ba5fd8bea501c68f2f754af3f74776f	Range-based for	17	598	663
kate	09/24/19	0bbb048bd2255c1082b939c2013bf3dd0b99c2a4	Range-based for	9	15	13
kcontacts	05/10/21	3971b6278f62a9568792f7f41bd2e4ff46daf021	Range-based for	23	222	343
kdenlive	04/20/17	2e508acc340ff1d6ddda55e538c9fe01f09975f6	Range-based for	59	226	228
kdepim-runtime	12/28/16	d10456bd725e2e724c778a58a779d850f1b69420	Range-based for	29	57	58
kdesvn	07/31/19	ad2ef6dd1302af5d04a97798a6ea4b01fc60e5ee	Range-based for	1	72	76
kimap	01/20/21	c5278fe4c77329b64cef6eb097665d4b1e2e16a3	Range-based for	7	40	20
kio	10/21/19	103e13c2765e7fb587fd778c1d04c90d3af42aff	Range-based for	1	8	11
kio-extras	04/05/19	a50a8f8082baef51132562c85931c9e7ab9c7814	Range-based for	14	45	64
kleopatra	02/27/17	96409339	Range-based for	9	19	16
konsole	11/24/19	c83bb19a68a65a59e149586d809c442168ba35aa	Range-based for	13	55	51
konversation	10/19/20	9f9bf118	Range-based for	6	27	43
kphotoalbum	04/10/20	233c3d7fc43217f6acf2a4e2db915bf5747cb50c	Range-based for	53	206	206
ktorrent	06/29/17	7f8795555185226bb17909b8a954592fa7c947de	Range-based for	14	66	66
ktorrent	07/06/17	2ad882b61101dfe8414e8f47f89360f4e2d96754	Range-based for	10	47	45
kwin	08/08/19	91faa589c70f9ca9a9d518ac072e186846c88827	Range-based for	1	184	112
labplot	12/03/17	5c28a3e5f994fbc586519fad3ec2aadff7357070	Range-based for	36	388	389
marble	01/27/17	33eff9bc4ea9581dc3d84783f62747463f2c047e	Range-based for	176	537	532
okular	03/26/19	f34ebf659f6cd93233cca345d873bd54c039b83a	Range-based for	3	50	90
partitionmanager	08/31/16	e7ac5e5fa2800c9583ec1223abd1fce2a6e51a50	Range-based for	6	30	16
krusader	02/24/19	15ca9d93c97b0729e29c74f38e6cdeb31793d09c	Auto-typed variables, range-based for	22	148	153
amarok	09/26/20	ce93fb34c396f1b7766daaf43669d82f949ff091	Auto-typed variables	104	7170	783
kdevelop	10/25/18	c50f4442ccbb6ecb95d2e0d7f0484060bddb5747	Lambda expressions	343	1224	1224
kwin	07/14/18	049c6e0966ceda533e023a22f29a5d43a65cef0c	Auto-typed variables	1	4	4

Appendix B

Appendix B — JavaScript Study

Empirical Software Engineering (2025) 30:107
https://doi.org/10.1007/s10664-025-10663-9

Understanding the adoption of modern Javascript features: An empirical study on open-source systems

Walter Lucas[1] · Rafael Nunes[1] · Rodrigo Bonifácio[1] · Fausto Carvalho[1] · Ricardo Lima[1] · Michael Silva[1] · Adriano Torres[1] · Paola Accioly[2] · Eduardo Monteiro[1] · João Saraiva[3]

Accepted: 21 April 2025

Abstract
JavaScript is a widely used programming language initially designed to make the Web more dynamic in the 1990s. In the last decade, though, its scope has extended far beyond the Web, finding utility in backend development, desktop applications, and even IoT devices. To circumvent the needs of modern programming, JavaScript has undergone a remarkable evolution since its inception, with the groundbreaking release of its sixth version in 2015 (ECMAScript 6 standard). While adopting modern JavaScript features promises several benefits (such as improved code comprehension and maintenance), little is known about which modern features of the language have been used in practice (or even ignored by the community). To fill this gap, in this paper, we report the results of an empirical study that aims to understand the adoption trends of modern JavaScript features, and whether or not developers conduct rejuvenation efforts to replace legacy JavaScript constructs and idioms with modern ones in legacy systems. To this end, we mined the source code history of 158 JavaScript open-source projects, identified contributions to rejuvenate legacy code, and used time series to characterize the adoption trends of modern JavaScript features. The results of our study reveal extensive use of JavaScript modern features which are present in more than 80% of the analyzed projects. Our findings also reveal that (a) the widespread adoption of modern features happened between one and two years after the release of ES6 and, (b) a consistent trend toward increasing the adoption of modern JavaScript language features in open-source projects and (c) large efforts to rejuvenate the source code of their programs.

Keywords Software evolution · Software engineering · Software maintenance · Source code rejuvenation · Mining software repositories

1 Introduction

Originally designed to enhance interactivity on Web pages, JavaScript has evolved into a versatile language used across web, mobile, and server-side development. Different JavaScript environments are available today to support a wide range of applications, including back-

Communicated by: Yasutaka Kamei

Extended author information available on the last page of the article

Published online: 05 May 2025

end and desktop development, as well as emerging technologies such as smart contracts and IoT. Additionally, JavaScript serves as a compilation target for languages such as Reason and TypeScript, enabling their code to run in JavaScript engines. It also underpins rendering frameworks like Vue.js, ReactJS, and many others (Silva et al. 2015).

By 1996, the JavaScript standardization process was initiated under Ecma International, leading to the first official ECMAScript specification in 1997 (Ecma 1997). Two additional revised editions were released by the end of 1999, reflecting Netscape's enhancements to the language. However, the early 2000s saw a period of stagnation. In 2008, the committee restored its operations, resulting in a modestly enhanced edition published in 2009. This success paved the way for the major enhancements seen in ECMAScript 2015 (ES6), which laid the foundation for JavaScript's recent evolution (Wirfs-Brock and Eich 2020; Flanagan 2020).

Post-2015, the committee adopted a process for faster, incremental releases, completing revisions on a yearly schedule. This structured approach has allowed JavaScript to continuously adapt and grow, maintaining its dominance in web development, where the evolution of the language has introduced numerous features-such as ES6 modules, asynchronous programming with Promises, and syntax improvements (Wirfs-Brock and Eich 2020).

However, adapting legacy code to newer JavaScript features is challenging and involves trade-offs between various factors, including the benefits of new features, project requirements, and compatibility with target environments (e.g., JavaScript engines like V8) (Nicolini et al. 2024). Even more importantly, rejuvenating old code to leverage new language features and idioms can require a substantial time investment-although previous research suggests that the expected benefits of source code rejuvenation pay off (Lucas et al. 2023).

Previous research has investigated the practices developers follow to rejuvenate their code and understand the motivations and challenges behind these efforts and trends for adopting programming language features. Albeit focusing on statically typed languages such as Java (Dyer et al. 2014a; Mazinanian et al. 2017; Dantas et al. 2018; Lucas et al. 2019; Zheng et al. 2021), C++ (Kumar et al. 2012; Uesbeck et al. 2016; Chen et al. 2020; Lucas et al. 2023), Kotlin (Mateus and Martinez 2020), and TypeScript (Scarsbrook et al. 2023). For instance, Mazinanian et al. (2017) investigate the adoption of lambda expressions in Java, particularly focusing on how developers transition to functional programming, while Lucas et al. (2023) show that C++ software developers contributing to KDE projects often conduct large source code rejuvenation efforts.

Despite the widespread use of the JavaScript programming language, relatively few studies have explored the efforts involved in adopting modern JavaScript features, as well as the motivations and challenges driving these efforts (Silva et al. 2015; Gokhale et al. 2021; Alves et al. 2022; Nicolini et al. 2024). This paper aims to advance the existing literature by addressing these gaps through the findings of an empirical study that analyzes the source code history of open-source JavaScript projects, shedding light on the adoption and rejuvenation efforts in this context.

Accordingly, we investigate (a) the adoption trends of modern JavaScript (JavaScript) features, (b) the extent of developers' efforts to rejuvenate legacy JavaScript source code, and (c) the motivations driving developers to rejuvenate legacy JavaScript code, as well as the challenges that hinder such efforts. Our study evaluated the adoption of modern JavaScript features in 158 open-source GitHub repositories created at least three years before the release of ES6, a landmark in modern JavaScript. Here, we provide an in-depth study on the adoption of modern JavaScript features in legacy open-source code, highlighting:

- **Early Adoption of Features**: Some modern JavaScript features are utilized even before their official release, reflecting developers' anticipation and readiness to embrace advancements in the language.
- **Variation in Feature Popularity**: While some features seem quite popular among the projects in our dataset, such as Const Declarations, Async Declarations, and Arrow Function Declarations that appear in more than 80% of the projects, others seem not that popular (like Computed Property Assignment from ES6 that appear in less than 35% of the projects).
- **Motivations for Adoption:** After analyzing code review discussions on source code rejuvenation, we identified several motivations for adopting modern features, including improved code readability, conciseness, and maintainability. These motivations align with trends observed in previous studies on statically typed languages (e.g., Java (Mazinanian et al. 2017) and Kotlin (Mateus and Martinez 2020)) and a prior study on JavaScript that examines the adoption of classes (Silva et al. 2015). In contrast, our study focuses on a broader set of JavaScript features.
- **Challenges in Adoption**: Compatibility issues, resistance to change, and performance concerns present significant barriers, underscoring the importance of strategies to mitigate these challenges.
- **Code Rejuvenation Practices**: Developers actively refactor and update existing codebases to incorporate modern features after their releases, emphasizing the need for tools that assist in identifying and automating these transformations.

This paper is organized as follows: Sections 2 present some background information about the evolution of JavaScript language. In Section 3 we detail our study settings. Section 4 present the first study, which used descriptive statistics and time series to examine the adoption trends of modern JavaScript features, Section 5 present a second study, which used thematic analysis of code review comments to highlight the motivations and challenges of updating JavaScript code. Section 6 present more details about our paper findings and implications. Section 7 compare our study with previous work about source software rejuvenation, adoption trends, motivations, and challenges from diverse programming languages in the literature. Finally, Section 8 present our conclusions and future works.

2 Background

In this section, we introduce the concept of source code rejuvenation (Section 2.1) and provide a brief overview of the evolution of the JavaScript language (Sections 2.2 and 2.3), highlighting the introduction of modern features. Section 2.4 then discusses the challenges of JavaScript cross-version compatibility and its potential impact on the rapid adoption of new language features.

2.1 Source Code Rejuvenation

Source code rejuvenation refers to the process of updating and transforming legacy code to incorporate modern programming language features and idioms (Pirkelbauer et al. 2010). Unlike general maintenance tasks, this approach focuses specifically on replacing outdated constructs and idioms with contemporary equivalents to align with advancements in language design. By addressing inconsistencies and legacy patterns, source code rejuvenation improves

```
var userName = "Alice";
var userAge = 30;
if (userAge > 18) {
    var isAdult = true;
}
console.log(isAdult);
```

```
const userName = "Alice";
const userAge = 30;
if (userAge > 18) {
    const isAdult = true;
    console.log(isAdult);
}
console.log(isAdult);
```

Fig. 1 Replacing `var` with `const`. The code on the left uses the `var` construct, while the code on the right uses the new `const` feature

code comprehension and maintainability, providing developers with a more cohesive and up-to-date codebase (Lucas et al. 2019, 2023).

For instance, modern JavaScript introduces Const Declarations and Let Declarations as block-scoped variable declarations that replace the function-scoped 'var', and Arrow Function Declarations as a concise syntax for defining anonymous functions (Rauschmayer 2015). Using Const Declarations ensures variables are immutable by default and enforces block scope, reducing the risk of unintended variable overwrites and improving code clarity. Figure 1 shows an example of a rejuvenation effort to adopt Const Declarations.

On left side, the legacy code uses `var`, which allows variable reassignment and does not enforce block scope. As a result, the variable `isAdult`, declared inside the `if` block, remains accessible outside of it. In contrast, the right side presents the rejuvenated code, where `var` is replaced with `const`, making variables immutable and ensuring they are confined to their respective blocks (Rauschmayer 2015). This prevents unintended modifications and scoping issues. Consequently, attempting to access `isAdult` outside its block results in a `ReferenceError`.

Arrow Function Declarations provide a shorter syntax and lexically bind the `this` context, making them particularly useful for callbacks and functional programming paradigms (Rauschmayer 2015). By systematically applying such transformations, developers can align codebases with modern JavaScript standards. On the left side, the legacy code defines a function using a traditional function expression, which requires an explicit `return` statement. In contrast, the right side showcases the rejuvenated version, where an arrow function replaces the function expression, reducing verbosity.

In previous work, Pirkelbauer and colleagues advocate that source code rejuvenation is not refactoring (Pirkelbauer et al. 2010). The main reason is that, unlike refactoring, source code rejuvenation focuses specifically on replacing outdated constructs and idioms with contemporary equivalents to align with advancements in language design (Fig. 2). In this paper, we prefer not to use the term refactoring for rejuvenation efforts, in particular, because the kinds of transformations we discuss here do not appear in refactoring references for JavaScript (Silva et al. 2021; Brito et al. 2022) and other programming languages as Java (Tsantalis et al. 2018).

```
function add(a, b) {
    return a + b;
}
console.log(add(1, 2));
```

```
const add = (a, b) => a + b;
console.log(add(1, 2));
```

Fig. 2 Replacing function expressions definitions with Arrow Function Declarations. The code on the left uses the a conventional syntax for implementing functions; while the code on the right use the Arrow Function Declarations feature introduced in ES6

2.2 Historical JavaScript

The JavaScript programming language, more formally known as ECMAScript (ES for short), was initially designed to make the World Wide Web more dynamic. Currently, it is widely used in many other fields of software development besides the Web, although it was not originally conceived with that intent. Brendan Eich led the initial design of the JavaScript language, which at the time was referred to as Mocha. After first appearing on Netscape Navigator, it went through significant changes, evolving from LiveScript to its third version and eventually being named JavaScript. This nomenclature change reflected the integration of certain syntactic elements inspired by the Java programming language (Wirfs-Brock and Eich 2020).

To relieve the problems of having different JavaScript flavors, the Netscape company asked the European Computer Manufacturers Association (ECMA) to standardize the JavaScript syntax and semantics as a general-purpose, cross-platform, vendor-neutral scripting language. This effort led to the ECMA-262 standard, and other browsers that aim to benefit from JavaScript could use the specification to implement a JavaScript runtime (Keith 2005). Nowadays, JavaScript is ubiquitous and has significantly changed in the past three decades of development. At the time of this writing, there are 14 official releases of the language, broadly used to develop frontend and backend components for web, mobile, and desktop applications, ranging from REST-based applications to more sophisticated ones. On the backend, JavaScript is mainly used with a runtime (such as NodeJS) that exposes APIs commonly found in other programming languages. The initial standardized version of JavaScript (ES1 standard) supported fundamental imperative programming features such as variables and function declarations, as well as conditional and repetitive statements (Wirfs-Brock and Eich 2020). In the subsequent versions, ECMAScript versions 2 and 3, the language was extended to include support for error handling, enhanced the support for regular expressions, and introduced new array manipulation capabilities (Wirfs-Brock and Eich 2020). Curiously, JavaScript version 4 was never published, as its development was stalled due to internal discussions about the future directions of the language (Wirfs-Brock and Eich 2020).

The ES5 standard, released in 2009, introduced significant improvements. It supports additional capabilities for string manipulation and the implementation of the JSON (Javascript Object Notation) data (de) serialization; object-oriented programming features; and a functional style for array manipulation, including methods such as `reduce()`, `filter()`, and `map()` (Wirfs-Brock and Eich 2020; Resig et al. 2016; Flanagan 2020).

Next we introduce more recent JavaScript releases, collectively referred to as "Modern JavaScript."

2.3 Modern JavaScript

The release of the ES6 standard in 2015 corresponds to a significant milestone in the evolution of JavaScript. ES6 introduces the support for the repetition statement `for ...of` on *iterable objects*; template literals; rest parameters (a.k.a., variadic functions in other languages); and lambda expressions (i.e., arrow functions). Additionally, ES6 also enhances the decomposition of JavaScript programs into modules, fostering a modular approach to code organization and reuse (Resig et al. 2016).

These additional features collectively enhanced the language's capabilities for programming in different paradigms (e.g., object-oriented programming and functional programming) (Wirfs-Brock and Eich 2020; Flanagan 2020), increased the expressivity of the

language, and contributed to improving code readability and the contextual separation of code.

ES6 also introduced the `let` and `const` keywords for block-scoped variables, destructuring assignments for concise variable assignments, improved iterators and generators to facilitate the implementation of control flow, and `Promises`, a mechanism for implementing asynchronous operations (Resig et al. 2016; Flanagan 2020). Indeed, the modifications in this version were so substantial that the term *"Modern JavaScript"* is now commonly associated with code written using ES6 syntax and beyond (Morgan and Stewart 2018).

From 2016 to 2020, ECMAScript versions adopted a yearly release cycle, shifting from numerical versioning (e.g., ES5, ES6) to a naming convention based on the year of publication (e.g., ES2016, ES2017). During this period, ES2016 to ES2020 introduced new language features while refining existing ones, following the standardization efforts of Technical Committee 39 (TC39)—the committee responsible for standardizing the JavaScript programming language. These versions consistently reviewed existing features throughout this period and introduced novel constructs (Wirfs-Brock and Eich 2020).For instance, in 2016, ES2016 introduced (a) the array method includes(), which verifies whether an array contains a given value, and (b) the exponentiation operator (**) for mathematical calculations. In 2017, ES2017 introduced the `async/await` keywords, simplifying asynchronous programming by allowing developers to write asynchronous code in a more readable, synchronous-like style (Flanagan 2020).

The release of ES2018 in 2018 introduced the spread operator for copying the properties of an existing object into a new object, along with the addition of the `for-await-of` loop, designed to work with asynchronous iterators (Flanagan 2020). In 2019, the language standard ES2019 introduced new methods for array manipulation (`Array.prototype`), such as `flat()` and `flatMap()`. Additionally, ES2019 standardized the `JSON.stringify()` behavior for `BigInt` values and refined various syntax and semantic rules (Flanagan 2020). These enhancements further enriched the language with additional functional programming features. In 2020, ES2020 introduced the `BigInt` data type, the `globalThis` object, the `nullish coalescing operator (??)` for providing default values, and `optional chaining (?.)` for safe property access. These new language features aim to simplify common programming patterns and provide a more concise syntax (Flanagan 2020).

ES2021 (ES2021) introduced several enhancements, including numeric separators, which improve program readability by allowing underscore characters within numeric literals. ES2021 also introduced the `replaceAll()` method, providing a direct way to replace all occurrences of a substring in a string. Additionally, ES2021 added the `WeakRef` and `FinalizationRegistry` APIs, enabling the garbage collector to manage weak references and perform cleanup operations more effectively. Furthermore, ES2021 introduced the `Promise.any()` combinator, which resolves as soon as the first promise in an array fulfills, streamlining the handling of multiple asynchronous operations (Kelhini 2021).

ES2022 (ES2022) introduced several syntax and library enhancements to improve code structure and usability. Among the syntax changes, it introduced class fields and private methods, enabling developers to define properties and methods with restricted access directly within class definitions, without relying on constructor-based initialization. Additionally, ES2022 introduced static blocks in classes, allowing static initialization logic to be executed once per class definition. In terms of library updates, ES2022 added the `Array.prototype.at()` method, which provides a more convenient way to access elements from an array using positive or negative indices. Furthermore, it introduced top-level `await` in modules, eliminating the need for wrapping asynchronous logic within an async function at the module level, thereby simplifying asynchronous code execution (Phang 2022).

ES2023 (ES2023) and ES2024 (ES2024) primarily introduced updates to built-in libraries, enhancing functionality and improving language consistency. ES2023 expanded the `at()` method to support Strings and TypedArrays, providing a more intuitive way to access elements using relative indexing. It also introduced the *cause* property for `Error` objects, allowing developers to track error causation chains more effectively. Additionally, ES2023 refined module handling with top-level `await`, enabling asynchronous operations at the module level without requiring an enclosing async function. ES2024 continued this trend of library enhancements by improving standard objects and refining existing methods, reinforcing JavaScript's commitment to usability and robustness (Rauschmayer 2024). Our study focused on ECMAScript versions up to ES2023, as data collection was completed by December 31, 2023.

2.4 JavaScript Cross-Version Compatibility

The issue of JavaScript cross-version compatibility is central to understanding the factors that might hinder developers from rejuvenating JavaScript code. Backward compatibility is a key JavaScript design constraint, ensuring that older code remains functional on newer JavaScript engines (Andreasen et al. 2017).

However, language modifications introduce incompatibilities in some scenarios, particularly when new features repurpose previously undefined keywords. Python addresses this challenge with the concept of *soft keywords*, introduced in Python 3.10 to enable the Structural Pattern Matching feature. While JavaScript does not formally implement soft keywords, it achieves a similar effect through context-sensitive parsing, where certain keywords (e.g., await) are only treated as reserved within specific contexts.

JavaScript runs in a wide range of environments, including application servers, IoT devices, and blockchain applications. Nevertheless, these uses of JavaScript often depend on JavaScript engines originally developed for popular browsers. For example, Node.js uses the V8 engine from Google Chrome. Therefore, understanding how quickly browsers adopt new versions of the JavaScript language also sheds light on JavaScript adoption trends in other scenarios. Major JavaScript engines, such as V8, SpiderMonkey (Firefox), and JavaScriptCore (Safari and Bun), tend to implement features at a similar pace across platforms [1], though modern JavaScript features achieve full compatibility in browsers at varying rates.

For instance, the ES6 version was released in 2015 and achieved broad browsers support by 2017, while features from ES2018 and ES2019 required additional years, reaching full compatibility by 2020. Surprisingly, although ES2021 was officially released in 2021, most browsers offer full support to its features already to 2020.

This apparent discrepancy-where the "Release Year" and "Browsers Full Support" dates do not coincide—reflects the reality of progressive feature integration by browser vendors. In other words, while the formal specification was published later, many engines had already incorporated its features before the official release. Such gradual and sometimes anticipatory adoption underscores the ongoing standardization challenge across browsers, which in turn influences the broader adoption trends of JavaScript in various environments [2].

It is important to highlight that the compatibility challenge varies between domains: browser environments require meticulous handling to accommodate a broad spectrum of user configurations, whereas server-side developers have more control over runtime environ-

[1] https://compat-table.github.io/compat-table/es2016plus/

[2] https://www.w3schools.com/js

ments, enabling them to adopt newer features more aggressively. Additionally, the evolution of JavaScript features is guided by a formal process outlined in the TC39 Process Document [3], which explains the different stages that proposed features must go through. This process is crucial because JavaScript engines only implement features that have reached either the third or the fourth stage, ensuring stability and interoperability across environments.

Transitioning to newer JavaScript versions presents challenges related to code adjustments (Gokhale et al. 2021; Alves et al. 2022). For instance, the migration to ES6 (ECMAScript 2015) turned JavaScript into a multi-paradigm language, introducing classes, arrow functions, and promises. To ensure compatibility with older engines, developers often use transpilers, e.g., Babel (Nicolini et al. 2024), which transform modern code into versions that older engines can execute. However, despite these tools, execution failures may still occur in unsupported environments.

3 Overall Research Design

Our study aims to build an understanding of (a) modern JavaScript features adoption (from ES6 onwards) in legacy open-source projects and (b) the potential benefits and challenges associated with JavaScript source code rejuvenation. Our analysis considers 32 modern JavaScript features (including Object Destructuring, Arrow Function Declarations, and Async Declarations) introduced in ECMAScript versions from 2015 to 2022 that modify the language's syntax. Therefore, we focus on new syntactic elements of the JavaScript language, excluding from our analysis changes to the standard libraries, such as enhancements introduced in the regular expression library of modern JavaScript. Table 1 provides a comprehensive overview of modern JavaScript features included in our study. The "Feature" column lists each specific language capability. The "JavaScript Version" column indicates the ECMAScript version in which each feature was formally introduced. The "Release Year" column shows the official release year of the respective ECMAScript version, offering a temporal context for when these features became part of the standardized language. Lastly, the "Simple Example" column presents a concise code snippet that demonstrates the usage of the feature, making it easier for readers to understand its practical application.

To achieve the goals of our study, we address the following research questions:

(RQ1) To what extent do JavaScript open-source repositories rely on modern features?
(RQ2) When did JavaScript developers start using each one of the modern JavaScript features?
(RQ3) What are the adoption trends of each one of the modern JavaScript features?
(RQ4) What are the main motivations that lead developers to rejuvenate legacy JavaScript code, and what are the main challenges that prevent them from doing so?

Answering the first research question helps us to understand to what extent JavaScript developers *embrace* modern JavaScript features and whether the new releases of the language are meant for the development of JavaScript applications. For example, arrow functions are anticipated to enhance developer expressiveness. Given that JavaScript functions are treated as first-class citizens, it is expected that developers would extensively utilize arrow functions. Conversely, JavaScript classes may not be as appealing since JavaScript already offers encapsulation through prototypes (Haverbeke 2014).

[3] https://tc39.es/process-document/

Table 1 Modern JavaScript Features, ECMAScript Version, Official Release Year, and Simple Usage Examples

Feature	JavaScript Version	Official Release Year	Simple Example
Const Declarations	ES6	2015	`const x = 10;`
Let Declarations	ES6	2015	`let y = 5;`
Object Destructuring	ES6	2015	`const {a, b} = obj;`
Array Destructuring	ES6	2015	`const [a, b] = arr;`
Yield Operators	ES6	2015	`function* gen(){yield 1;}`
Enhanced Property Assignment	ES6	2015	`const obj = {x, y};`
Arrow Function Declarations	ES6	2015	`const f = ()=> {};`
Default Parameters	ES6	2015	`function f(x = 1){}`
Class Declarations	ES6	2015	`class MyClass {}`
Import Statements	ES6	2015	`import {x} from 'y';`
Export Declarations	ES6	2015	`export const x = 1;`
Rest Statements	ES6	2015	`function f(...args){}`
Template String Expressions	ES6	2015	`const s = 'Hello ${n}';`
Function Property Declarations	ES6	2015	`const obj = {f(){}};`
Spread Arguments	ES6	2015	`const arr = [...iterable];`
Computed Property Assignment	ES6	2015	`const obj = {[key]: value};`
For of Statements	ES6	2015	`for (const item of array){}`
Exponentiation Assignments	ES2016	2016	`x **= 2;`
Async Declarations	ES2017	2017	`async function foo(){}`
Await Operators	ES2017	2017	`await asyncFunction();`
For Await Of Statements	ES2018	2018	`for await (const x of xs){}`
Spread in Objects	ES2018	2018	`const obj = {...otherObj};`
Rest in Objects	ES2018	2018	`const {a, ...rest} = obj;`
Optional Catch Binding	ES2019	2019	`try {} catch {}`
Optional Chain	ES2020	2020	`const value = obj?.prop;`
Null Coalesce Operators	ES2020	2020	`const value = a ?? b;`
BigInt	ES2020	2020	`const big = 123n;`
Numeric Separator	ES2021	2021	`const num = 1_000_000;`
Assignment Operators	ES2021	2021	`x ||= y;`
Static Block in Classes	ES2022	2022	`class C { static { }};`
Private Fields	ES2022	2022	`class MyClass { #x; }`
Private Methods	ES2022	2022	`class MyClass { #m(){}};`

Answering the second question helps us understand when developers start to adopt modern JavaScript features. The second question explores how confident developers are to switch from already established constructs and idioms to new ones that might eventually increase productivity, code conciseness, and even lead to security improvements. Exploring the third question helps us evaluate whether there is a trend of increasing adoption of modern JavaScript features in open-source projects.

JavaScript is known for its rapid iteration cycle when developing applications, but the question of how that reflects on maintenance efforts to rejuvenate legacy code is still open. The fourth question focuses on the motivations for JavaScript code rejuvenation and investigates the challenges developers face that prevent them from rejuvenating legacy systems.

3.1 Projects Selection Procedures

Our approach to project selection was inspired by the work of Zhao et al. (2017). In the first step, we searched for GitHub repositories using JavaScript as the main programming language, specifically targeting those created in or before 2012 and active in 2023. We chose projects that were at least three years old by 2015 to focus on potentially mature and legacy systems, as systems that have been operational for at least a couple of years are likely to face challenges related to technological obsolescence and misalignment with current practices (Rosenkranz et al. 2024). This decision allows us to investigate projects that evolved using the "pre-modern" JavaScript language, relying on outdated constructs and idioms, and that might have undergone rejuvenation efforts to remain active until 2023.

In addition, according to Avelino et al. (2019), inactive repositories are less likely to show code evolutions efforts due to the low frequency of updates after the main developers have abandoned them. The study highlights that even popular projects face difficulties in attracting new contributors, with only 41% of projects fully recovering their maintenance activity after the departure of their main developers. This makes it more challenging to identify sustained rejuvenation efforts in abandoned projects. During data collection, we focused on active repositories to ensure that our analysis captures projects likely to be addressing technological debt and aligning with current best practices.

We used the SEART tool (Dabic et al. 2021) to assist in selecting candidate GitHub repositories, initially identifying 726 JavaScript repositories (S1). In the second step, we aimed to filter out non-relevant repositories. Starting with S1, we applied specific criteria: excluding repositories with fewer than 1,000 commits, fewer than 10 contributors, and any forks or repositories labeled as tutorials. This filtering resulted in a refined set of 286 repositories (S2).

In the third step, we computed the number of JavaScript files for each repository in S2 using the CLOC tool (Danial 2021). We then calculated the quartiles (Q1, Q2, Q3) based on the number of JavaScript files. We established lower limits for the number of files to filter out repositories with fewer files than these thresholds. As a result, we excluded small projects, that is, those with fewer JavaScript files than the 25th percentile (Q1). For example, repositories that are not actual software products, like the JavaScript style guide found at https://github.com/airbnb/javascript, were also removed from our analysis. Additionally, 58 repositories were removed due to issues encountered during data collection. Our final selection consists of 158 JavaScript repositories.

3.2 Research Organization

We structured our research into two studies. The first one focused on using descriptive statistics and time series to investigate the adoption (and adoption trends) of modern JavaScript features. The first study addresses the research questions RQ1, RQ2, and RQ3; it primarily relies on mining the source code history of the selected repositories. The second study uses thematic analysis (Clarke and Braun 2017) to highlight the motivations and challenges

associated with JavaScript code rejuvenation. We collected information from code review comments during the second study and addressed our last research question (RQ4).

4 First Study: Prevalence and Adoption Trends of Modern JavaScript Features

4.1 Data Collection and Analysis

We examined the source code history of 158 GitHub legacy projects in our dataset to comprehend the prevalence and adoption trends of modern JavaScript features. We developed our own tool (JavaScriptm) to traverse the source code history of the selected projects and collect usage scenarios for the features we are interested in.

JavaScriptm relies on an existing JavaScript ANTLR grammar (Parr 2013). ANTLR is a parser generator widely used for processing structured text or binary data. We tailored an existing ANTLR grammar that specifies the JavaScript concrete syntax. Using the ANTLR tooling, we generate an abstract syntax tree (AST), a parser, and code for traversing the JavaScript AST using the visitor design pattern. This infrastructure enables the analysis of JavaScript programs by parsing and extracting information on specific features and syntactic elements from the source code. Figure 3 presents the data collection pipeline that we use in our study.

Our pipeline uses JSMiner to analyze the evolution of JavaScript projects from 2012-01-01 to 2023-12-31. We begin by generating a list of JavaScript repositories from GitHub using SEART (Dabic et al. 2021). Then, our pipeline uses a Python script to clone all the selected repositories into a local directory.

For each project, JSMiner iterates through the repository's commit history, capturing revisions at 30-day intervals to reduce the total number of analyzed commits. For each selected revision, JSMiner performs a git checkout to retrieve the repository state for the specific commit hash. It then traverses the JavaScript files to extract metrics related to the usage of modern JavaScript features.

We store the collected results for each project in individual CSV files, which contain the extracted data and their corresponding revision identifiers. Finally, we consolidate the individual CSV files into one, enabling the analysis of the results through Python scripts. This workflow ensures a reproducible approach to understanding the adoption of modern JavaScript features across multiple projects over time.

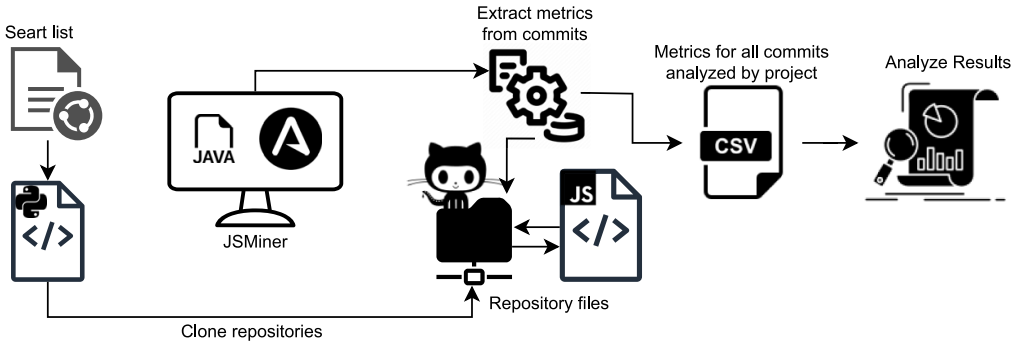


Fig. 3 JSMiner Pipeline for Collecting JavaScript Evolution Data

We use this dataset to answer our research questions RQ1, RQ2, and RQ3. To answer the first, we carried out a descriptive analysis that considers (a) the total usage of features in each repository revision and (b) the median number of feature occurrences to estimate the central tendency across the projects. For the second research question, we identified the first occurrence of all modern JavaScript features in the repositories. For the third research question, we modeled feature adoption using a time series model, assigning time and files with feature usage occurrences as variables. We conducted the data analysis using Python scripts.

Table 2 Summary of JavaScript Features: Total Occurrences, Adoption Rate in Projects, First Occurrence, and Official Release Year

Feature	Total of Occurrences (#)	Projects Adoption (%)	First Occurrence	Official Release Year
Const Declarations	218542	91.13	2012-01	ES6 (2015)
Async Declarations	22905	89.24	2012-01	ES2017 (2017)
Arrow Function Declarations	109874	87.97	2014-04	ES6 (2015)
Let Declarations	63274	82.27	2012-01	ES6 (2015)
Enhanced Property Assignment	40697	79.11	2012-01	ES6 (2015)
Template String Expressions	26778	77.21	2015-06	ES6 (2015)
Object Destructuring	8436	68.98	2012-01	ES6 (2015)
Await Declarations	46373	60.75	2015-09	ES2017 (2017)
Spread Arguments	3458	60.12	2015-07	ES6 (2015)
For of Statments	2923	58.22	2014-04	ES6 (2015)
Default Parameters	5102	56.32	2014-05	ES6 (2015)
Class Declarations	5564	53.79	2014-09	ES6 (2015)
Function Property Declarations	7062	51.89	2015-06	ES6 (2015)
Import Statements	48427	51.89	2015-01	ES6 (2015)
Array Destructuring	1811	46.20	2012-01	ES6 (2015)
Export Declarations	18005	43.67	2015-01	ES6 (2015)
Rest Statements	931	37.34	2015-02	ES6 (2015)
Computed Property Assignment	6782	33.54	2015-07	ES6 (2015)
Optional Chain	2909	29.74	2018-05	ES2020 (2020)
Spread in Objects	522	29.11	2015-08	ES2018 (2018)
Null Coalesce Operators	514	15.18	2020-02	ES2020 (2020)
Optional Catch Biding	78	13.29	2019-04	ES2019 (2019)
Yield Declarations	319	10.75	2012-01	ES6 (2015)
Rest in Objects	52	6.96	2016-01	ES2018 (2018)
Private Fields	1004	5.69	2017-07	ES2022 (2022)
Assignment Operators	149	3.79	2020-10	ES2021 (2021)
BigInt	67	3.79	2020-03	ES2020 (2020)
For Await Of Statments	16	3.79	2018-10	ES2018 (2018)
Numeric Separator	32	3.79	2020-09	ES2021 (2021)
Private Methods	657	3.16	2021-11	ES2022 (2022)
Static block in Classes	5	0.63	2023-06	ES2022 (2022)

4.2 Results of Descriptive Analysis

This section presents the results of a quantitative assessment of modern JavaScript feature adoption. We obtained this data by traversing each project's source code history and counting occurrences of the analyzed features. This process allowed us to extract metrics on feature prevalence across projects and determine the first appearance of modern features in our dataset. The results are summarized in Tables 2 and 3, as well as Fig. 4, which shows the median percentage of files using each modern JavaScript feature across projects.

Table 3 Summary statistics for the occurrence of JavaScript features in GitHub projects

feature	median	mean	std	max	min
Array Destructuring	0	11.46	38.61	340	0
Arrow Function Declarations	121	695.40	1949.18	18569	0
Assignment Operators	0	0.94	7.79	82	0
Async Declarations	23	144.96	392.73	2861	0
Await Declarations	8.5	293.5	1022.1	8443	0
BigInt	0	0.42	3.12	28	0
Class Declarations	1	35.21	119.94	878	0
Computed Property Assignment	0	42.92	447.34	5621	0
Const Declarations	195	1383.18	3224.63	21434	0
Default Parameters	1	32.29	110.89	1159	0
Enhanced Property Assignment	33	257.57	539.81	2953	0
Exponentiation Assignments	0	0	0	0	0
Export Declarations	0	113.95	301.40	2015	0
For Await Of Statments	0	0.10	0.65	6	0
For of Statments	2	18.5	54.65	472	0
Function Property Declarations	1	44.69	149.04	1136	0
Import Statements	4	306.5	737.74	5319	0
Let Declarations	62.5	400.46	824.66	4347	0
Null Coalesce Operators	0	3.25	15.36	137	0
Numeric Separator	0	0.20	1.67	20	0
Object Destructuring	4	53.39	128.28	888	0
Optional Catch Biding	0	0.49	2.77	33	0
Optional Chain	0	18.41	71.23	513	0
Private Fields	0	6.35	42.87	412	0
Private Methods	0	4.15	28.39	255	0
Rest in Objects	0	0.32	2.11	20	0
Rest Statements	0	5.89	28.56	340	0
Spread Arguments	1	21.88	62.74	551	0
Spread in Objects	0	3.30	9.98	69	0
Static block in Classes	0	0.03	0.39	5	0
Template String Expressions	20	169.48	428.71	3478	0
Yield Declarations	0	2.018	17.20	212	0

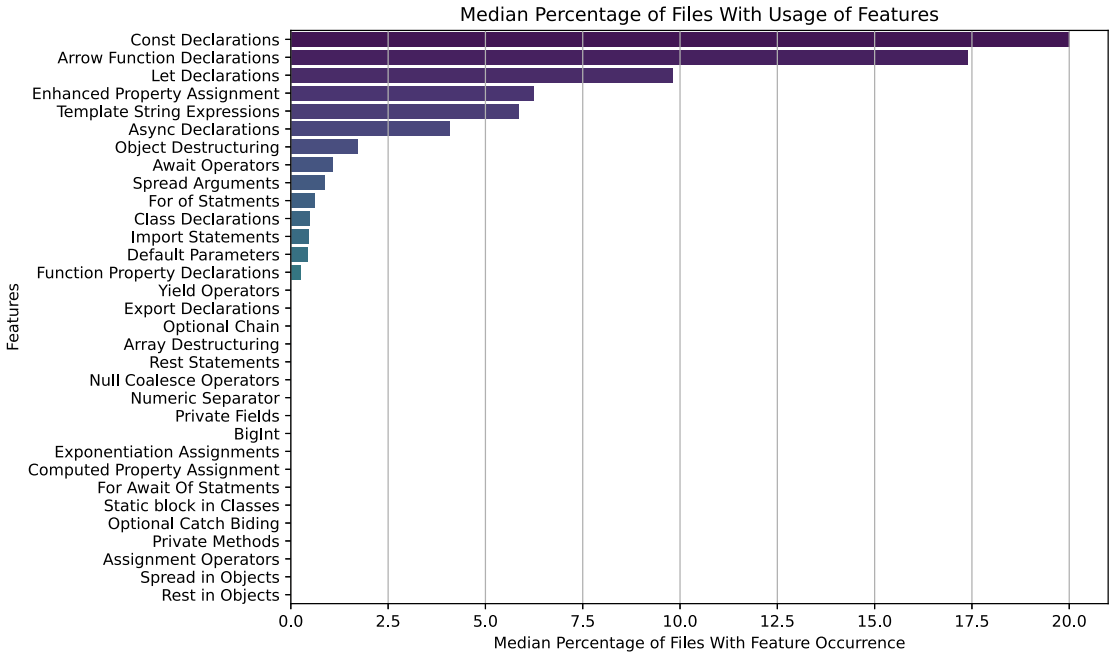


Fig. 4 Modern JavaScript language feature usage by percent of files within projects

The most frequently adopted features in our dataset are Const Declarations, Async Declarations, Arrow Function Declarations, Let Declarations, Enhanced Property Assignment, and Template String Expressions, which are commonly used by more than 75% of projects in our dataset.

Other features that also appear in several projects within our dataset are dataset are Object Destructuring, Spread Arguments, Await Operators, For of Statments, Default Parameters, Class Declarations, Function Property Declarations, and Import Statements, whose prevalence ranges from 51 to 68% across projects. We also found evidence of less frequent feature adoption of Array Destructuring, Export Declarations, Rest Statements, Computed Property Assignment, Optional Chaining, and Spread in Objects ranging from 29 to 46% across the projects.

Still, eleven features do not appear in many projects in our dataset Null Coalesce Operators, Yield Operators, Private Fields, Numeric Separator, BigInt, Optional Catch Biding, Assignment Operators, Rest in Objects, Private Methods, For Await Of Statments, and Static block in Classes. These features are used by less than 15% of the analyzed projects, suggesting a slight interest in their usage. In particular, our dataset shows no evidence of using the feature Exponentiation Assignments.

Table 3 presents summary statistics for the occurrences of various JavaScript features in GitHub projects. Each feature is listed with its median, mean, standard deviation (std), maximum (max), and minimum (min) number of occurrences across the analyzed projects. JavaScript feature usage varies significantly across projects. For instance, the median usage of Arrow Function Declarations is 121 occurrences, while Const Declarations has a median of 195 occurrences. Other features, such as Async Declarations and Let Declarations, have a median of 23 and 62 occurrences, respectively. These variations highlight differing adoption patterns and usage of JavaScript features within our dataset. In particular, the Handsontable project alone accounts for 18569 occurrences of Arrow Function Declarations, which represents 16.9% of the total, and 21434 occurrences of Const Declarations, making up 9.8% of the total. We also observe a similar lack of uniformity in the usage of Let Declarations across projects.

Figure 4 shows the median percentage of files across all projects that adopt modern JavaScript features in our dataset. Considering the latest version of the projects, the median percentage of files that use Const Declarations and Arrow Function Declarations is 20% and 17.39%, respectively. This suggests that these features are commonly adopted in the projects and distributed throughout a significant number of JavaScript modules. Using the median instead of the mean avoids the influence of outliers, ensuring a more reliable representation of typical usage.

Let Declarations appear in 9.80% of the files, suggesting its relevance to solving recurrent concerns within the projects and an increasing preference of software developers to adopt modern variable declarations with the modifiers `let` and `const`. Even modern concurrency features like Enhanced Property Assignment (6.25% of the files), Template String Expressions (5.84% of the files), and Async Declarations (4.08% of the files) show recurrent adoption across project files. Features such as Object Destructuring (1.70% of the files) and (1.08% Await Operators of the files) appear less frequently across JavaScript project files, indicating a smaller prevalence of classical object-oriented patterns. Some features seem to address less recurrent concerns, such as Spread Arguments, For of Statments, Class Declarations, Import Statements, Default Parameters, and Function Property Declarations, which appear in less than 1% of the projects files. This suggests that developers might either be unfamiliar with them or use these features for highly specific problems.

For RQ1, our findings show that modern JavaScript features are used unevenly across projects. Features like Const Declarations, Arrow Function Declarations, Let Declarations, Async Declarations, Enhanced Property Assignment, and Template String Expressions appear in over 75% of projects, with Const Declarations and Arrow Function Declarations found in 20% and 17.39% of files, respectively, whereas features such as Object Destructuring, Await Operators, and Computed Property Assignment appear in less than 5% of files, and Class Declarations, Spread Arguments, and Import Statements in under 1%.

Our dataset reveals that some modern JavaScript features were adopted well before their official language introduction in specific ECMAScript versions, as shown in Table 2. Early-adopted features include Const Declarations, Async Declarations, Let Declarations, Object Destructuring, Array Destructuring, Yield Operators, and Enhanced Property Assignment, all of which began seeing use in January 2012. These features were officially introduced with ES6 in 2015 and ES2017 in 2017. Features like Arrow Function Declarations, For of Statments, Default Parameters, and Class Declarations saw adoption in April 2014, May 2014, and September 2014, respectively, also preceding the ES6 release. Additionally, Await Operators, Private Fields, Spread in Objects, Rest in Objects, and Optional Chaining saw adoption in September 2015, July 2017, August 2015, January 2016, and May 2018, preceding their release with ES2017 in 2017, ES2022 in 2022, and ES2020 in 2020, respectively.

This early adoption pattern indicates a proactive approach by developers to integrate emerging JavaScript features before their formal standardization. The same was also observed for Java (Dyer et al. 2014a). The majority of these features were supported by browsers before the release of ES6 and ES2018. For instance, on June 25, 2013, Arrow Function Declarations

was released on Firefox and Firefox for Android [4]. Other features, such as Const Declarations, were added to Safari in 2011 and Opera in 2006 [5].

In contrast, some features were adopted around their official release dates or shortly thereafter. These include Import Statements, Export Declarations, Rest Statements, Template String Expressions, Function Property Declarations, Spread Arguments, Computed Property Assignment, Null Coalesce Operators, BigInt, Numeric Separator, For Await Of Statments, Assignment Operators, Private Methods, Static block in Classes and Optional Catch Biding.

For *RQ2*, the data indicates early adoption by developers. For instance, Const Declarations, Async Declarations, Let Declarations, and Object Destructuring were used as early as 2012, before their official release in ES6 and ES8. Likewise, Arrow Function Declarations and Class Declarations were adopted before the ES6 release in 2015. This pattern suggests that developers actively explore and incorporate emerging features as browser support becomes available, demonstrating a proactive approach to modernization.

4.3 Trends of Features Adoption

In this section, we analyze the adoption trends of various modern features in a software development context. We employed statistical techniques to evaluate the stationarity and trend direction of time series data associated with each feature. The dataset comprises monthly counts of unique files containing occurrences of feature usage, extracted from software development repositories. The features analyzed include Const Declarations, Async Declarations, Arrow Function Declarations, Let Declarations, Enhanced Property Assignment, Template String Expressions, Object Destructuring, Await Operators, Spread Arguments, For of Statments, Default Parameters, Class Declarations, Function Property Declarations, Import Statements, Array Destructuring, Export Declarations, Rest Statements, Computed Property Assignment, Optional Chaining, Spread in Objects, Null Coalesce Operators, Optional Catch Biding, and Yield Operators, which are found in at least 10% of the programs analyzed in our dataset, as shown in Table 2 in Section 4.2.

We performed an Augmented Dickey-Fuller (ADF) test to assess stationarity (Vishwas and Patel 2020; Huang and Petukhina 2022). If the p-value of the ADF test was less than or equal to 0.05, the series was deemed stationary, indicating a rejection of the null hypothesis. Conversely, if the p-value exceeded 0.05, the series was considered non-stationary.

For instance, Const Declarations, which began its uptake in 2017, closely followed its introduction in ES6 (2015). ES6 introduced modern features like Let Declarations, Object Destructuring, Arrow Function Declarations, Default Parameters, For of Statments, Spread in Objects and Template String Expressions, which developers gradually integrated into their coding practices between 2016 and 2017 after the initial release. Similarly, features such as Class Declarations, Import Statements, and Export Declarations saw increased adoption starting around 2017, aligning with their inclusion in ES6. Figure 5 present the time series for

[4] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions#browser_compatibility

[5] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/const#browser_compatibility

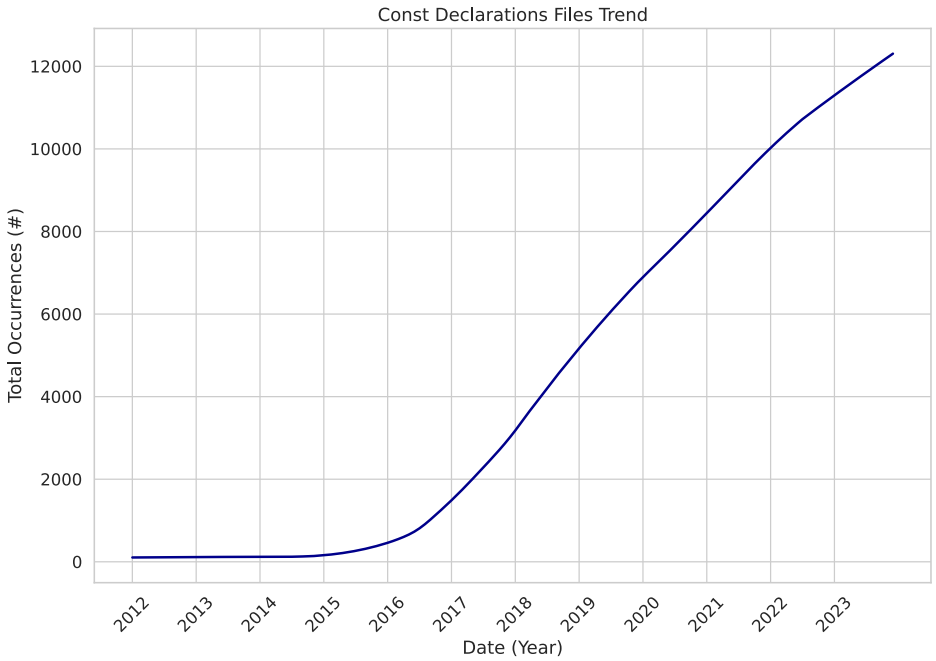


Fig. 5 Initial growth in the adoption of Const Declarations in 2017 after the release in ES6 (2015)

Const Declarations and Fig. 6 present the data from Let Declarations, Object Destructuring, Arrow Function Declarations, Default Parameters, and Template String Expressions.

Features like Rest Statements and Function Property Declarations began gaining traction earlier, dating back to 2015, reflecting developers' early recognition and utilization of these modern features shortly after their introduction in ES6. In contrast, features introduced in subsequent ECMAScript versions, such as Await Operators (ES2017, 2017) and Optional Chaining and Null Coalesce Operators (ES2020, 2020), show a delay in initial growth adoption compared to their release dates, with noticeable upticks in usage appearing around 2020 and 2021, respectively.

This trend might underscore a typical pattern in programming language adoption. Developers often wait for features to stabilize, tooling support to mature, and community best practices to emerge before incorporating new language constructs extensively into their projects. A study of KDE developers (Lucas et al. 2023) reveals a similar pattern: although

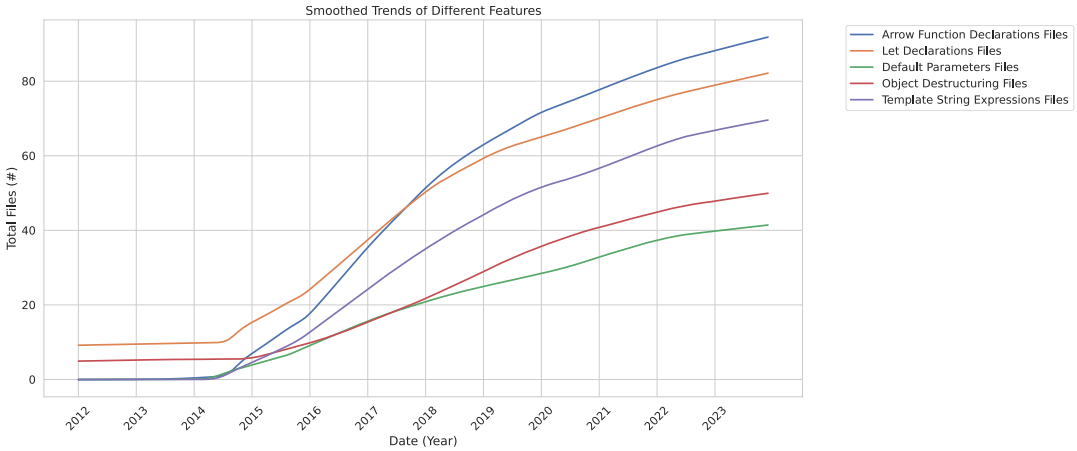


Fig. 6 Initial growth in the adoption of Declarations, Object Destructuring, Arrow Function Declarations, Default Parameters, and Template String Expressions between 2016 and 2017 after the release in ES6 (2015)

developers started using *auto-typed declarations, lambda expressions, and range-based for loops* before 2011, significant growth in their usage was observed only after 2016, five years after the release of the C++11 specification. This delay highlights the necessity for a maturation period before growth adoption can occur. This suggests a broader trend where the adoption of new language features is not immediate. Instead, it requires time for the development ecosystem to fully integrate modern features, during which developers adapt to new paradigms, update their codebases, and establish new best practices. Consequently, the pattern seen in JavaScript's adoption of modern features like Const Declarations and Arrow Function Declarations aligns with this broader phenomenon observed across different programming languages and development communities.

For *RQ3*, the adoption of modern features evolves over time. Some features were quickly embraced after their ES6 release, while others, such as Await Operators and Optional Chaining, only increased in use after 2020, suggesting a gradual integration as tooling and best practices mature.

5 Second Study: Motivations and Challenges for Rejuvenating JavaScript Code

5.1 Data Collection and Analysis

To understand the motivations and challenges that developers face when rejuvenating JavaScript code, and to answer our fourth research question, we manually analyzed a subset of code review comments from the repositories analyzed in our first study. In the scope of this research, a code review consists of evaluating merge requests (also known as pull requests) to identify potential issues with the code, such as defects, security vulnerabilities, and poor design choices (Oliveira 2021; Bogachenkova et al. 2022; Lin and Thongtanunam 2023).

Currently, this is a common practice in collaborative development, where a developer submits changes to the central repository and asks other developers to review those changes, providing feedback and improvement recommendations. The interaction between contributors (developers proposing changes) and reviewers continues until the changes are considered ready to be integrated. Similar to other software development platforms (such as GitLab and BitBucket), GitHub facilitates the code review process through pull requests, where reviewers can attach general feedback by commenting on specific parts of the modified source code. These comments are important for highlighting specific and fine-grained concerns of reviewers.

We developed a Python infrastructure that leverages the GitHub API to scrape comments from pull requests, automating the data collection process. This infrastructure iterates over each pull request in the repositories in our dataset and retrieves the related comments, extracting relevant information (such as user login names, URLs, timestamps, and message bodies) from the response data. Subsequently, it locates the corresponding repository in the database and stores the pull request comments, associating them with the respective pull request IDs.

In total, we mined 122,248 pull request comments and populated a database for analysis. We then executed a textual query to identify potential pull request comments related to source code rejuvenation. This query searched for mentions of modern JavaScript features (e.g., *"arrow function"*, *"let declaration"*, *"const declaration"*) and specific ECMAScript versions (e.g., *"ES6"*, *"ECMAScript 2015"*). Additionally, the query included variations and related terms, such as *"arrow function implementations"*, *"async declarations"*, *"template string usage"*, and *"numeric separator inclusion"* to capture a broader range of relevant discussions. This approach resulted in 447 pull request comments, which were manually analyzed to determine their relevance to rejuvenation efforts and feature adoption.

Regarding data analysis, we employ a thematic analysis (Clarke and Braun 2017) approach to extract **motivations** and **challenges** for rejuvenating JavaScript code from code review comments. We leverage Saldaña's recommendations for data coding (Saldana 2012). Accordingly, we start with open coding, which refers to the initial step of analyzing qualitative data by selecting and associating relevant segments of text to generate labels or tags that directly correspond to concepts and categories derived from the data. We then proceed to classification, wherein we systematically arrange the original codes into distinct categories based on their shared qualities. During this process, we consistently improved and reevaluated codes and categories as new information and insights emerged.

This data analysis procedure ran in two cycles to ensure the relevance and accuracy of the selected pull request comments. In the first cycle, one author of this paper reviewed the 447 code review comments identified in the textual query, separating those relevant to rejuvenation efforts from those merely discussing the use of a modern JavaScript feature. From this process, 93 comments were selected as potentially relevant to rejuvenation. In the second cycle, three authors of this paper independently evaluated a subset of 31 comments. They independently analyzed the comment and the surrounding pull request discussion to verify whether it represented efforts to rejuvenate existing code to adopt modern features. After coding these comments independently, the other two authors reviewed each other's work, indicating agreement or disagreement with the coding. For comments where disagreements arose, all three authors discussed the coding until a consensus was reached.

Finally, the first author of this paper categorized the codes into thematic groups, resulting in three primary categories for JavaScript source code rejuvenation: Motivations (Section 5.2), Challenges (Section 5.3), and Transpiler-based approach (Section 5.4). These categories are detailed in the remainder of this section. Furthermore, during the assessment, we identified several contributions to the repositories that significantly rejuvenate legacy JavaScript code (e.g., one contribution replaces dozens of old-style constructs and idioms with arrow functions). In Section 5.5, we discuss some of these *"rejuvenation efforts"*.

5.2 Motivations to Source Code Rejuvenation

This category presents the motivations that lead JavaScript developers to make efforts to rejuvenate the source code of their applications-that is, replace legacy constructs and idioms with new features of a programming language. According to our findings, ***improving code comprehension*** is the most frequently mentioned by developers during the code review process in our study that leads JavaScript developers to rejuvenate legacy source code (with 10 occurrences). Code comprehension is the process through which developers actively acquire knowledge about a software system by exploring and studying various software artifacts. This includes reading and understanding the source code and related documentation (Daka et al. 2015). Different coding aspects, such as programming constructs, naming conventions,

travis-web — Jun 17, 2016

> If you wanted to get more ES2015 and Ember-pro with this you could shorten to this:
>
> – `return !cJobs.any(c => branch.get('name')=== c.get('branch.name'));`
>
> Concise and elegant!

Fig. 7 Suggestion to adopt Arrow Function Declarations to improve code comprehension

and formatting, influence how easily developers can comprehend code (Oliveira et al. 2020). We considered here aspects like *"conciseness"* and *"clearness"* as code comprehension.

For instance, in the code review comment of Fig. 7, a developer suggests that using an Arrow Function Declarations would make the code more concise and elegant. The motivation behind the comment suggestion is to encourage the adoption of ES6 features, specifically Arrow Function Declarations, to make the code more concise. By replacing the traditional `anonymous function` with an Arrow Function Declarations, the Arrow Function Declarations syntax simplifies the code by eliminating the need for the `function` keyword and reducing the overall verbosity of the code. According to Rauschmayer (2015), one of Arrow Function Declarations goals is to shorten syntactical form.

Another benefit of the adoption of Arrow Function Declarations that improves code comprehension is that it binds the `this` keyword to the surrounding lexical context. According to Rauschmayer (2015), the developers can eliminate the need for the statement `self = this` to preserve the correct `this` context within nested functions. This happens because the use of Arrow Function Declarations eliminates the need for manual binding of `this`, ensuring that the correct context is preserved. This makes the code more concise and reduces the risk of errors due to incorrect `this` context. Also, developers recommend adopting Async Declarations, Await Operators, and Object Destructuring to make the code more clear and concise.

One developer suggested a source code rejuvenation (in Fig. 8) to use asynchronous programming and object destructuring. By employing Async Declarations and Await Operators, the code now handles the promise returned by readArmored more cleanly, ensuring that this block of code is not executed before the promise is resolved or rejected. Additionally, Object Destructuring simplifies the extraction of the first key from the keys array, resulting in more

```
it('require_uid_self_cert=true: cannot use pubkey w/o self certs', async
     function() {
- let k = openpgp.key.readArmored(no_self_cert one_user_no_self_cert).keys[0];
+ let {keys: [k]} = await
     openpgp.key.readArmored(no_self_cert.one_user_no_self_cert);
  ...
}
```

Fig. 8 Async/await and destructuring for cleaner promise handling and key extraction from project *openpgpjs* in the pull-request #753

Fig. 9 Avoiding source code rejuvenation due to incompatibility issues with old browsers

concise code. According to the proposal by tc39 [6], one of the goals of async and await features is to reduce the promise's boilerplate code and make syntax concise.

5.3 Challenges to Source Code Rejuvenation

This category covers the challenges that hinder JavaScript developers from rejuvenating the source code of their applications. ***Incompatibility*** is the most frequent challenge in our assessment, appearing a total of 25 times. For instance, developers emphasize possible incompatibility issues between modern JavaScript features as older browsers and dependencies, which often prevent them from using modern JavaScript. In Section 2.4 we presented a brief discussion about about JavaScript Cross-Version Compatibility.

In the code review comments in Figs. 9 and 10, developers discuss incompatibility issues between modern JavaScript and browsers like PaleMoon, Chrome versions prior to 41, IE11, and older versions of Safari.

We also found code review comments where developers report compatibility issues between modern features and JavaScript libraries, platforms, and tools-such as UglifyJS and PhantomJS. See code review comments in Figs. 11 and 12. UglifyJS [7] is a JavaScript tool used to minify and compress JavaScript code in order to improve processing time by removing unnecessary characters, optimizing code structure, and reducing file size. PhantomJS [8] is a headless web browser, meaning it operates without a graphical interface. It is used primarily for automating web page interactions, testing, and scripting web pages, allowing developers to run and test their code in a controlled environment.

Also, developers report incompatibility issues between modern JavaScript and runtime engines (see the code review in Fig. 13). For instance, versions of Node.js up to and including v0.12, as well as the JavaScript Nashorn [9], do not support modern JavaScript. Nashorn was embedded in versions 8 – 15 of the Java Development Kit. Legacy versions of these engines provide limited or nonexistent support for the new features introduced in ES6. Nonetheless, the projects we investigate often need to maintain compatibility with these legacy versions.

Code conventions might prevent the adoption of modern JS features is the second most common challenge in our assessment, with 4 occurrences. In this case, developers report that they prefer to maintain consistency with existing code conventions rather than adopt modern features. For instance, the code review comment in Fig. 14 shows that a specific developer prefers to avoid using modern JavaScript features to maintain consistency with the legacy code. This resistance to change can hinder the adoption of new features, potentially limiting the benefits of source code rejuvenation.

[6] https://tc39.es/proposal-async-await/#intro

[7] UglifyJS: https://www.npmjs.com/package/uglify-js

[8] PhantomJS: https://github.com/ariya/phantomjs

[9] JEP 372: https://openjdk.org/jeps/372

sinon — Sep 17, 2018

> Sinon is an ES5.1 project. If we allow ES6 syntax, we will drop support for IE11 and older Safari. I don't think we're quite there yet.

Fig. 10 Avoiding source code rejuvenation due to incompatibility issues with old browsers

id — Oct 29, 2021

> I think it was just that UglifyJS never got support for full ES6 syntax. so people use other tools to minify JS nowadays.

Fig. 11 Avoiding source code rejuvenation due to incompatibility issues with dependencies

qunit — Jul 25, 2017

> Should be a normal function as 'PhantomJS' doesn't support arrow functions. The same goes for the other test.

Fig. 12 Avoiding source code rejuvenation due to incompatibility issues with platforms

tiddlywiki5 — Sep 18, 2021

> I'm afraid we can't use arrow functions in the core. We stick to ES5.1 (roughly) because it's important to retain compatibility with older JS runtime engines.

Fig. 13 Avoiding source code rejuvenation due to incompatibility issues with JS runtime engines

cordova-android — Mar 25, 2021

> Well I love ES6 (and TypeScripe [sic]) but when I always try to follow the pattern of existing code and found that existing code is still using `var` mostly, and snake case variables, my decision is to continue using `var` (instead of more recent constructs such as `let`). Thanks anyways. Updated this for new code.

Fig. 14 Developers prefer to maintain consistency with the existing code instead adopt modern features

webpack — Jan 24, 2017

> Contributor 1: I checked the speed and I got bad results for Map.
>
> Map x 417 ops/sec ±1.76% (82 runs sampled) Native x 8,021 ops/sec ±1.69% (84 runs sampled)
>
> https://gist.github.com/Tom910/8642ee4d9a4913cd40d1581ab3d95738
>
> For this reason, it is better not to rewrite Map. node - 7.2.1
>
> Contributor 2: Newer features are gonna be a bit slower. let, const, class, arrow functions, all of these have performance penalties. There are issues on it, they're working on, but they're slower right now.

Fig. 15 Modern features might reduce the program performance

Code review comments also suggest that ***modern features might reduce the program's performance***. In the code review comment in Fig. 15, a contributor states that newer features like `Map`, Let Declarations, and Const Declarations might compromise system performance. The full discussion in this pull request highlights a performance issue with using modern features in certain contexts, where slower performance was observed compared to "legacy" alternatives. As a result, there's a tradeoff decision about whether to use a `Map` or stick with the legacy approach. The suggestion to add a `TODO` to consider `Map` when the performance improves indicates a willingness to revisit the issue in the future.

Additionally, there's consideration of optimizing code for performance on Long Term Support (LTS) versions of Node.js versus staying with the latest versions, with a leaning towards staying up-to-date and incorporating updates into the build performance guide. This code review underscores the importance of balancing the benefits of new features with performance considerations when making decisions about source code rejuvenation.

We also found recommendations to ***avoid using features that were recently introduced***. Our analysis and the previous results outlined in this section suggest that the reluctance to adopt recent features (such as Optional Chaining) may stem from its novelty and the lack of full support across all execution environments. Therefore, developers might prefer to avoid modern features like Optional Chaining and opt for more traditional methods of verifying nested properties to ensure compatibility with various execution environments. Code review comments also suggest that modern features might be more complex for less experienced JavaScript developers. Introducing new syntactic constructs, such as Arrow Function Declarations, could potentially raise the learning curve, making it harder to understand and work with the code, especially for new contributors or those who don't have JavaScript as their primary programming language. Although the reviewer comment in Fig. 16 acknowledges the benefits of Arrow Function Declarations, it highlights that these benefits do not outweigh the disadvantages and additional complexity introduced. The same comment highlights that development tools with built-in shortcuts can prevent developers from adopting new syntax simply because these tools make it easier to write code using the old syntax (see Fig. 16). For instance, an editor shortcut that quickly generates an empty function template might discourage the use of Arrow Function Declarations, even though the latter would make the code more concise. This reliance on shortcuts for older syntax can hinder efforts to rejuvenate the codebase, as developers might default to using the more familiar and easily accessible patterns provided by their tools.

video.js — Apr 11, 2015

> I don't know how I missed these but they're the same as the other example we reverted. I looked at how babel translates arrow functions and it at least doesn't do an extra 'fn.bind()' so that's good. I want to make sure I'm not just being a luddite here so if there's fault in my reasoning let me know. I think it's an important issue. I still don't think the shortened syntax alone is benefit enough to warrant upping the javascript reading level for contributors. For most contributors JavaScript is probably still not their primary programming lang and we'd be replacing one of the most most well known keywords with one of the least known that does the exact same thing (in this specific case). Also Babel makes named functions when translating arrow functions. That goes against the "Arrow functions are always anonymous" rule and could potentially [create issues in IE8](http://kiro.me/blog/nfe_dilemma.html). Though I'm not sure how likely that is and I wish we could use named functions more for stack traces. Also if you have a text editor with a function shortcut there's not much benefit left over for this specific case. 'f' then 'tab' quickly in Atom makes an empty function. I do think there are some cool examples of using arrow functions like: (...) I would have a harder time arguing against that and it even seems more readable than the example above for some reason. And using it for it's 'this' binding. (...) But in all the examples here we seem to just be using it for it's shorter anonymous function syntax which if that's correct then for consistency there's over 500 other anonymous functions in the codebase we would need to update too. Am I missing another benefit in these examples?

Fig. 16 Examples of challenges in adopting modern features. First the comment illustrates the reluctance to use Arrow Function Declarations due to the potential increased complexity for contributors unfamiliar with JavaScript. Second, demonstrates how development tools with built-in shortcuts for older syntax can discourage the use of newer, more concise syntax, thus hindering source code rejuvenation efforts

5.4 Transpiler-Based Approach to Rejuvenate JavaScript Code

This category summarizes some of the code review assessment findings related to ***the adoption of a transpiler (e.g., Babel) to support developers*** in using modern JavaScript features

novnc — Jan 30, 2018

> Contributor: Yup. Declaring variables as touchButton = 1 as you can do with methods is ES7 so it would need transpilation to all browsers. So the way to declare value properties in ES6 is to declare them like this in the constructor. I've put them always at the bottom so it should be trivial to take them out once ES7 is more widespread.

> Member: Urgh. Are there any other patterns that can be used? I'd like to avoid the confusion of putting it in the constructor with other stuff. And for RFB it gets extra messy as it has getters and setters for some properties but not all.

> Contributor: I think that we could use https://babeljs.io/docs/plugins/transform-class-properties/ However we would need to transpile it for all browsers AFAIK. It's not even part of es2017 preset.

> Member: That's a no-go in that case. We need something that everyone (except IE) supports.

> Contributor: Agreed. That's why I opted for adding them at the bottom of the constructor with a comment stating that they are properties. Once the class properties become a standard, it's trivial to move them out of there.. . .

Fig. 17 Suggestion to use Babel transpiler to maintain compatibility with old browsers

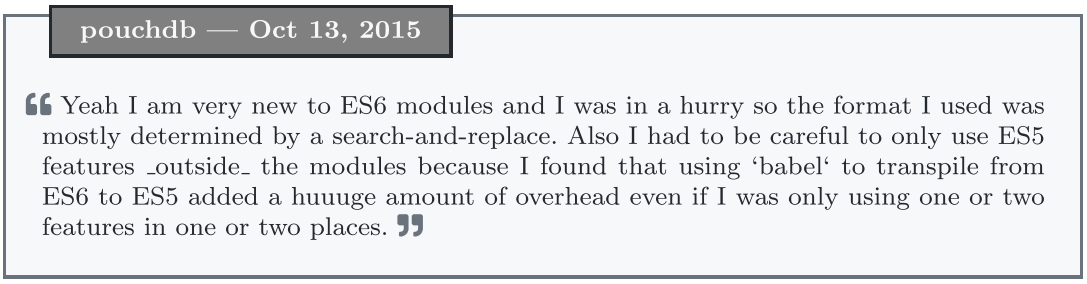


Fig. 18 Babel's transpilation might increase overhead

in legacy code. Indeed, we found code review comments suggesting that developers transpile modern JavaScript features using Babel (see Fig. 17). Transpiling modern JavaScript code can increase the compatibility of the code with legacy JS engines, addressing challenges related to their differing support for new syntax features. As such, developers can write code using the latest syntaxes, and transpile it to widely supported legacy constructs and idioms. This approach allows for the immediate use of advanced features without waiting for general support in JS engine implementations, facilitating the source code rejuvenation process.

However, as developers noted, the use of a transpiler might introduce complexity in the build process and potential confusion in code organization, particularly when mixing properties in constructors with other initialization code. Another side effect is that ***using a transpiler might introduce bugs***. The code review comment in Fig. 16 highlights that Babel's transpilation of Arrow Function Declarations into named functions can cause compatibility issues with older browsers. Additionally, issues have been reported in the Babel repository regarding incorrect transpilation of arrow functions, as seen in GitHub issues Issue #1742, Issue #1453, and Issue #1887. These issues illustrate how the use of transpilers can inadvertently introduce new bugs, complicating efforts to rejuvenate the codebase.

We also found code review comments reporting that ***the Babel transpiler might significantly increase the overhead of the resulting code***. In the comment in Fig. 18, a code reviewer explains that they used ES5 features outside the modules because of the significant overhead added by Babel when transpiling from ES6 to ES5. Even if only a small number of ES6 features were used, the resulting transpiled code might degrade the performance of the scripts. This is due to the additional code Babel includes to ensure compatibility with older browsers, leading to a much larger final bundle that potentially impact performance and load times.

For *RQ4*, our findings indicate that enhancing readability, conciseness, and maintainability drives the adoption of modern JavaScript features. Yet, challenges like compatibility, resistance to change, and performance concerns can delay their uptake. Tools such as Babel help introduce modern syntax in older environments, though they also add complexity and potential risks. This highlights the need for a balanced strategy that aligns modernization efforts with project constraints and ecosystem stability.

Commit 1 *(project `jasmine`)*

- *Hash: 1166d10e4*
- *Date: April 16, 2022*
- *Message: Use const/let in specs, not var*

Fig. 19 Rejuvenation effort commit from project `jasmine`

```
it('falls back to setTimeout', function() {
- var setTimeout = jasmine.createSpy('setTimeout').and.callFake(function(fn) {
- fn();
- }),
+ const setTimeout = jasmine
+ .createSpy('setTimeout')
+ .and.callFake(function(fn) {
+ fn();
+ }),
      global = { setTimeout: setTimeout },
- clearStack = jasmineUnderTest.getClearStack(global),
- called = false;
+ clearStack = jasmineUnderTest.getClearStack(global);
+ let called = false;
...
```

Fig. 20 Example of introducing Const Declarations and Let Declarations in a large commit from `jasmine`

5.5 Rejuvenation Efforts

During our code review assessment, we identified 31 commits aimed at rejuvenating JavaScript code (referred to as Rejuvenation Efforts Commits). Although this finding does not directly address any research question, we believe it is still useful to characterize JavaScript rejuvenation efforts in this paper. These 31 commits come from 19 distinct projects in our dataset, representing 12% of the 158 projects. They target 12 modern JavaScript features in total. Table 5 shows the number of features adopted in the projects by source code rejuvenation with corresponding commit. In the rest of this section, we will describe some of the source code rejuvenation efforts.

For instance, a large effort to adopt modern JavaScript in commit (Fig. 19) introduces more than **2038 Const Declarations** and **152 Let Declarations** to replace *Var Declarations*. The listing in Fig. 20 is also part of a test case from *Jasmine* project, focusing on a scenario where a fallback to `setTimeout` occurs. The rejuvenation replaces the traditional variable declaration (`var`) with the modern Const Declarations keyword for the `setTimeout` function. Additionally, the code updates the assignment of the variables `clearStack` and `called` using the Let Declarations keyword for second variable. These changes are part of a large effort to adopt more modern JavaScript syntax and follow some best practices (Haverbeke 2014; Cantelon et al. 2013). The functionality of the test case, which checks the fallback behavior to `setTimeout`, remains unchanged.

Several projects have adopted Arrow Function Declarations to rejuvenate their codebase. For instance, the commit depicted in Fig. 21 introduces **92 Arrow Function Declarations**. This commit is particularly interesting because it also introduces Await Operators, Async Declarations, and Object Destructuring. This specific change replaces a traditional JavaScript function declaration with the more concise Arrow Function Declarations syntax (Haverbeke

Commit 2 *(project `highlight.js`)*

- *Hash: a69eb6a2e*
- *Date: May 26, 2019*
- *Message: Switch to fs.promises vs bluebird for tests (#2226) - also switch to more idiomatic JS using the newer syntax*

Fig. 21 Rejuvenation effort commit from project `highlight.js`

Commit 3 *(project noVNC)*

- *Hash: 651c23ece*
- *Date: Jul 12, 2018*
- *Message: Use fat Arrow Function Declarations* `const foo = () => ... ;` *for callbacks and any other function that is passed around and it's not a top level function.*

Fig. 22 Rejuvenation effort commit from project `noVNC`

2014) and revealing the developer interest in rejuvenating test code. Commit in the Fig. 22 introduces **111 Arrow Function Declarations** in a big effort to rejuvenate their code.

Finally, commit in Fig. 23 introduces the Import Statements syntax in a JavaScript module. Specifically, the transformation replaced the CommonJS `require` statements with ES6 Import Statements. The transformation from CommonJS to ES6 module syntax in the PouchDB project providing a more concise and clear structure for dependencies. Finally, ES6 modules support tree shaking, which can reduce bundle sizes and improve performance by eliminating dead code. The listing in Fig. 24 present one of the transformations in this commit.

6 Discussion

In this section, we discuss the implications of our findings (Section 6.1) and the limitations that may threaten the validity of our research (Section 6.2).

6.1 Implications of our Findings

We performed an extensive analysis of 158 JavaScript programs from GitHub. We assessed the adoption of modern JavaScript features, such as Const Declarations, Async Declarations, Arrow Function Declarations, Let Declarations. Additionally, we manually analyzed a subset of code review comments from the repositories used in our dataset. Within this part, we provide a concise overview of the implications of our research on some topics:

The results for our first research question (RQ1) suggests that the adoption of modern JavaScript features varies significantly across open-source projects. Widely used features like Const Declarations, Async Declarations/Await Operators, and Arrow Function Declarations are prevalent, while others like Null Coalesce Operators and BigInt are less commonly integrated. This suggests that developers prioritize features that offer immediate benefits in code quality and maintainability, while more specialized features see selective use.

The findings from RQ2 and RQ3 reveal a nuanced pattern in adopting modern JavaScript features. On the one hand, RQ2 highlights that developers often adopt new features before they are officially standardized, using this early period to test and understand their potential. This

Commit 4 *(project PouchDB)*

- *Hash: 99c24a58*
- *Date: Jan 1, 2016*
- *Message: Migrate to ES6/Rollup, build one index.js.*

Fig. 23 Rejuvenation effort commit from project `PouchDB`

```
- 'use strict';
-
- var createBlob = require('../../deps/binary/blob');
- var Promise = require('../../deps/promise');
- var idbConstants = require('./constants');
- var DETECT_BLOB_SUPPORT_STORE = idbConstants.DETECT_BLOB_SUPPORT_STORE;
+ import createBlob from '../../deps/binary/blob';
+ import Promise from '../../deps/promise';
+ import { DETECT_BLOB_SUPPORT_STORE } from './constants';
...
```

Fig. 24 Refactoring was to rejuvenate the source code by adopting ES6 module syntax from `pouchdb`

proactive approach allows developers to experiment with cutting-edge tools and integrate them into their projects even before the browser's full support is available. However, the broader trends identified in RQ3 suggest that widespread adoption typically follows a more measured approach. Developers tend to wait for features to stabilize, for tooling to mature, and for best practices to emerge before fully embracing these new modern features across their projects. For the effective integration of new language features into development practices, this maturation period is crucial, balancing the benefits of innovation with the reliability required for long-term project success. This pattern is not unique to JavaScript but is also observed in the adoption of new features in other programming languages (Lucas et al. 2023).

For the implications from RQ4, our results suggest that while modern JavaScript features improve code comprehension and maintainability, they also introduce challenges, particularly regarding compatibility with older environments. Tools like Babel help bridge these gaps, but they can also add complexity and potential bugs, requiring developers to balance the benefits of modern features with the practicalities of legacy support. Our findings offer practical examples that can assist software developers in rejuvenation the source code of their programs, along with insights for tool developers to identify opportunities for suggesting the adoption of modern JavaScript features.

6.2 Threats to Validity

A potential threat to the internal validity of our study arises from the influence of automated tools, such as transpilers and refactoring tools, on the observed adoption of modern JavaScript features. For example, the Handsontable project alone accounts for 18,569 occurrences of Arrow Function Declarations (16.9% of the total) and 21,434 occurrences of Const Declarations (9.8% of the total). Such outliers may indicate tool-driven transformations rather than adoption by developer actions. However, our study does not include an in-depth evaluation to distinguish whether the usage of these features resulted from automated rejuvenation efforts or deliberate developer actions. While this limitation prevents us from asserting "manual rejuvenation efforts," this limitation does not compromise the validity of our findings.

For external validity, our analysis involved 158 out of 270849 JavaScript projects (0.05%) sourced from the GitHub community. These selected projects met specific criteria, including a development history of more than ten years and recent contributions (after January 1, 2023). It is important to acknowledge that our dataset represents only a small fraction of the total number of applications, limiting the generalization of our findings. Additionally, since our study focused solely on open-source repositories, we cannot generalize our results to industry projects.

Furthermore, the difference in the distributions of feature adoption based on our current dataset (with extremely small projects removed and without our filtering criteria) reveals

an important limitation in generalizing our findings to smaller projects. When considering projects with less than 25% of the JavaScript files, we identified repositories that have lower adoption of certain features compared to larger projects. For example, in Table 4, the adoption of Const Declarations was 71.52% among projects, while in the table with removal of smaller projects, this adoption rose to 91.14%. Similarly, adoption of Arrow Function Declarations increased from 67.09% to 87.97%, reflecting the higher prevalence of more modern features in projects with larger amounts of JavaScript files. This phenomenon indicates that the results of feature adoption are not evenly distributed, with an indication that modern features are used in a reduced number of smaller projects. As a consequence, we cannot generalize our conclusions to smaller-scale projects, as the characteristics of larger repositories significantly influence the prevalence of these technologies.

Table 4 Summary of JavaScript Features: Total Occurrences, Adoption Rate in Projects, First Occurrence, and Year of Release without filtering smaller projects

Feature	Total of Occurrences (#)	Projects Adoption (%)	First Occurrence
Const Declarations	87337	71.51	2012-01
Async Declarations	13288	68.98	2012-01
Arrow Function Declarations	52318	67.08	2013-11
Let Declarations	37484	66.45	2012-01
Enhanced Property Assignment	26266	60.75	2013-08
Template String Expressions	14048	58.86	2014-07
Object Destructuring	5119	46.83	2013-08
Spread Arguments	1224	43	2014-11
Default Parameters	1444	42.40	2014-06
Await Declarations	19560	37.34	2016-02
Import Statements	17581	36.70	2013-07
Function Property Declarations	6508	36.70	2014-11
For of Statments	1398	36.07	2014-07
Class Declarations	1716	35.44	2014-01
Export Declarations	6498	32.91	2013-06
Array Destructuring	556	29.11	2013-08
Computed Property Assignment	714	24.05	2014-11
Rest Statements	287	22.78	2014-05
Spread in Objects	247	15.82	2015-11
Optional Chain	1004	13.29	2020-09
Null Coalesce Operators	185	6.32	2020-09
Rest in Objects	25	5.69	2019-03
Optional Catch Biding	58	3.16	2016-12
Yield Declarations	242	3.16	2012-01
Private Fields	83	2.53	2022-09
For Await Of Statments	8	1.89	2020-11
Private Methods	24	1.26	2022-09
BigInt	1	0.63	2023-01
Numeric Separator	2	0.63	2021-10

However, we believe that our study offers valuable insights into the development practices of JavaScript developers, which can be relevant for similar contexts and provide guidance for developers working on mature and evolving codebases.

Regarding the qualitative investigation, we utilized coding methods to analyze the pull-request comments. Despite efforts to mitigate bias by involving multiple coders, the authors' experience and motivations might have influenced the process. While we aimed to uncover and report the motivations and challenges of source code rejuvenation through code review, some limitations may still exist. We describe these limitations based on the steps taken to enhance confidence and validity. Additionally, we followed approaches explored in previous studies (Bacchelli and Bird 2013; Sadowski et al. 2018; Oliveira 2021).

Still, on external validity, another potential threat is related to the temporality of the analyzed code review comments. Although we collected comments from 2012 to 2023, the relevant comments regarding code rejuvenation efforts found in our study were only from the period between 2015 and 2021. During this period, the JavaScript programming language have undergone considerable change and maturation, especially with support for modern features through transpilers such as Babel or syntactic supersets such as TypeScript. As a result, the challenges identified in the study may not fully reflect the current reality, limiting the generalizability of the results in terms of more recent development practices. To overcome this limitation, we recommend that developers and readers should also interpret the study's recommendations as cautions about potential challenges, such as compatibility issues and performance overhead.

7 Related Works

Our study builds upon prior research that explores the adoption of modern programming language features across various languages, including Kotlin, Python, TypeScript, C++, Java, and JavaScript. Specifically, it investigates the motivations, challenges, and trends associated with adopting and utilizing these features. In this context, we categorize the related work into two main areas:

(i) **Tools and Techniques for Code Rejuvenation** - Studies that focus on methodologies, frameworks, and automation strategies for integrating modern language features into existing codebases.

(ii) **Feature Adoption and Usage Analysis** - Research examining how and why developers adopt modern language features, as well as the reasons behind developers' adoption of modern language features and the challenges they face during this process.

7.1 Tools and Techniques for Code Rejuvenation

As programming languages evolve, new features are introduced to enhance code comprehension, improve conciseness, and streamline development. Following these advancements, developers seek strategies and techniques to adapt existing codebases to incorporate these modern features. For example, Kumar et al. examined the rejuvenation of legacy C++ programs by replacing C preprocessor macros with C++11 features like generalized constant expressions, perfect forwarding, and lambda expressions. This transition enhances code clarity and maintainability, supporting modern software design. However, adoption faces challenges, particularly in automating refactoring without breaking functionality, as some macros reference free variables or recursive instantiations. The study emphasizes the need for careful analysis and iterative refinement to ensure a successful transformation (Kumar et al. 2012).

Dantas et al. investigated Java legacy system evolution by applying 2,462 transformations to 40 open-source projects via pull requests. Their findings show that transformations improving readability are generally accepted, while substantial changes may be rejected. This underscores the importance of code clarity and highlights the subjective nature of perceived improvements, especially when mixed idioms are involved and functional programming styles are less familiar to some developers (Dantas et al. 2018).

Gokhale et al. introduced a static analysis technique for refactoring JavaScript applications from synchronous to asynchronous APIs using async/await, aiming to minimize code disruption. Evaluating 12 applications with 316 API calls, their tool, Desynchronizer, successfully refactored 244 out of 256 candidates. While asynchronous APIs enhance performance, responsiveness, and scalability, challenges include higher syntactic complexity and potential behavioral changes, emphasizing the need for thorough testing during migration (Gokhale et al. 2021).

Finally, Nicolini et al. examined JavaScript feature adoption via transpilers, analyzing 1,000 open-source projects to assess Babel plug-in usage and adoption challenges. They find that 73 projects use at least one new JavaScript proposal, relying on Babel to handle compatibility. With 86% average browser support, transpilers play a key role in modern web development. Developers adopt new features for performance and early adoption, but face cross-browser compatibility issues and integration complexity (Nicolini et al. 2024).

7.2 Feature Adoption and Usage Analysis

After the release of new programming language versions that introduce modern features, researchers began analyzing how developers adopt these modern features. Studies on feature adoption in languages like Kotlin, TypeScript, Python, C++, Java and JavaScript explore when and how developers integrate these features, as well as the challenges that hinder their adoption. For example, Mateus and Martinez investigated Kotlin feature adoption in Android development using 387 open-source applications, finding that 15 out of 26 features are frequently used, with type inference, lambda, and safe call being the most prevalent. Essential features are often adopted early, while less common ones appear later. However, developers face challenges due to the lack of benchmarks and training materials, limiting the adoption of features like property delegation. The study highlights the need for better tutorials to encourage broader use of advanced Kotlin capabilities (Mateus and Martinez 2020).

Peng et al. analyzed Python feature usage across 35 projects from eight domains, highlighting developer preferences for simple and safe features like safety checks, testing, and dynamic capabilities while avoiding complex constructs such as heterogeneous lists and diamond inheritance. Exception handling, decorators, and nested classes/functions are frequently used, with a focus on custom decorators and ImportError handling. However, modern features like gradual typing and keyword-only arguments face low adoption due to complexity and potential errors, despite their benefits for safety and performance (Peng et al. 2021).

Scarsbrook et al. analyzed 454 open-source TypeScript repositories to examine feature adoption trends over three years. They find that type modifiers on import and template literal types are frequently adopted, with over 20 repositories using them within a year, while features like static blocks in classes see much lower adoption. Despite frequent compiler updates, many language features are adopted more slowly, suggesting that developers prioritize compiler upgrades over new features, possibly due to complexity or perceived utility (Scarsbrook et al. 2023).

Uesbeck et al. analyzed the impact of C++11 lambda expressions through a randomized controlled trial, comparing them with iterators among students and professional developers. The study finds no significant benefits in time efficiency or error reduction, with students facing more compiler errors and longer task completion times. Developers reported debugging difficulties, complicating adoption in areas like concurrent programming. While lambdas can simplify code, they may introduce unintended complexities, suggesting the need for further research on their practical impact (Uesbeck et al. 2016).

Chen et al. investigated C++ template usage across 1,267 revisions from 50 open-source systems, finding that templates mainly prevent code duplication, but their benefits are limited to a few frequently used cases. Function templates do not fully replace C-style generics, and developers with a C background show no strong preference between them. Adoption of modern C++11 templates varies, with variadic class templates frequently used, while variadic function templates and alias templates see lower adoption. The study highlights compatibility challenges and the need for better tooling to support modern template adoption (Chen et al. 2020).

Lucas et al. examined C++ rejuvenation practices in KDE projects, finding growth adoption of lambda expressions, range-based for loops, and auto-typed variables in over 60% of projects, while modern multi-threading features remain rarely used. Developers leverage tools like Clazy and Clang-Tidy to facilitate large-scale updates, driven by goals of improving readability, conciseness, and attracting new contributors. However, challenges such as high update costs, framework incompatibilities, and resistance to change hinder the adoption of modern features (Lucas et al. 2023).

Dyer et al. analyzed 31k open-source Java projects, revealing varied adoption rates of new language features, with enhanced-for loops and annotations being the most popular. The study finds that some features are adopted before their official release, showing developer anticipation and readiness. Motivations for adoption include improving code efficiency and clarity, while challenges such as lack of awareness and training hinder broader usage (Dyer et al. 2014a).

Mazinanian et al. investigated lambda expression adoption in Java using 241 open-source projects and surveys from 97 developers. The study finds a two-fold increase in lambda usage from 2015 to 2016, driven by motivations like improving readability, reducing duplication, and enabling lazy evaluation. However, challenges include inefficient use of functional interfaces and difficulties adapting to a functional programming mindset (Mazinanian et al. 2017).

Similarly, Lucas et al. examined the impact of lambda expressions on Java program comprehension through a mixed-method study, analyzing 66 code snippet pairs. The findings reveal a contradiction: while quantitative analysis found no readability improvement, qualitative insights suggest enhanced comprehension. Developers adopt lambdas for simplified code, but challenges include understanding functional programming and the cognitive load of transitioning from traditional constructs (Lucas et al. 2019).

Zheng et al. analyzed the removal of lambda expressions in Java, analyzing 117 issues from Apache JIRA, code commits, GitHub issues, and a user study. The findings show that performance issues and memory leaks are key reasons for removal. Developers remove lambdas to simplify code and prevent bugs, but face challenges in managing inappropriate lambda usage that introduces side effects (Zheng et al. 2021).

Silva et al. examined class usage in JavaScript, finding that 40% of systems rely on class-based design, while prototype-based inheritance remains unpopular. Using static analysis on 50 GitHub projects, they show that developer experience influences class adoption more than system size. Challenges include prototype complexity and the lack of encapsulation mechanisms, highlighting the need for better tools to support class-based development (Silva et al. 2015).

Alves et al. analyzed functional programming adoption in JavaScript, examining 91 open-source repositories and trends in recursion, immutability, lazy evaluation, and functions as values. The study finds a 255% increase in immutability-related structures and a rise in lazy evaluation and higher-order functions, while recursion and callbacks declined. Functional structures are less likely to be removed in bug-fixing commits, suggesting lower error-proneness. Developers adopt these features for modularity, clarity, and maintainability, but refactoring existing systems remains a key challenge (Alves et al. 2022).

Our study reveals the early and frequent adoption of modern JavaScript features, such as const declarations and arrow functions, reinforcing prior research that highlights similar adoption patterns in JavaScript (Silva et al. 2015) and other programming languages like Kotlin (Mateus and Martinez 2020), TypeScript (Scarsbrook et al. 2023), and Java (Mazinanian et al. 2017). Motivations for adopting these features include improved code readability, conciseness, and maintainability. However, the variability in feature usage-such as the high adoption of Template String Expressions and the lower adoption of Await Operators-highlights contextual differences influenced by tooling, compatibility issues, and developer familiarity (Gokhale et al. 2021; Scarsbrook et al. 2023).

Challenges in adopting modern features are consistent with findings across programming languages, including compatibility issues with older constructs (Lucas et al. 2023), performance concerns (Kumar et al. 2012), and the cognitive burden of adopting functional programming paradigms (Alves et al. 2022). Transpilers like Babel play a pivotal role in facilitating feature adoption by addressing cross-browser compatibility but introduce trade-offs, such as increased code complexity and potential bugs (Nicolini et al. 2024). These findings emphasize the need for tools and strategies that balance modern feature adoption with the practical constraints of legacy systems.

This study advances the literature on JavaScript by providing a comprehensive analysis of the adoption patterns, motivations, and challenges associated with modern language features introduced from ES6 onwards. It highlights trends of early adoption, the critical role of transpilers like Babel, and the nuanced decision-making developers face in balancing modern feature integration with legacy constraints. By identifying gaps in adoption, such as low uptake of Await Operators and challenges in functional programming, the study offers insights that complement findings in other languages, emphasizing the unique dynamics of JavaScript's ecosystem.

8 Conclusions and Future Works

This paper has explored the general question related to the adoption of modern JavaScript features—that is, features introduced in the sixth version of the ECMAScript standard (ES6) onwards. The research brings insights into the adoption patterns, timeline, and trends of modern JavaScript features.

The reliance on modern JavaScript features in open-source repositories is quite high for certain modern features, with a varied adoption pattern for others. Developers tend to adopt new features early, reflecting a forward-thinking approach to utilizing the language's evolving capabilities. The adoption trends of modern JavaScript features demonstrate a clear pattern of gradual uptake following their official introduction in ECMAScript versions. While some features are quickly integrated into development practices, others require a maturation period before seeing significant usage. This behavior aligns with broader trends observed in the adoption of new programming language features across different development communities (Dyer et al. 2014b; Lucas et al. 2023).

The exploration of motivations and challenges in rejuvenating legacy JavaScript code reveals a complex landscape where developers balance the benefits of adopting modern language features with practical impediments. Improving code comprehension emerges as a primary motivation, driven by the desire to enhance conciseness and maintainability through features like Arrow Function Declarations, Async Declarations, and Await Operators. However, pervasive compatibility issues with older browsers, libraries, and runtime environments present formidable challenges. Resistance to change rooted in existing code conventions, concerns over performance impacts, and the perceived complexity of new syntax further complicate adoption efforts. These findings underscore the nuanced decision-making process developers face when rejuvenating codebases, emphasizing the need for pragmatic strategies that reconcile technological advancement with practical constraints.

Regarding the findings related to the transpiler-based approach for rejuvenating JavaScript code, it is evident that while tools like Babel facilitate the adoption of modern language features, they introduce complexities and potential challenges. Developers utilize transpilers to bridge compatibility gaps between modern JavaScript syntax and legacy environments, enabling immediate integration of advanced features like Arrow Function Declarations, Async Declarations, and Await Operators constructs. However, our analysis highlights significant trade-offs, as transpilation can lead to increased code complexity and potential bugs, illustrated by issues in Babel's handling of Arrow Function Declarations and performance overhead concerns.

Additionally, our analysis of rejuvenation efforts in JavaScript code revealed growth adoption of modern language features across various projects, reflecting a concerted effort to enhance code quality and maintainability. These rejuvenation efforts collectively contribute to modernize JavaScript codebases, enhancing overall software quality, yet underscore the need for careful consideration and strategic implementation of transpiler-based approaches to ensure effective code rejuvenation efforts in JavaScript projects.

For future work, we identified two opportunities to delve deeper into the factors influencing the adoption patterns of programming language features. By looking at how things like community support, the availability of tools, and the quality of the documentation affect the use of a feature, researchers can find out what makes people adopt it.

Additionally, another interesting direction for future work would be to investigate whether the limited adoption of certain modern JavaScript features is influenced by a lack of external support, such as compatibility with libraries, runtime environments, or less common browsers. While our study acknowledges that major browsers fully support the features analyzed, the extent to which external constraints impact feature adoption remains unclear. A dedicated study exploring these potential barriers could provide deeper insights into the challenges open-source developers face and further contextualize the adoption trends observed in our analysis.

These opportunities for future research may contribute to our understanding of software evolution and programming language development dynamics.

Open Access

A: Appendix

Table 5 Feature adoption in various projects by source code rejuvenation with corresponding commit details and quantities

Project name	commit link	Feature adopted & Qtd
bookreader	a2f4c92e7	Arrow Function = 4
bookreader	17beab4b4	Const = 5
bookreader	d21cc72d5	Arrow Function = 1
chrome-extensions-samples	0672807d4b	Optional Chaining = 1
cinnamon	fb0c23a363	Default Parameter = 1, Spread Statements = 1
client-nodejs	ad024e1a4a0	Arrow Function = 1
converse.js	654e72baad8	Optional Chaining = 2
ember.js	f8812ee1d7	Class = 4
encoded	0d9dd05be0	Optional Chaining = 1
encoded	3ce1713736	Import = 1, Export = 2
hexo	1ff82654cf	Arrow Function = 1
hexo	4d071c4102	Arrow Function = 3
highlight.js	a69eb6a2e	Arrow Function = 92
jasmine	1166d10e4	Const = 2038, Let = 152
modernizr	4c263c258	Import = 5, Export = 3
node-postgres	039b5baef6e	Arrow Function = 1
novnc	cdb860ad84	Const = 5, Let = 7
novnc	2b5f94fa6a	Const = 681, Let = 214
novnc	0e4808bf6f3	Class = 10
novnc	651c23ece37	Arrow Function = 111
openpgpjs	5142003b09	Await Declarations = 6, Object Destructuring = 6
pouchdb	f30b9fbab9e2	Const = 1, Let = 1, Arrow Function = 2, Async = 1, Await = 2
pouchdb	99c24a5861e	Import = 341
qunit	f0da7b88197	Arrow Function = 1, Default Parameter = 1
qunit	df70c00a864	Spread Statements = 1
qunit	dc2887b7fe7	Arrow Function = 2
sefaria-project	182f74872e	Default Parameter = 3, Object Destructuring = 1
travis-web	0e6dc8db4e	Arrow Function = 1
travis-web	4cac82832f	Arrow Function = 1
video.js	9e76477b492	Arrow Function = 1
video.js	ffac358bf745a	Arrow Function = 1

Acknowledgements We also thank the anonymous reviewers for their careful reading of our manuscript and their many insightful comments and helpful suggestions.

Author Contributions *Walter Lucas, Rafael Nunes, Rodrigo Bonifácio, Fausto Carvalho, and Michael Silva* - Conceptualization, Data curation, Methodology, Software, Formal analysis, Investigation, Writing - Original Draft, Visualization, Supervision and *Ricardo Lima, Adriano Torres, Paola Accioly, Eduardo Monteiro, and João Saraiva* Data curation, Supervision, Writing - review & editing.

Funding Not applicable.

Code and Data Availability Our study is fully reproducible. All developed tools, collected data, and Python scripts are available online for statistical analysis in Mendonça (2025).

Declarations

Conflicts of interests The authors declare that they have no conflict of interest.

Ethical approval and Informed consent Not applicable.

References

Alves F, Oliveira D, Madeiral F, Castor F (2022) On the bug-proneness of structures inspired by functional programming in javascript projects. https://doi.org/10.48550/arXiv.2206.08849

Andreasen E, Gong L, Møller A, Pradel M, Selakovic M, Sen K, Staicu C (2017) A survey of dynamic analysis and test generation for javascript. ACM Comput Surv 50(5):66:1–66:3https://doi.org/10.1145/3106739

Avelino G, Constantinou E, Valente MT, Serebrenik A (2019) On the abandonment and survival of open source projects: An empirical investigation. In: 2019 ACM/IEEE international symposium on empirical software engineering and measurement (ESEM), pp 1–12, https://doi.org/10.1109/ESEM.2019.8870181

Bacchelli A, Bird C (2013) Expectations, outcomes, and challenges of modern code review. In: Notkin D, Cheng BHC, Pohl K (eds) 35th international conference on software engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013, IEEE Computer Society, pp 712–721, https://doi.org/10.1109/ICSE.2013.6606617

Bogachenkova V, Nguyen L, Ebert F, Serebrenik A, Castor F (2022) Evaluating atoms of confusion in the context of code reviews. In: IEEE international conference on software maintenance and evolution, ICSME 2022, Limassol, Cyprus, October 3-7, 2022, IEEE, pp 404–408, https://doi.org/10.1109/ICSME55016.2022.00048

Brito A, Hora AC, Valente MT (2022) Towards a catalog of composite refactorings. CoRR arXiv:2201.04599

Cantelon M, Harter M, Holowaychuk T, Rajlich N (2013) Node.Js in Action, 1st edn. Manning Publications Co., USA

Chen L, Wu D, Ma W, Zhou Y, Xu B, Leung H (2020) How c++ templates are used for generic programming: An empirical study on 50 open source systems. ACM Trans Softw Eng Methodol 29(1), https://doi.org/10.1145/3356579

Clarke V, Braun V (2017) Thematic analysis. J Posit Psychol 12(3):297–298

Cleveland WS (1979) Robust locally weighted regression and smoothing scatterplots. J Am Stat Assoc 74(368):829–836

Dabic O, Aghajani E, Bavota G (2021) Sampling projects in github for MSR studies. In: 18th IEEE/ACM international conference on mining software repositories, MSR 2021, IEEE, pp 560–564

Daka E, Campos J, Fraser G, Dorn J, Weimer W (2015) Modeling readability to improve unit tests. In: Nitto ED, Harman M, Heymans P (eds) Proceedings of the 2015 10th joint meeting on foundations of software engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015, ACM, pp 107–118, https://doi.org/10.1145/2786805.2786838

Danial A (2021) cloc: v1.92. https://doi.org/10.5281/zenodo.5760077

Dantas R, Carvalho A, Marcilio D, Fantin L, Silva U, Lucas W, Bonifácio R (2018) Reconciling the past and the present: An empirical study on the application of source code transformations to automatically rejuvenate java programs. In: Oliveto R, Penta MD, Shepherd DC (eds) 25th international conference on software analysis, evolution and reengineering, SANER 2018, Campobasso, Italy, March 20-23, 2018, IEEE Computer Society, pp 497–501, https://doi.org/10.1109/SANER.2018.8330247

Dyer R, Rajan H, Nguyen HA, Nguyen TN (2014a) Mining billions of ast nodes to study actual and potential usage of java language features. In: Proceedings of the 36th international conference on software engineering, association for computing machinery, New York, NY, USA, ICSE 2014, p 779-790, https://doi.org/10.1145/2568225.2568295

Dyer R, Rajan H, Nguyen HA, Nguyen TN (2014b) Mining billions of AST nodes to study actual and potential usage of java language features. In: Jalote P, Briand LC, van der Hoek A (eds) 36th international conference on software engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014, ACM, pp 779–790, https://doi.org/10.1145/2568225.2568295

Ecma E (1997) 262: Ecmascript language specification. Tech. rep., ECMA (european association for standardizing information and communication systems), pub-ECMA: adr,https://www.ecma-international.org/publications-and-standards/standards/ecma-262/

Flanagan D (2020) JavaScript: The Definitive Guide : Master the World's Most-used Programming Language. O'Reilly Media, Incorporated, https://books.google.com.br/books?id=b5G4zAEACAAJ

Gokhale S, Turcotte A, Tip F (2021) Automatic migration from synchronous to asynchronous javascript apis. Proc ACM Program Lang 5(OOPSLA), https://doi.org/10.1145/3485537

Haverbeke M (2014) Eloquent JavaScript: A Modern Introduction to Programming, 2nd edn. No Starch Press, USA

Huang C, Petukhina A (2022) Applied Time Series Analysis and Forecasting with Python. Statistics and computing, springer international publishing. https://books.google.com.br/books?id=TfaVEAAAQBAJ

Keith J (2005) DOM Scripting. Apress. https://doi.org/10.1007/978-1-4302-0062-8

Kelhini FK (2021) Modern Asynchronous JavaScript. Pragmatic Bookshelf, https://www.amazon.com/dp/B09NQQ8715

Kumar A, Sutton A, Stroustrup B (2012) Rejuvenating c++ programs through demacrofication. In: 2012 28th IEEE international conference on software maintenance (ICSM), pp 98–107, https://doi.org/10.1109/ICSM.2012.6405259

Lin HY, Thongtanunam P (2023) Towards automated code reviews: Does learning code structure help? In: Zhang T, Xia X, Novielli N (eds) IEEE international conference on software analysis, evolution and reengineering, SANER 2023, Taipa, Macao, March 21-24, 2023, IEEE, pp 703–707, https://doi.org/10.1109/SANER56733.2023.00075

Lucas W, Bonifácio R, Canedo ED, Marcilio D, Lima F (2019) Does the introduction of lambda expressions improve the comprehension of java programs? In: do Carmo Machado I, Souza R, Maciel RSP, Sant'Anna C (eds) Proceedings of the XXXIII Brazilian symposium on software engineering, SBES 2019, Salvador, Brazil, September 23-27, 2019, ACM, pp 187–196, https://doi.org/10.1145/3350768.3350791

Lucas W, Carvalho F, Nunes RC, Bonifácio R, Saraiva J, Accioly P (2023) Embracing modern c++ features: An empirical assessment on the kde community. J Softw Evol Process n/a(n/a):e2605. https://doi.org/10.1002/smr.2605

Mateus BG, Martinez M (2020) On the adoption, usage and evolution of kotlin features in android development. In: Proceedings of the 14th ACM / IEEE international symposium on empirical software engineering and measurement (ESEM), association for computing machinery, New York, NY, USA, ESEM '20, https://doi.org/10.1145/3382494.3410676

Mazinanian D, Ketkar A, Tsantalis N, Dig D (2017) Understanding the use of lambda expressions in java. Proc ACM Program Lang 1(OOPSLA), https://doi.org/10.1145/3133909

Mendonça WL (2025). Understanding the adoption of modern javascript features: An empirical study on open-source systems. https://doi.org/10.5281/zenodo.14796287

Morgan J, Stewart A (2018) Simplifying JavaScript: Writing Modern JavaScript with ES5, ES6, and Beyond. The pragmatic programmers, pragmatic bookshelf, https://books.google.com.br/books?id=SIyAtAEACAAJ

Nicolini T, Hora AC, Figueiredo E (2024) On the usage of new javascript features through transpilers: The babel case. IEEE Softw 41(1):105–112. https://doi.org/10.1109/MS.2023.3243858

Oliveira D (2021) Recommending code understandability improvements based on code reviews. In: 36th IEEE/ACM international conference on automated software engineering, ASE 2021 - Workshops, Melbourne, Australia, November 15-19, 2021, IEEE, pp 131–132, https://doi.org/10.1109/ASEW52652.2021.00035

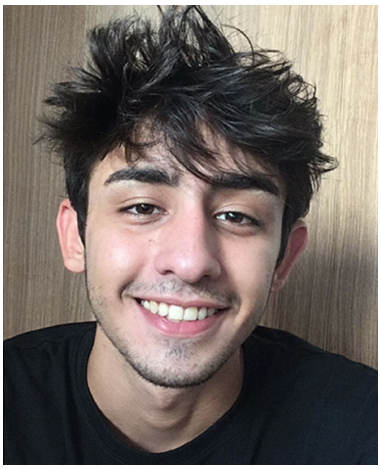
Oliveira D, Bruno R, Madeiral F, Castor F (2020) Evaluating code readability and legibility: An examination of human-centric studies. In: IEEE international conference on software maintenance and evolution, ICSME 2020, Adelaide, Australia, September 28 - October 2, 2020, IEEE, pp 348–359, https://doi.org/10.1109/ICSME46990.2020.00041

Parr T (2013) The definitive antlr 4 reference. The definitive ANTLR 4 Reference pp 1–326

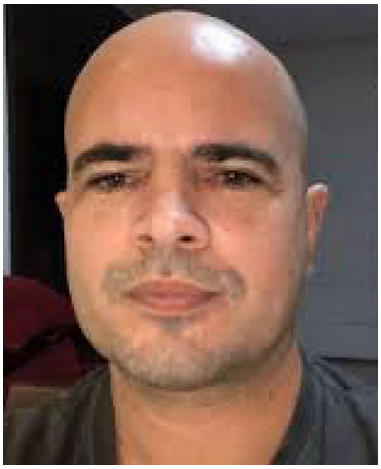
Peng Y, Zhang Y, Hu M (2021) An empirical study for common language features used in python projects. In: 28th IEEE international conference on software analysis, evolution and reengineering, SANER 2021,

Honolulu, HI, USA, March 9-12, 2021, IEEE, pp 24–35, https://doi.org/10.1109/SANER50967.2021.00012
Phang CL (2022) ECMAScript 2022: The definitive guide to modern JavaScript. Self-published, https://www.amazon.com/ECMAScript-2022-Definitive-Modern-JavaScript/dp/B0BLG6SWRT
Pirkelbauer P, Dechev D, Stroustrup B (2010) Source code rejuvenation is not refactoring. In: van Leeuwen J, Muscholl A, Peleg D, Pokorný J, Rumpe B (eds) SOFSEM 2010: Theory and practice of computer science, 36th conference on current trends in theory and practice of computer science, Spindleruv Mlýn, Czech Republic, January 23-29, 2010. Proceedings, Springer, Lect Notes Comput Sci, vol 5901, pp 639–650, https://doi.org/10.1007/978-3-642-11266-9_53
Rauschmayer A (2015) Exploring ES6: Upgrade to the next version of JavaScript. Leanpub, https://exploringjs.com/es6/
Rauschmayer A (2024) Exploring JavaScript. Self-published, https://exploringjs.com/js/index.html
Resig J, Bibeault B, Maras J (2016) Secrets of the JavaScript Ninja, Second Edition. ITpro collection, Manning Publications, https://books.google.com.br/books?id=dq16zQEACAAJ
Rosenkranz S, Staegemann D, Volk M, Turowski K (2024) Explaining the business-technological age of legacy information systems. IEEE Access 12:84579–84611. https://doi.org/10.1109/ACCESS.2024.3414377
Sadowski C, Söderberg E, Church L, Sipko M, Bacchelli A (2018) Modern code review: a case study at google. In: Paulisch F, Bosch J (eds) Proceedings of the 40th international conference on software engineering: software engineering in practice, ICSE (SEIP) 2018, Gothenburg, Sweden, May 27 - June 03, 2018, ACM, pp 181–190, https://doi.org/10.1145/3183519.3183525
Saldana J (2012) The coding manual for qualitative researchers. English short title catalogue 18th Century collection, SAGE Publications, https://books.google.com.br/books?id=kUms8QrE_SAC
Scarsbrook JD, Utting M, Ko RKL (2023) Typescript's evolution: An analysis of feature adoption over time. arXiv:2303.09802
Silva D, da Silva JP, de Souza Santos GJ, Terra R, Valente MT (2021) Refdiff 2.0: A multi-language refactoring detection tool. IEEE Trans Software Eng 47(12):2786–2802, https://doi.org/10.1109/TSE.2020.2968072
Silva LH, Ramos M, Valente MT, Bergel A, Anquetil N (2015) Does javascript software embrace classes? In: Guéhéneuc Y, Adams B, Serebrenik A (eds) 22nd IEEE international conference on software analysis, evolution, and reengineering, SANER 2015, Montreal, QC, Canada, March 2-6, 2015, IEEE Comput Soc, pp 73–82, https://doi.org/10.1109/SANER.2015.7081817
Tsantalis N, Mansouri M, Eshkevari LM, Mazinanian D, Dig D (2018) Accurate and efficient refactoring detection in commit history. In: Chaudron M, Crnkovic I, Chechik M, Harman M (eds) Proceedings of the 40th international conference on software engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018, ACM, pp 483–494, https://doi.org/10.1145/3180155.3180206
Uesbeck PM, Stefik A, Hanenberg S, Pedersen J, Daleiden P (2016) An empirical study on the impact of C++ lambdas and programmer experience. In: Dillon LK, Visser W, Williams LA (eds) Proceedings of the 38th international conference on software engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016, ACM, pp 760–771, https://doi.org/10.1145/2884781.2884849
Vishwas B, Patel A (2020) Hands-on time series analysis with python: From basics to bleeding edge techniques. Apress, https://books.google.com.br/books?id=cTeHzQEACAAJ
Wirfs-Brock A, Eich B (2020) Javascript: The 1st 20 years. Proc ACM Program Lang 4(HOPL), https://doi.org/10.1145/3386327
Zhao Y, Serebrenik A, Zhou Y, Filkov V, Vasilescu B (2017) The impact of continuous integration on other software development practices: a large-scale empirical study. In: Rosu G, Penta MD, Nguyen TN (eds) Proceedings of the 32nd IEEE/ACM international conference on automated software engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017, IEEE Comput Soc, pp 60–71, https://doi.org/10.1109/ASE.2017.8115619
Zheng M, Yang J, Wen M, Zhu H, Liu Y, Jin H (2021) Why do developers remove lambda expressions in java? In: 36th IEEE/ACM international conference on automated software engineering, ASE 2021, Melbourne, Australia, November 15-19, 2021, IEEE, pp 67–78, https://doi.org/10.1109/ASE51524.2021.9678600

Walter Lucas is a PhD candidate in Computer Science at the University of Brasília since 2019, holding an MSc from UNB and a Bachelor's in Information Systems. From 2020 to 2022, he contributed to a collaborative R&D project with FAPDF/UNB/CNJ. His work focuses on software engineering, code comprehension, software evolution, and program transformations, with over five years of R&D experience.



Rafael Nunes is a Computer Science BSc undergraduate at the University of Brasília. Currently working as a Software Engineer for the Payment Industry, he's interested in studying computer networks, compilers, and distributed systems.



Rodrigo Bonifácio is an associate professor at the Computer Science Department, University of Brasília, Brazil, and a member of the Software Productivity Group, led by Prof. Paulo Borba. His research interests lie in understanding the common usage of programming language features, program transformations, software architecture and modularity, and software security.

Fausto Carvalho is a Master's student in Informatics at the University of Brasília with a B.Sc. in Computer Science and more than 24 years of experience in software engineering across Brazil and Canada. Research interests include program analysis, software security, fuzzing, and programming languages.

Ricardo Lima is a Ph.D. candidate in Computer Science at Universidade de Brasília. His research interests include artificial intelligence, software engineering and digital government. He works as General Coordinator of Digital Economy, at the Ministry of Industry in Brazil, working with the formulation and coordination of public policies related to the Digital Economy, with a focus on the digital transformation of the Brazilian industrial sector.

Mchael Silva is a PhD student in Machine Learning applied to Software Engineering at the University of Brasilia. He holds a Master's degree in Computer Science from the same university where he studied the application of machine learning to improve user experience in banking systems. He is currently a Senior Application Architect at Amazon focusing on the financial services area.

Adriano Torres is a Master of Philosophy graduate at the University of Adelaide and a PhD candidate at the University of Brasilia. Whilst pursuing his academic interests, Adriano has kept working in both startups and big corporations in the software industry throughout his entire career. He is a strong believer that in the software industry, theory should follow mostly from practice, and that it is essential for software engineering research to both seek to involve actual practitioners and to learn from them. His main interests are programming languages and compilers.

Paola Accioly is a professor at the Center for Informatics of the Federal University of Pernambuco, Brazil. Her research interests include support for collaborative software development and empirical software engineering. She holds a PhD in Computer Science from the Federal University of Pernambuco and has previously worked at the Federal University of Cariri.

Eduardo Monteiro is a professor in the Department of Statistics at the University of Brasília. His research focuses on applied statistics, particularly in the domains of technology, agriculture, and public sector data analysis.

Jo ao Saraiva is an Associate Professor at the department of Informatics, University of Minho, Portugal, and a senior research member of HASLab/INESC TEC. He obtained a Ph.D. degree in Computer Science from Utrecht University in 1999. His main research contributions have been in the field of programming language design and implementation, green software, and functional programming.

Authors and Affiliations

Walter Lucas[1] · Rafael Nunes[1] · Rodrigo Bonifácio[1] · Fausto Carvalho[1] · Ricardo Lima[1] · Michael Silva[1] · Adriano Torres[1] · Paola Accioly[2] · Eduardo Monteiro[1] · João Saraiva[3]

Walter Lucas
waltimlmm@gmail.com

Rafael Nunes
nunes.rafael@aluno.unb.br

Rodrigo Bonifácio
rbonifacio@unb.br

Fausto Carvalho
faustocarva@gmail.com

Ricardo Lima
ricdelima@gmail.com

Michael Silva
michaeldf.ti@gmail.com

Adriano Torres
adrianortorres@gmail.com

Paola Accioly
prga@cin.ufpe.br

Eduardo Monteiro
edumonteiro@unb.br

João Saraiva
saraiva@di.uminho.pt

1 University of Brasília, Brasília, Brazil

2 Federal University of Pernambuco, Recife, Pernambuco, Brazil

3 University of Minho, Braga, Portugal

Appendix C

Appendix C — Python Study

“It Makes the Code Clearer”: Why Developers Adopt Modern Python Features in Open Source Projects

Walter Mendonça[a,*], Marcos Leite[a], Oseias Romeiro[a], Fausto Carvalho[a], Rodrigo Bonifácio[c], Eduardo Monteiro[a], Gustavo Pinto[b], Paola Accioly[c] and João Saraiva[d,e]

[a] University of Brasília, Brasília, DF, Brazil
[b] Federal University of Pará, Belém, PA, Brazil
[c] Universidade Federal de Pernambuco, Recife, PE, Brazil
[d] Universidade do Minho, Braga, Portugal
[e] HASLab, INESC TEC, Braga, Portugal

ARTICLE INFO

Keywords:
Empirical software engineering
Mining software repositories
Software evolution
Source code Rejuvenation
Code Comprehension

ABSTRACT

Python has become one of the most widely used programming languages, yet the transition from Python 2 to 3 introduced a tension between innovation and compatibility. While new features such as formatted string literals, type annotations, and structural pattern matching expanded the language’s expressiveness, they also required substantial adaptation of legacy code. Despite the increasing relevance of these features, there is still limited empirical evidence on how modern Python features are being adopted in practice—when developers start using them, how adoption unfolds over time, and what motivations drive these decisions. This paper addresses this gap through a large-scale empirical study of 424 open-source Python projects. Our analysis reveals two distinct adoption patterns: rapid adoption of small syntactic improvements and slower integration of features that require extensive refactoring or ecosystem support. On average, projects begin using with new features within 16 months after their release but take roughly 4 years to achieve broader and sustained adoption. This observation may be partially explained by the transition from Python 2 to 3, which did not preserve full backward compatibility. Complementary qualitative evidence from pull-request discussions indicates that developers are primarily motivated to rejuvenate Python code through improvements in comprehension, safety, and performance, yet often constrained by compatibility requirements and maintenance costs. Together, these findings offer practical insights for tool developers and maintainers seeking to balance innovation and stability in the ongoing rejuvenation of Python source code.

1. Introduction

The Python programming language is among the most popular and versatile languages today, supported by a rich ecosystem that enables developers to build applications across a wide range of domains, including web development, data science, and artificial intelligence [2, 27]. Its clear syntax and extensive library support have contributed to its widespread adoption. This broad reach underscores the strength of the Python community, which actively contributes tools and best practices tailored to diverse application areas, solidifying Python as a cornerstone of modern software development.

Unlike languages that emphasize backward compatibility, such as Java [8, 38] and C++ [33], Python underwent a major breaking change between versions 2 and 3, prompting extensive debate on compatibility and language evolution [36, 37, 27]. At the same time, Python 3 introduced substantial enhancements—native support for asynchronous programming, formatted string literals (f-strings) [28], and new syntax for variable and function type annotations [37]—followed more recently by structural pattern matching, which enables clearer and more concise code [30, 47, 54].

Despite these improvements, the lack of backward compatibility posed significant challenges for Python developers, who needed to adapt legacy code and carefully manage the trade-offs between embracing language innovation and

*Corresponding author
waltimlmm@gmail.com (W. Mendonça); maldsjr@hotmail.com (M. Leite); magalhaes.oseias@aluno.unb.br (O. Romeiro); faustocarva@gmail.com (F. Carvalho); rbonifacio@cin.ufpe.br (R. Bonifácio); edumonteiro@unb.br (E. Monteiro); gpinto@ufpa.br (G. Pinto); prga@cin.ufpe.br (P. Accioly); araiva@di.uminho.pt (J. Saraiva)
ORCID(s):

source code rejuvenation efforts. Rejuvenation is a special type of software maintenance aimed at replacing legacy constructs and idioms of a programming language with their modern counterparts [42]. This leads to a general question: *How, when, and why do open-source Python projects adopt modern language features, and what factors facilitate or hinder the rejuvenation of legacy code?*

Although prior studies have examined the use of specific Python 3 features and language transitions through feature-specific analyses, migration studies, and controlled evaluations [36, 37, 30, 41, 47, 54], existing research remains fragmented in scope and methodology. In particular, there is a lack of systematic, large-scale empirical studies that characterize how modern Python features are adopted over time across open-source projects, especially in the years following the end-of-life of Python 2.7. Moreover, most prior work either focuses on isolated language constructs or relies on static snapshots of feature usage, offering limited insight into when features transition from early experimentation to sustained adoption and how developers' motivations and constraints shape this process. Addressing these gaps can improve our understanding of Python code rejuvenation practices and support developers in making informed decisions about when and how to rejuvenate existing codebases.

To address this gap, we report the results of two empirical studies in this paper. The first investigates the evolution of open-source Python projects to identify adoption trends for 17 modern features introduced in Python 3, from version 3.0 to 3.12. This study includes features previously examined in the literature (such as type annotations) as well as newer features (such as structural pattern matching), for which we provide the first large-scale analysis of their adoption. To understand the motivations and challenges for rejuvenating Python source code, in the second study we review the pull-request comments available in the GitHub repository of the projects we consider in the first study, filtering out those that clearly relate motivations or impediments to rejuvenation efforts. Using thematic analysis, we identified common themes related to motivations (such as improving code comprehension, increasing safety and performance) and challenges (like compatibility issues, maintenance costs, and learning overhead) that might explain the practices for rejuvenating source code in Python. Some of our key findings include:

- **Improving code comprehension drives rejuvenation.** Qualitative findings indicate that enhancing code readability and understanding is the most frequent motivation (53.8% of codes).
- **Compatibility constraints hinder rejuvenation.** Qualitative results show that maintaining compatibility with earlier Python versions and dependent libraries is one of the most frequent challenges.
- **Early use is relatively fast, but generalized adoption lags behind.** While projects often exhibit initial use of new Python features within the first one to two years after release, the transition to generalized and sustained adoption typically unfolds over several years, particularly for features introduced in early Python 3 releases.

Our findings also reveal that some features are widely used in open-source projects—for instance, formatted string literals and type annotations in functions appear in almost 80% and 59% of projects, respectively. By contrast, newer or more specialized constructs, such as structural pattern matching, asynchronous comprehensions, and the `Exception-Group` and `except-*` features, are present in fewer than 5% of projects.

2. The History and Evolution of Python

The Python programming language was created by Guido van Rossum in the late 1980s, with development beginning in 1989 and the first official release—Python 1.0—arriving in 1994 [23, 24]. From the outset, Python distinguished itself through a clean, readable syntax designed to improve code clarity and developer productivity. The language also incorporated key features such as object-oriented programming (e.g., classes and inheritance) and structured exception handling, supporting the development of modular and maintainable software systems. Furthermore, its extensive standard library and smooth integration with other technologies rapidly broadened its adoption, particularly for scripting and automation tasks [37].

The release of Python 2.0 in October 2000 introduced several enhancements, including augmented assignments (e.g., `x += 1`), qualified-name imports, list comprehensions, and a more efficient memory management system, while also marking a period of increased community involvement [36, 37]. This era saw the emergence of key libraries and frameworks such as *NumPy*[1] and *Django*[2], which helped establish Python's strong presence in scientific computing,

[1] https://numpy.org/
[2] https://www.djangoproject.com/

data analysis, and web development. In December 2008, the introduction of Python 3.0 represented a pivotal moment in the language's evolution [37]. Unlike the evolution of other programming languages (e.g., Java), Python 3.0 did not maintain backward compatibility with the Python 2 series, requiring code modifications for migration. The transition aimed to establish a more consistent syntax, improved Unicode support, greater readability, and enhanced robustness. However, as noted by Malloy and Power [37], the lack of backward compatibility posed practical challenges for developers, who often chose to maintain support for Python 2 rather than fully embrace Python 3 features. As a result, many projects limited themselves to the common subset shared between both versions, delaying the widespread adoption of Python's latest advancements.

The Python community and the Python Software Foundation (PSF) have played a crucial role in the language's evolution by organizing events such as PyCon and ensuring a collaborative, transparent development process guided by Python Enhancement Proposals (PEPs) [9]. PEPs are official design documents and the primary mechanism for proposing Python features, collecting community input, and recording design decisions; authors are expected to build consensus around their proposals. Each PEP is discussed in a public, linked thread (e.g., on the Python Discourse via the Discussions-To header), where authors actively gather feedback that will be considered during the review. Standards-Track PEPs pair a precise specification with (ideally) a prototype/reference implementation, so review influences both API design and practical adoption. Under Python's current governance, final acceptance or rejection is decided by the elected Steering Council or a delegated decision-maker. Proposals progress through statuses such as Draft, Accepted, Final, Rejected, or Provisional, which determine when and how new features are introduced and stabilized.

2.1. Evolution of Python 3 Language Constructs

From Python 3.0 (2008) through Python 3.12 (2023), the language syntax evolved through a series of PEP-driven enhancements. In what follows, we outline this evolution chronologically, focusing on changes that directly modified the language grammar while noting releases without syntax updates. This structure helps situate individual features within the broader trajectory toward a more expressive and modern Python language. Python 3.0 (2008) introduced major structural changes to the language grammar. These include keyword-only arguments (PEP 3102), function annotations (PEP 3107), the new `nonlocal` statement (PEP 3104), and extended iterable unpacking (PEP 3132). Developers also gained more control over exception handling with exception context suppression (PEP 409). Together, these features made the language more expressive and flexible, encouraging modern programming practices [10].

Versions 3.1 and 3.2 did not introduce new language features [11, 12]. In turn, Python 3.3 (2012) added syntax for delegating to subgenerators via `yield from` (PEP 380), simplifying generator composition, reducing boilerplate, and enabling more efficient data-processing pipelines [13]. Python 3.4 improved the standard library and the import system, without major changes to the grammar [14]. By contrast, Python 3.5 (2015) introduced additional unpacking generalizations (PEP 448), native coroutine syntax with `async` and `await` (PEP 492) keywords, and the matrix multiplication operator `@` (PEP 465), symplifying the use of asynchronous programming and the implementation of advanced mathematical operations [15]. Python 3.6 (2016) enriched the language with formatted string literals (f-strings, PEPs 498 and 701), asynchronous comprehensions (PEP 530), variable annotations (PEP 526), and asynchronous generators (PEP 525). Collectively, these features improved support for modern applications, particularly in data science and asynchronous programming [16].

Python 3.7 mainly improved performance and internal features, without significant language changes [17]. In 2019, Python 3.8 introduced assignment expressions (PEP 572), also known as the "walrus operator," which enable assignments within conditional statements and loops [18]. In 2020, Python 3.9 allowed direct use of generics in type hints (PEP 585), allowing constructs such as `List[int]` without additional imports. [19]. The following year, Python 3.10 (2021) introduced structural pattern matching (PEPs 634, 635, 636), adding the `match/case` statement and substantially enriching the grammar [20]. In 2022, Python 3.11 introduced exception groups and the new `except*` block (PEP 654) to handle multiple concurrent exceptions, which is especially useful in asynchronous programming [21]. Finally, Python 3.12 (2023) added type parameter syntax (PEP 695), making it easier to define generic type aliases directly in the language, bringing Python closer to languages with advanced static typing [22].

2.2. Source Code Rejuvenation in Python

Source code rejuvenation refers to transforming legacy code to adopt modern language features and idioms [42]. It involves replacing outdated patterns and deprecated constructs with newer, more expressive, and efficient alternatives, ultimately making the code easier to read and maintain [33]. Whereas refactoring is often performed continuously

throughout a project's lifecycle, rejuvenation typically occurs gradually and is more prominent following major language updates.

In Python, this broader migration commonly centers on adopting "Pythonic" idioms that reflect the language's design philosophy. Pythonic idioms are community-accepted patterns and language features that embody Python's characteristic programming style [32, 37, 55]. In the literature, idioms are described as recommended programming styles and features, and in Python they are widely promoted for concise code and, in many cases, improved performance. This culture—often summarized by the "Zen of Python"—encourages developers to prefer idiomatic solutions.

Consequently, Python rejuvenation often entails replacing outdated constructs with more modern, "Pythonic" idioms. Migrating from legacy constructs (e.g., %-based string formatting) to modern features (e.g., f-strings) not only introduces new syntax but also aligns the codebase with contemporary idiomatic standards. For instance, Listing 1 shows how to introduce function annotations in a legacy code. Function annotations, introduced in Python 3.0, enable developers to document expected types directly in the function signature. This change enhances static analysis, tool support, and helps other developers understand code intent, moving the codebase towards better maintainability and compatibility with type checker tools [28].

Legacy style

```
def add(a, b):
    return a + b
```

Modern rejuvenated style

```
def add(a: int, b: int) -> int:
    return a + b
```

Listing 1: Adding function annotations for type hints

Listing 2 illustrates a source code rejuvenation to adopt structural pattern matching—a feature introduced in Python 3.10 that provides a more concise and expressive mechanism for conditional dispatch. By replacing multiple `if/elif` branches with `match/case`, the example makes the intent clearer: each case directly matches the structure of the tuple `cmd` argument and binds the operands accordingly. Moreover, the ordered cases allow exceptional conditions (e.g., division by zero) to be handled explicitly before the general rule, reducing boilerplate, lowering the risk of indexing errors, and improving overall maintainability [54].

In summary, these rejuvenation steps replace verbose, error-prone patterns with clearer idioms, reducing boilerplate and cognitive load while strengthening readability and maintainability.

3. Related Works

The adoption of modern programming language features has been widely studied in software engineering, particularly in the context of software evolution, maintenance, and source code rejuvenation. Prior research consistently shows that the availability of new language constructs does not imply their immediate or uniform adoption, and that adoption is shaped by both technical and social factors. Despite this broad agreement, important gaps remain regarding how adoption is operationalized, how it unfolds over long time spans, and how the transition from early experimentation to widespread use can be systematically identified.

Prior research examines adoption via the acceptance of code modifications, including refactorings or patches that introduce new features. Dantas et al. [7] show that simple transformations are more likely to be accepted than complex refactorings, indicating that cognitive effort and perceived risk affect maintenance choices. Similarly, proactive studies that submit pull requests introducing new language features, such as default methods in Java interfaces, demonstrate that adoption depends not only on technical benefits but also on project governance and maintainer preferences [29]. While these works offer helpful details about decision-making at the moment of change acceptance, they conceptualize adoption as a discrete event and therefore do not capture the gradual diffusion of language features within a codebase.

Another line of research measures adoption based on the prevalence of feature usage in existing systems. Studies on Java Generics show that, even after long periods of availability, usage remains concentrated in specific contexts, revealing partial and uneven adoption [40]. Research on dynamic languages, such as Smalltalk, analyzes the frequency of reflective and dynamic features to assess their impact on static analysis and tool support [4]. Although these studies demonstrate that real-world usage often diverges from language designers' expectations, they typically adopt static or aggregated perspectives and provide limited insight into how usage evolves.

Legacy style

```
def calc(cmd):
    if cmd[0] == "add":
        return cmd[1] + cmd[2]
    elif cmd[0] == "sub":
        return cmd[1] - cmd[2]
    elif cmd[0] == "div" and cmd[2] != 0:
        return cmd[1] / cmd[2]
    elif cmd[0] == "div":
        raise ZeroDivisionError("division by zero")
    else:
        raise ValueError("unknown command")
```

Modern rejuvenated style

```
def calc(cmd):
    match cmd:
        case ("add", x, y):
            return x + y
        case ("sub", x, y):
            return x - y
        case ("div", x, 0):
            raise ZeroDivisionError("division by zero")
        case ("div", x, y):
            return x / y
        case _:
            raise ValueError("unknown command")
```

Listing 2: Dispatching tuple-commands: from chained indexing to structural pattern matching.

Some studies focus specifically on the usage and distribution of Python language features in real-world systems. Peng et al. [41] propose PyScan, an AST-based approach to identify Python features across projects, providing an overview of how different constructs are used in practice. Similarly, large-scale empirical studies analyze the adoption of Python typing-related features, such as type annotations and static typing constructs, revealing uneven uptake across projects and domains [43, 25]. While these works provide valuable snapshots of feature usage, they primarily adopt static or cross-sectional perspectives and do not explicitly characterize how feature adoption evolves over long periods of time.

Complementary research investigates the perceived usefulness and readability of specific Python features through qualitative methods and controlled experiments. For instance, recent user studies on Python structural pattern matching show that developers often prefer this construct over traditional conditional branching in terms of readability and modifiability, although preferences may vary depending on the problem context [54]. Other work explores automated transformations that refactor legacy Python code to adopt structural pattern matching, demonstrating technical feasibility and potential readability gains [47]. However, these studies focus on isolated features and controlled settings, and do not examine whether such features achieve sustained and widespread adoption across open-source ecosystems.

More recent work analyzes adoption as a temporal process by tracking feature usage longitudinally in languages such as JavaScript, TypeScript, and C++. Large-scale empirical studies on JavaScript show that, despite frequent updates to toolchains, the effective use of modern language features often lags behind their introduction and varies substantially across projects [34]. Similarly, studies on TypeScript reveal that upgrading to newer language versions does not necessarily lead to the adoption of newly introduced features [50]. In the C++ ecosystem, analyses of the KDE community report a gradual increase in the use of modern features, often associated with explicit modernization efforts and tool support [33]. Together, these studies establish that adoption is gradual, heterogeneous, and context-dependent;

however, they largely rely on descriptive trend analysis or fixed thresholds, offering limited methodological support to identify when widespread adoption effectively begins.

In summary, prior work demonstrates that language feature adoption is uneven and influenced by multiple factors, but it provides limited means to determine, in a systematic and reproducible way, the onset of general adoption—particularly over long time spans and with sufficient granularity to capture diffusion within projects. This study advances the state of the art by proposing a statistically grounded operationalization of the onset of general adoption based on offline change-point detection applied to long-term time series of feature usage. By analyzing 16 years of Python evolution (2008–2024) and measuring adoption at the file level, our approach reveals that features within the same language may exhibit substantially different adoption regimes, including multi-year delays between their introduction and sustained growth.

Also, in contrast to prior work that either quantifies feature usage statically, evaluates individual language constructs in isolation, or relies on controlled qualitative studies, our work integrates long-term, feature-level adoption analysis with qualitative evidence extracted from real pull request discussions. This combined perspective enables a deeper understanding of not only when Python features begin to generalize, but also why developers choose to adopt, postpone, or avoid them in practice, thereby advancing empirical research on language evolution and source code rejuvenation in software engineering.

4. Study Settings

Our research focuses on understanding how Python developers adopt modern language features in open-source projects. We analyzed 17 Python features introduced from 2006 to 2022 that changed the language's constructs, such as formatted string literals (f-strings), function annotations, and pattern-matching. We specifically examined new syntactic elements of Python, deliberately excluding library-level changes. To guide our investigation, we address the following research questions:

(RQ1) To what extent do open-source Python repositories use modern features?

(RQ2) When did Python developers start adopting modern features?

(RQ3) What are the adoption trends for each modern Python feature?

(RQ4) What are the main motivations that lead developers to rejuvenate legacy Python code, and what are the main challenges that prevent them from doing so?

Answering the first research question helps us understand how much Python developers adopt modern language features and whether recent Python releases help develop Python applications. Python developers often prefer features that make their code simpler [41], so we expect wide adoption of easy-to-use syntax like f-strings. On the other hand, developers might not widely use features such as Structural Pattern Matching or Assignment Expression (walrus operator) because they already rely on familiar solutions.

Understanding these differences shows why it is important to study how open-source Python projects use each modern feature. The second research question examines when developers begin using new Python features and whether they feel confident moving away from familiar practices toward newer ones. By examining the third research question, we aim to identify if there is a growing trend in adopting modern Python features among open-source projects. The fourth question focuses on the motivations for Python code rejuvenation and examines the challenges developers encounter that hinder them from rejuvenating legacy systems.

Our research was organized into two distinct studies. The initial study utilized descriptive statistics and time series analysis to examine the adoption and trends of modern Python features. Initially, we outline the criteria used to select the 17 contemporary features of the Python language for examination. We detail the procedure employed to select the open-source repositories that formed our dataset. This section outlines the study's methodology, specifying the tools and methods utilized for data collection and analysis in addressing research questions RQ1, RQ2, and RQ3. Next, we employed thematic analysis [5] to elucidate the motivations and challenges associated with Python code rejuvenation. In the second study, we gathered data from code review comments to address our final research question (RQ4). We have made our replication package [44] accessible to the public. It encompasses our datasets, the scripts for analysis, and all the materials to replicate the findings of our study.

Table 1
Python Features, PEPs, Versions, Release Date, and Examples (ordered by Python version and PEP number).

Feature	Python Version	PEP(s)	Release Date	Simple Example
Keyword-only Arguments	3.0	3102	2008-12	`def func(*, arg): pass`
`nonlocal` Statement	3.0	3104	2008-12	`def outer(): nonlocal x; x = 10`
Function Annotations	3.0	3107	2008-12	`def func(x: int) -> int: return x`
Extended Iterable Unpacking	3.0	3132	2008-12	`a, *rest, b = range(5)`
Syntax for Delegating to Subgenerator (Yield From)	3.3	380	2012-09	`yield from subgenerator()`
Suppressing Exception Context	3.3	409	2012-09	`try: 1/0 except ZeroDivisionError: raise ValueError('Invalid') from None`
Additional Unpacking Generalizations	3.5	448	2015-09	`{**{'a':1}, **{'b':2}}`
Matrix Multiplication Operator (`@`)	3.5	465	2015-09	`a @ b`
Async and Await Syntax	3.5	492	2015-09	`async def foo(): await bar()`
Formatted String Literals (f-strings)	3.6	498,701	2016-12	`f"Hello, {name}"`
Asynchronous Generators	3.6	525	2016-12	`async def gen(): yield x`
Variable Annotations	3.6	526	2016-12	`x: int = 10`
Asynchronous Comprehensions	3.6	530	2016-12	`[i async for i in aiter()]`
Assignment Expressions	3.8	572	2019-10	`if (squared := num ** 2) > 10`
Type Annotations Generics	3.9	585	2020-10	`x: list[int] = [1, 2, 3]`
Structural Pattern Matching	3.10	634,635,636	2021-10	`match x: case 0: print("zero")`
Exception Groups and `except*`	3.11	654	2022-10	`try: raise ExceptionGroup('', [ValueError()]) except* ValueError: pass`
Type Parameter Syntax	3.12	695	2023-10	`type ListOrSet[T] = list[T] | set[T]`

4.1. Features Selection

Table 1 lists the modern Python features analyzed in our study. The table includes the Python version that introduced each feature, the associated Python Enhancement Proposals (PEPs), their official release dates, and simple code examples illustrating their use. Although our selection does not cover four out of the 21 modern Python 3.x features that modified the language syntax, it still provides a comprehensive view of Python's evolution.

We excluded certain Python features from our analysis due to limitations in detecting their usage through static analysis alone. Specifically, we removed "Underscores in Numeric Literals" because the Python AST library requires converting numeric nodes back into source code strings, leading to frequent exceptions and false positives during analysis. Similarly, we excluded "Postponed Evaluation of Type Annotations" as this feature delays evaluating type annotations until runtime, making it difficult to detect reliably through static analysis.

Additionally, we omitted "Relaxed Grammar Restrictions on Decorators" since decorators have existed since Python 2, complicating the identification of differences introduced exclusively from Python 3 onward. Lastly, we did not consider the "Union Operators for dict" feature, as static analysis cannot reliably track cases where variables initially defined as dictionaries might later reference different types at runtime, leading to potential inaccuracies in detection.

4.2. Dataset Selection

Our method for selecting projects was inspired by the approach of Zhao et al. [56]. First, we searched GitHub repositories that use Python as their main language, created before 2012, and still active in 2024. We chose projects at least 10 years old to examine mature and legacy systems, as older systems are more likely to face challenges related to outdated technologies and coding practices [46]. This decision helps us study projects that transitioned from Python 2 to Python 3, which may have required source code rejuvenation to stay relevant and active.

We focused our analysis on active repositories, identified using the SEART tool [6]. Initially, we gathered 1,199 repositories and excluded forks, tutorials, and repositories with fewer than 1,000 commits, reducing the list to 460 projects. Finally, we excluded 36 repositories due to data-collection issues. The resulting dataset in Table 2 comprises 424 Python projects that together amount to approximately 56M lines of code, 1.45M stars, and 357K forks, with a median of 753 stars, 219 forks, and 69 contributors per project, indicating a set of mature and actively maintained repositories.

The dataset covers a broad range of domains within the Python ecosystem, including data science and machine learning, web development, scientific computing, developer tooling, and general-purpose libraries. Most projects are libraries or frameworks intended for reuse, alongside a smaller number of end-user applications. Well-known and widely adopted projects such as Flask, scikit-learn, Requests, and pandas illustrate the prominence of domains where language evolution and long-term maintenance pressures are particularly relevant.

Table 2
Dataset characterization of the 424 analyzed Python projects.

Characteristic	Value
Projects	424
Total lines of code (LOC)	∼56M
Median LOC per project	35,561
Median contributors per project	69
Median repository size (files)	15,275
Total stars	∼1.45M
Median stars per project	753
Total forks	∼357K
Median forks per project	219
Project types	Libraries, frameworks, applications
Main domains	Data science, ML, web, DevOps, utilities

4.3. Data Collection and Analysis Procedures

In the first study, we analyzed the source code history of 424 GitHub projects to investigate the utilization and adoption patterns of modern Python features. To accomplish this, we built PyMiner, a tool intended to analyze the source code history of selected projects and collect data regarding specific Python features. PyMiner uses Python's built-in AST library to parse Python files, generate abstract syntax trees (AST), and traverse the Python code using visitors.

Our analysis pipeline uses PyMiner to inspect Python projects from January 1, 2008, to December 31, 2024. First, we compiled a list of Python repositories from GitHub using the SEART tool [6], then cloned these repositories locally. PyMiner then reviews each repository's commit history at 30-day intervals, reducing the total number of commits to be analyzed. For each selected commit, PyMiner retrieves the project state and traverses Python files to collect information on modern features. The results for each project are stored in individual CSV files, which we later combine into a single file for analysis.

We used this consolidated dataset to address our first three research questions (RQ1, RQ2, and RQ3). For RQ1, we conducted a descriptive analysis to measure how often each feature was used and calculated the median number of files per revision containing these features. To answer RQ2, we identified when each modern Python feature first appeared in the repositories. Lastly, for RQ3, we applied a time-series analysis, using the timing and the count of files with occurrences data of each feature to model their adoption trends. All analyses were carried out using Python scripts.

In the second study, to address RQ4, we developed a Python-based infrastructure that leverages the GitHub API to scrape comments from pull requests, automating the data collection process. This infrastructure iterates over each pull request (PR) in the repositories in our dataset and retrieves the related PR comments. In this study, we mined 395,702 pull request comments and populated a database for analysis.

Next, we employed a three-stage filtering process to identify pull request comments pertinent to source code rejuvenation in Python projects. Initially, a large corpus of comments (3,565) was mined using a curated set of keywords derived from Python language features introduced in recent PEPs, including "asynchronous comprehensions", "formatted string literals (f-strings)", "structural pattern matching", and "assignment expressions", as well as specific Python Enhancement Proposals (PEPs) like PEP 492, PEP 498, and PEP 634. Additionally, the query incorporated common variations and related terms—such as "async def", "fstring formatting", "match-case statements", and "walrus operator"—to capture a broader range of relevant discussions.

Subsequently, the filtered comments underwent a second-level screening using a simple prompt-based classification with ChatGPT (gpt-4o). The prompt used in this study (available in the Zenodo repository [44], file *source_code_rejuvenation_search.py*) asked the model to decide whether each comment pertained to source code rejuvenation—defined as updating legacy code with modern Python idioms—and to provide a brief summary when relevant.

To assess whether the LLM-based filtering stage systematically excluded relevant discussions, we conducted a manual validation of a random sample of comments classified as unrelated. We randomly selected 100 comments from the approximately 3,000 comments rejected by the LLM and manually inspected them to identify potential false negatives. The majority of the sampled comments were clearly unrelated to source code rejuvenation, focusing instead

on localized design decisions, documentation concerns, or implementation details without reference to legacy code or language evolution. For example, a comment discussing whether expressions of default arguments and annotations should be collapsed in documentation—"This change has pros and cons… there might be a case for collapsing the expressions of default arguments and keeping the expressions of annotations"—was classified as irrelevant because it concerns how existing language features are represented and implemented, rather than any effort to rejuvenate legacy code.

Only a small subset of the sampled comments (approximately 5%) could be considered potential false negatives. These cases typically involved implicit or weakly articulated signals related to modern language features that did not explicitly frame their use as part of a rejuvenation effort. For instance, a comment such as "Are f-strings supported in lambdas?" reflects a clarification or doubt about the applicability of a modern Python feature, but does not express a motivation to replace legacy constructs, nor does it describe challenges or impediments to modernization. Overall, this validation indicates that the LLM-based filtering did not systematically exclude rejuvenation-related discussions, while substantially reducing the manual analysis effort.

This lightweight step complements the keyword filter and scales semantic screening to large volumes while reducing manual effort, consistent with evidence that prompt-based LLMs can perform effective text classification in low-resource settings and capture task-relevant cues via prompts [35, 52]. This process yielded 494 pull request comments, which were then manually examined in a third phase using thematic analysis. The thematic analysis was conducted manually by the authors to ensure interpretive depth.

Following, we adopted a thematic analysis approach, following Saldaña's guidelines for qualitative data analysis [49]. We began with open coding, an initial phase in which relevant text segments are identified and associated with descriptive labels or tags that reflect emerging concepts and themes within the data. Next, we proceed to classification, systematically grouping the initial codes into broader categories based on shared characteristics. Throughout this process, we continuously refined and revisited both codes and categories as new data and insights emerged.

The data analysis procedure was conducted in two cycles to verify the relevance and accuracy of the selected pull request comments. Three authors collaboratively conducted the thematic coding process, with the first having prior experience in qualitative software engineering research. During the initial cycle, the first and third authors of this paper examined the 494 code review comments identified through the textual query, distinguishing those pertinent to motivations and challenges for rejuvenation efforts from those that merely cited the name of modern Python features. From this process, 77 quotes from comments were identified as potentially relevant.

During the second cycle, the second and third authors conducted independent reviews of each other's work, noting areas of agreement and disagreement regarding the coding. In cases of disagreement, the authors, along with an additional author, discussed the coding until a consensus was reached. The first author of this paper categorized the codes into thematic groups. In Section 6, we describe and present our findings.

5. First Study: Prevalence and Adoption Trends of Modern Python 3 Features

This section reports a quantitative analysis of the adoption of modern *Python* features. By traversing the complete revision history of each repository we counted every occurrence of the 17 features listed in Table 3.

5.1. Prevalence Assessment

Table 3 highlights that *function annotations* stand out as the most prevalent modern Python feature: they appear 306,196 times, and are present in nearly 59% of the analyzed projects. This extensive adoption reflects their central role in enabling static type checking and facilitating gradual typing through tools such as `mypy`. While formatted string literals (f-strings) remain highly popular—appearing 124,312 times and present in 79% of analyzed projects —the sheer volume of function annotations underscores their fundamental importance in the evolution of Python's type system and modern development practices. In total (considering the last commit of all projects), our dataset contains 1,144,365 traditional (non-annotated) functions and 135,171 annotated functions, meaning that only about 10% of all functions include at least one type annotation. This disproportion illustrates that, although annotations are widespread across projects, their internal coverage remains limited—supporting the findings of Grazia and Pradel [25] that annotations are still sparsely distributed within individual codebases. Similarly, *variable annotations* (84,853 occurrences, 52.6% of projects) follow a comparable pattern of gradual integration since Python 3.6.

Table 3
Project–level adoption of modern Python features. Δ Months = (First Occurrence) − (Release month from Table 1); negative values indicate pre-release (alpha/beta) occurrences.

Feature	Total of Occurrences (#)	Projects Adoption (%)	First Occurrence	Δ Months
Formatted String Literals (f-strings)	124312	79.71	2016-05	-7
Function Annotations	306196	58.96	2012-05	41
Variable Annotation	84853	52.59	2017-01	1
Keyword-only Arguments	49622	47.16	2011-11	35
Additional Unpacking Generalizations	4029	42.92	2015-09	0
Yield From Expression	2093	36.08	2012-05	-4
Assignment Expression	2772	23.58	2019-12	2
Nonlocal Statements	586	20.28	2010-12	24
Coroutines (async and await syntax)	39496	19.57	2015-12	3
Extended Iterable Unpacking	416	19.33	2009-12	12
Suppressing Exception Context	933	17.68	2015-03	30
Matrix Multiplication	2304	8.25	2018-05	32
Structural Pattern Matching	2054	4.48	2021-05	-5
Asynchronous Comprehensions	359	2.59	2019-11	35
ExceptionGroup	11	0.70	2022-05	-5
Type Parameter Syntax	5	0.47	2023-07	-3
Except (*)	1	0.23	2022-05	-5

Keyword-only arguments and the generalised unpacking operators also display moderate diffusion, each reaching more than 40% of projects. In contrast, several features have yet to gain traction. Exception groups (PEP 654) and the type-parameter syntax (PEP 695) appear in fewer than 1% of projects, while structural pattern matching (Python 3.10) is present in only 4.5% of repositories. Conversely, we did not observe evidence of the use of Asynchronous Generators that were in our initial subset.

Table 4 presents summary statistics for the occurrences of various Python features in GitHub projects. Each feature is listed with its median, mean, standard deviation (std), maximum (max), and minimum (min) number of occurrences across the analyzed repositories. Feature usage varies considerably across projects. For example, Formatted String Literals (f-strings) exhibit a median of 31 occurrences per project, while Variable Annotations and Function Annotations present medians of 1 and 3, respectively. In contrast, most other features display a median of zero. The distribution is also highly uneven, as reflected by the large standard deviations. For instance, in the *sentry* project, Function Annotations reach a maximum of 26,605. We also observe a similar lack of uniformity in the usage of Variable Annotations and f-strings across projects.

Answer to RQ1

Our results show that modern Python features are adopted unevenly. *Function annotations* dominate in absolute frequency with more than 306,000 occurrences, while other widely used constructs include *f-strings*, present in nearly 80% of projects. Several features, such as *variable annotations*, *keyword-only arguments*, and *additional unpacking generalizations*, appear in around half of the projects, whereas recent constructs like *structural pattern matching* and *exception groups* remain rare.

We also compare each feature's official release date in the Python language with the date of its first occurrence in our dataset. In our analysis, negative Δ values indicate the adoption of language features from pre-release (alpha or beta) versions of Python prior to their official stable release. Interestingly, some features appear earlier than their official final release dates. The first case is *formatted string literals (f-strings)*, which officially became part of Python 3.6 in December 2016, but whose first appearance in our dataset was in May 2016. This aligns with the release of Python 3.6.0a1, the first alpha version, published on May 17, 2016 (Python 3.6.0a1, Python 3.6 Release Schedule). The second case is the *yield from expression*, formally introduced with Python 3.3 in September 2012, but observed in our dataset as early as May 2012. This corresponds to the alpha releases of Python 3.3, such as 3.3.0a3 (May 1, 2012) and 3.3.0a4 (May 31, 2012), which already included support for PEP 380 (PEP 380, Python 3.3.0, Python 3.3 Release Schedule). Furthermore, other features that appear before their release include structural pattern matching,

Table 4
Summary statistics for the occurrence of Python feature in GitHub projects.

Feature	Min	Median	Mean	Max	Std
Function Annotations	0	3	361.08	26605	1452.81
Formatted String Literals (f-strings)	0	31	293.19	13188	966.48
Variable Annotation	0	1	200.13	24599	1332.99
Keyword-only Arguments	0	0	58.52	3497	291.56
Coroutines (async and await syntax)	0	0	23.29	4170	213.73
Assignment Expression	0	0	6.54	229	27.52
Matrix Multiplication	0	0	5.43	924	49.69
Yield From Expression	0	0	4.94	289	19.80
Suppressing Exception Context	0	0	2.20	265	15.09
Additional Unpacking Generalizations	0	0	1.58	523	13.39
Nonlocal Statements	0	0	1.38	64	5.88
Extended Iterable Unpacking	0	0	0.98	45	4.07
Structural Pattern Matching	0	0	0.54	345	9.25
ExceptionGroup	0	0	0.03	5	0.33
Type Parameter Syntax	0	0	0.00	3	0.09
Except (*)	0	0	0.00	1	0.05
Asynchronous Comprehensions	0	0	0.28	248	7.10

ExceptionGroup/except*, and Type Parameter Syntax. These findings show that developers sometimes adopt new language features directly from alpha or beta versions, well before they are available in stable releases.

When comparing all the features in our dataset, we observe different timelines between their official release dates and their first appearance. *Function annotations* were released in December 2008 with Python 3.0, but the first use in our dataset occurred in May 2012. *Keyword-only arguments*, also introduced in December 2008, first appeared in November 2011. *Nonlocal statements* were released in December 2008 and first appeared in December 2010, while *extended iterable unpacking*, from the same December 2008 release, appeared in December 2009. *Suppressing exception context* was added in September 2012 with Python 3.3, and first appeared in March 2015. In contrast, other features appeared soon after their official releases: *variable annotations* released in December 2016 first appeared in January 2017, *assignment expressions* released in October 2019 appeared in December 2019, and *coroutines (async/await)* released in September 2015 appeared in December 2015. Some features were observed immediately: *additional unpacking generalizations*, released in September 2015, first appeared in the same month.

Answer to RQ2

Our results show that modern Python features have been adopted at very different rates. Some features (e.g., variable annotations, assignment expressions, async) were adopted quickly, while others (e.g., function annotations, keyword-only arguments, `nonlocal`) saw slower adoption. We also observe that certain features—including f-strings, `yield from`, structural pattern matching, and ExceptionGroup/`except*`—appeared in projects even during alpha or beta releases, indicating early experimentation among some developers.

5.2. Adoption Trend Assessment

Although a feature may appear in repositories shortly after its official release, substantial growth in adoption often occurs later. Here, we analyze the delay between initial use and the onset of adoption growth for each feature. We analyze monthly commit time series spanning from 2008 to 2024 to identify adoption patterns. For each month in this period, and for each project and feature, we counted the number of Python source files containing at least one occurrence of the feature, considering all Python files present in the commit analyzed for that month. These per-project counts were then aggregated across all projects to obtain a monthly, dataset-level time series of file-level usage for each feature. Our analysis considers only features present in at least 10% of the projects in the dataset.

To identify the onset of broader adoption of Python language features, we analyze monthly time series of feature usage using offline change point detection. Specifically, we apply the PELT algorithm with an ℓ_2 cost function as implemented in the `ruptures` library [53] to identify candidate structural breakpoints in each time series. Prior to

detection, raw occurrence counts are transformed using $\log(1 + x)$ to mitigate heteroscedasticity commonly observed in count-based temporal data. Change points are detected by minimizing a penalized segmentation objective, where the penalty is deterministically derived from the series length and variance, ensuring reproducibility and avoiding manual parameter tuning.

For each candidate change point b, we assess whether it corresponds to the onset of generalized adoption by comparing robust central tendencies before and after b within a symmetric temporal window H. Specifically, we compute the median feature usage in the pre-onset window $[b-H, b)$ and the post-onset window $[b, b+H)$, and define an adoption score as:

$$\text{score}(b) = \frac{\text{median}\big(y_{b:b+H}\big) - \text{median}\big(y_{b-H:b}\big)}{\max(1.4826 \cdot \text{MAD}(y),\ \varepsilon)},$$

The scale is given by $1.4826 \cdot \text{MAD}(y)$, where $\text{MAD}(y) = \text{median}(|y - \text{median}(y)|)$ denotes the Median Absolute Deviation, a nonparametric and outlier-resistant measure of dispersion [51, 45, 26]. The constant 1.4826 ensures consistency with the standard deviation under normality. To ensure numerical stability and avoid inflated scores in near-constant series, the denominator is lower-bounded by a small constant $\varepsilon = 10^{-3}$. This formulation yields a robust, dimensionless effect size that captures sustained shifts in feature usage while mitigating the influence of noise, outliers, and early sporadic adoption.

When comparing the release of each feature with the onset of generalized adoption trends, we observe substantial delays for features introduced in Python 3.0. *Keyword-only arguments*, released in December 2008, reached their adoption onset only in February 2012, corresponding to a delay of approximately 3.2 years. Similarly, the *nonlocal* statement, also released in December 2008, exhibited adoption onset in October 2012, after about 3.8 years.

Function annotations, despite being introduced in Python 3.0, entered a generalized growth phase only in October 2016, resulting in a markedly longer delay of nearly 7.8 years. *Extended iterable unpacking* followed a comparable long-term pattern, with adoption onset observed in September 2019, almost 10.8 years after its release. This slow and staggered adoption aligns with the findings of Di Grazia and Pradel [25], who reported that, despite steady growth since 2017, the overall coverage of function annotations remained low—only about 8% of function arguments and return types were annotated in 2021.

Together, these results indicate that although several Python 3.0 features are present in a non-trivial fraction of projects (e.g., function annotations appear in nearly 59% of projects in our dataset), their integration into individual codebases unfolds as a gradual rejuvenation process, characterized by incremental and localized adoption over extended periods.

For Python 3.3, adoption delays are noticeably shorter. The *yield from* expression, released in September 2012, reached its adoption onset by May 2014, approximately 1.7 years later. Likewise, *suppressing exception context*, introduced in the same release, entered its growth phase in January 2020, reflecting a longer but still moderate delay of about 7.3 years. Python 3.5 features display heterogeneous adoption dynamics. *Coroutines* (`async/await`), released in September 2015, reached generalized adoption by September 2016, after only 1 year. In contrast, *additional unpacking generalizations* required around 2.6 years, with growth beginning in April 2018.

Features introduced in Python 3.6 generally exhibit faster adoption. *Formatted string literals (f-strings)*, released in December 2016, reached widespread growth by January 2019 (about 2.1 years), while *variable annotations* followed closely, with adoption onset in May 2018 (approximately 1.4 years). Finally, among later features, *assignment expressions* (Python 3.8) show one of the shortest delays: released in October 2019, they reached adoption onset by January 2021, after only 1.3 years.

Several inflection points observed in our adoption curves temporally align with major ecosystem-level events in the Python community. For example, Python 2 officially reached its end-of-life on January 1, 2020, after which it no longer received security patches or official support, effectively removing Python 2 as a viable long-term platform and increasing pressure for migration to Python 3.[3] This transition was reinforced by key ecosystem decisions, including the release of Django 2.0 in December 2017, which dropped support for Python 2 entirely and required Python 3, as well as the coordinated phase-out of Python 2.7 support by major scientific libraries such as NumPy, aligned with the Python core team's deprecation timeline.[4][5] In parallel, the maturation of the Python tooling ecosystem between 2016

[3] https://www.python.org/doc/sunset-python-2/

[4] https://docs.djangoproject.com/en/2.0/releases/2.0/

[5] https://numpy.org/neps/nep-0029-deprecation_policy.html

and 2019 — particularly the emergence and consolidation of static type checking tools such as `mypy`, which builds on function annotations — lowered the practical cost of adopting modern language features.[6] While these events do not establish causality, their temporal proximity to the observed acceleration in feature adoption suggests that external, ecosystem-driven pressures played an important contextual role in reducing long-standing inertia and enabling broader source code rejuvenation.

Answer to RQ3

Our results indicate that modern Python features require, on average, approximately **4 years** to reach the onset of generalized adoption, although this process exhibits substantial variability across language versions and feature types. Adoption delays range from as little as **1–1.5 years** for features such as *coroutines* (`async/await`), *variable annotations*, and *assignment expressions*, to nearly **11 years** for *extended iterable unpacking*, a feature introduced in Python 3.0. Early Python 3.0 features consistently display longer maturation periods—often exceeding **7 years**—suggesting prolonged inertia in legacy codebases. In contrast, features introduced from Python 3.5 onward typically reach generalized adoption within **1–3 years**, revealing a clear acceleration in uptake over Python's evolution. The temporal alignment of these adoption inflection points with ecosystem-level events, such as the Python 2 end-of-life and the consolidation of Python 3–only frameworks and tooling, suggests that generalized adoption is best understood as a socio-technical rejuvenation process shaped not only by language design, but also by external pressures that reduce postponement and facilitate modernization.

6. Second Study: Motivations and Challenges for Rejuvenating Python Code

In order to address our fourth research question, we conducted a manual analysis of a subset of code review comments from the repositories examined in our first study to gain insight into the motivations and challenges developers encounter when rejuvenating Python code. In the scope of this research, code review is a collaborative software engineering practice where developers systematically examine each other's code to improve quality, correctness, and maintainability [39, 3, 31]. Each pull request comment was treated as an individual review unit, and the excerpts reported in this paper correspond to substrings of these comments selected for their relevance to rejuvenation discussions. Themes were derived inductively based on conceptual relevance rather than frequency, although the number of codes per category is reported for transparency.

6.1. Motivations

Our qualitative analysis of developer discussions in pull requests identified a set of recurring motivations that drive Python code rejuvenation efforts. These motivations reflect not only the pursuit of cleaner and more maintainable code, but also the desire to adopt modern language features that enhance safety, and performance. Each motivation was grouped into a specific subcategory, capturing a distinct aspect of why contributors decide to conduct source code rejuvenation efforts. In the following paragraphs, we describe each subcategory, report its frequency in the analyzed dataset, and illustrate it with representative comments from developers, highlighting both the feature being adopted and the intended benefit for the project.

Improve code comprehension This subcategory refers to changes aimed at making code easier to read and understand, especially during reviews and maintenance. It appears 37 times in the analyzed pull request comments, capturing its role in shaping how source code rejuvenation unfolds in Python open-source projects. For example: *"Consider adding type hints for better code clarity and to leverage static type checking."* (`materialsproject/pymatgen • PR #3758`), *"Use f-strings for more readable and efficient string formatting."* (`materialsproject/pymatgen • PR #3785`), and *"I much prefer the f-string equivalent as much easier to read."* (`pylons/ pyramid • PR #3701`). These examples show how contributors use modern constructs such as f-strings and type annotations to make the intent of the code more clear, readable and easier to read. Improving code comprehension reduces the mental effort required for understanding logic during reviews and maintenance, allowing both current and future developers to reason about the code with less cognitive load.

[6] `https://mypy-lang.org/`

Improve safety This subcategory focuses on changes that reduce the likelihood of runtime errors and improve the robustness of the system. It appears 7 times in the analyzed pull request comments. For example: *"This function should be `async def` and the execute commands should `await`. This way, if someone tries to await a function that raises an error, a second error won't pop up saying you tried to await an exception."* (`redis/redis-py • PR #2099`) and *"Maybe these should all become keyword-only arguments for safety, meaning an error if they try to pass the `slice_order` in by position, breaking backward compatibility and warning them of the problem."* (`nipy/nipy • PR #266`). These comments reflect the importance of aligning code with safer execution patterns, including keyword-only arguments to prevent accidental misuse, and asynchronous functions to avoid blocking I/O. Such measures reduce the risk of subtle runtime errors and make systems more robust.

Improve performance This subcategory focuses on rejuvenation efforts motivated by potential gains in execution speed or resource efficiency. It appears a time in the analyzed pull request comments. For example: *"f-strings and percent formatting are marginally faster than the ".format() too."* (`pylons/pyramid • PR #3701`). These observations illustrate that even marginal gains from features like `f-strings` and % formatting can matter in performance-critical paths. Contributors weigh these potential benefits against clarity and maintainability, reflecting a pragmatic approach to conduct rejuvenation efforts in the Python community.

6.2. Challenges

Each challenge was grouped into a subcategory that captures a distinct obstacle faced by developers. In the following paragraphs, we present each subcategory and illustrate it with representative comments, highlighting the tension between adopting modern Python features and maintaining project stability.

Compatibility issue This subcategory refers to constraints from supporting specific Python versions or environments, which can delay or block modern features. It appears **5** times in the analyzed pull request comments. For example: *"X | Y syntax for unions requires Python 3.10."* (`jazzband/tablib • PR #516`) and *"But until Python 3.6 support is dropped, we can only add annotations which are Python 3.5 compatible."* (`twisted/twisted • PR #1294`). These cases show how maintainers must align rejuvenation efforts with version support policies, often deferring the use of features like f-strings or advanced type annotations until older versions are no longer supported, ensuring the code remains functional across its intended deployment range.

Maintenance cost This subcategory refers to the increasing workload and complexity that modern language features can impose on testing and maintenance processes. It appears two times in the analyzed pull request comments. For example: *"If we're going to be eventually using keyword-only arguments in many places in the library, the complete tests could become onerous."* (`networkx/networkx • PR #6267`) and *"Thank you, @contributor! I implemented this, but I agree that this won't scale well in general if keyword-only arguments are more widely used."* (`networkx/networkx • PR #6267`). These examples show that the introduction of features such as keyword-only arguments can increase the combinatorial burden of testing and the overall maintenance effort. Developers often delay widespread adoption of such features until their usage patterns, testing strategies, and supporting infrastructure can handle the added complexity efficiently.

Learning overhead This subcategory reflects the perception that certain rejuvenation efforts may introduce complexity that outweighs their immediate benefits, particularly for contributors less familiar with newer language constructs. It appears 1 time in the analyzed pull request comment. In the context of PEP 465, which proposed adding the `@` and `@=` operators for matrix multiplication, a contributor expressed: *"[Q1] It provides no balanced view of the status quo. [Q2] You need to convince those outside the NumPy community that the addition of two new operators is justified for all Python users, most of whom are not users of linear algebra. [Q3] It adds to the Python learning curve with no benefit to most and very limited benefit to the users of linear algebra."* (`numpy/numpy • PR #4351`). The developer suggest that even well-intended syntax can be seen as overhead, which can slow the adoption of rejuvenation efforts.

Answer to RQ4

Python source code rejuvenation is mainly driven by goals such as improving code comprehension, safety, and performance often through modern features like `f`-strings, type annotations, keyword-only arguments, `async`/`await` patterns, and advanced typing annotations. Key challenges include version compatibility issues, high maintenance costs, and the learning curve of new features. Effective rejuvenation balances these benefits and obstacles to rejuvenate the source code without compromising stability or accessibility.

7. Implications

Our findings highlight several important aspects regarding the adoption of modern Python features across open-source projects, combining large-scale quantitative evidence with qualitative insights derived from developers' discussions in pull request comments.

For **RQ1**, the results reveal substantial variability in the adoption rates of modern Python features. Constructs that provide immediate and tangible benefits to code readability and comprehension—such as *formatted string literals (f-strings)*, *function annotations*, and *variable annotations*—achieved broad adoption across projects. In contrast, features that introduce new programming paradigms or require more substantial conceptual shifts, including *structural pattern matching*, *asynchronous comprehensions*, and *exception groups (except*)*, exhibit slower and more fragmented adoption. Qualitative evidence from pull request discussions provides contextual explanations for these patterns, as some developers explicitly mention concerns related to learning effort, refactoring scope, and uncertainty about long-term benefits when evaluating such features.

Regarding **RQ2**, our analysis shows that developers typically begin experimenting with new Python features within one to two years after their official release. Evidence of feature usage often appears shortly after release, indicating that initial adoption tends to occur relatively early, even if usage remains limited in scope at this stage. This pattern is observed across multiple features and suggests a general willingness within the Python ecosystem to explore new language constructs soon after they become available.

In addition, some features appear in repositories even before their stable release, including *f-strings*, *yield from*, *structural pattern matching*, and *ExceptionGroup/except**. Qualitative observations indicate that such early experimentation is typically confined to a small subset of projects and is often accompanied by discussions related to feature maturity, documentation, or potential impact on existing code, reflecting cautious exploration rather than immediate, widespread use.

The analysis of adoption trends (**RQ3**) further shows that, although initial usage often emerges shortly after release, widespread and stable adoption typically takes several years to materialize. On average, modern Python features require approximately **four years** to transition from early experimentation to sustained growth across projects. Lightweight syntactic enhancements tend to diffuse more rapidly, whereas features that demand broader architectural changes or deeper language understanding exhibit longer maturation periods. Developers' comments occasionally reference compatibility constraints—particularly support for older Python versions—as a factor that may delay broader adoption.

Finally, with respect to **RQ4**, our qualitative analysis reveals that source code rejuvenation in Python is primarily motivated by the desire to improve code comprehension, followed by concerns related to safety and correctness. Developers commonly adopt features such as `f`-strings, type and variable annotations, and keyword-only arguments to enhance clarity and reduce ambiguity. At the same time, rejuvenation efforts face notable impediments, including compatibility with legacy environments, the maintenance cost of large-scale changes, and the learning overhead associated with newer language constructs. These qualitative findings complement the quantitative results by providing contextual explanations for why certain features, in many cases, exhibit prolonged adoption delays despite their potential benefits.

Overall, our results suggest that, in open-source settings, successful rejuvenation strategies require balancing the immediate gains of adopting modern language features with the practical constraints imposed by project scale, version support policies, and developer expertise. By combining longitudinal adoption analysis with qualitative insights from real-world developer discussions, this study provides a more nuanced understanding of how and why Python features transition from experimentation to general use in open-source software ecosystems.

8. Threats to Validity

Our study has severeal limitations. A few large or highly active repositories could disproportionately influence aggregate trends—for example, the `sentry` project shows unusually high counts of `f`-strings and annotations. To reduce this bias, we analyzed the *monthly number of files with at least one use* of each feature instead of raw counts and applied smoothing to limit short-term spikes. Although large-scale adoptions within a single project may still affect the curves, they accurately reflect real repository-wide changes. RQ1 and RQ2 provide complementary project-level views that are less sensitive to these effects, and future work could further normalize by project size or count projects using each feature.

Our operationalization of *generalized adoption onset* relies on detecting sustained shifts in the central tendency of monthly feature usage time series. To mitigate sensitivity to outliers, early sporadic usage, and heavy-tailed count distributions, we employ robust statistics throughout the analysis. In particular, adoption strength is quantified using the difference between pre- and post-onset medians, normalized by the Median Absolute Deviation (MAD), a well-established robust estimator of scale [51, 45, 26]. The MAD is rescaled by the constant 1.4826 to ensure consistency with the standard deviation under normality. This choice reduces the risk that isolated spikes or noise-driven fluctuations are misinterpreted as meaningful adoption events.

Our qualitative analysis relies on an LLM-based filtering stage to reduce the volume of pull request comments requiring manual inspection, which constitutes a potential threat to construct validity. Although large language models are effective in semantic filtering tasks, their behavior is not deterministic and may lead to false negatives, particularly when relevant discussions are implicit, brief, or highly context-dependent. Importantly, the goal of this study is not to construct or evaluate a robust classification model for source code rejuvenation discussions. Instead, we use the LLM as a pragmatic tool to reduce manual effort while preserving semantically rich discussions that explicitly address motivations, challenges, or benefits of modernizing legacy Python code. Our manual validation of a random sample of rejected comments indicates that only a small fraction were potential false negatives, suggesting that the filtering did not systematically bias the qualitative findings. Nevertheless, some relevant discussions may have been excluded, and our results should be interpreted as capturing prominent and explicit rejuvenation-related rationales rather than an exhaustive set of all possible mentions.

Regarding external validity, our analysis involved 424 out of 343,878 Python projects (approximately 0.12%) hosted on GitHub. The selected projects fulfilled specific criteria: they are actively maintained, have at least a decade-long development history, and regularly integrate modern language features. However, this limited dataset may not comprehensively represent the broader Python community, restricting the generalizability of our results. Additionally, because our study focused exclusively on open-source repositories, our conclusions may not extend directly to industry or proprietary software development contexts.

For our qualitative investigation, we applied thematic coding to analyze pull-request comments relevant to Python source code rejuvenation. While our coding process followed established guidelines [49] and involved multiple authors to reduce subjectivity, the personal experience and perspectives of the researchers may still have influenced code interpretation and categorization. To mitigate this risk, we adopted a multi-cycle review process in which different authors independently coded the data, compared results, and resolved disagreements through discussion and consensus, with the participation of an additional author when needed. We also based our procedures on approaches validated in prior empirical studies on code review analysis [1, 48, 39]. Furthermore, we combined automated filtering—via a curated keyword search and large language model–based classification—with manual validation, allowing us to handle large volumes of comments while ensuring that the final dataset reflected to motivations and challenges for rejuvenation efforts. These measures aimed to enhance the reliability and validity of our findings, although we acknowledge that some bias and interpretive limitations may remain.

9. Conclusion

This study investigated how modern Python language features are adopted in open-source software projects. In particular, we examined when Python developers begin using new language features, how their adoption evolves over time, and which motivations and challenges influence source code rejuvenation in practice—especially in the period following the end-of-life of Python 2.7.

To this end, we employed a mixed-methods approach. First, we conducted a large-scale quantitative analysis of 424 open-source Python projects spanning 16 years of evolution (2008–2024), using an AST-based mining tool to track

feature usage at the file level over time and applying offline change-point detection to identify the onset of sustained adoption. Second, we complemented this analysis with a qualitative study of pull request discussions, systematically examining developers' comments to uncover explicit motivations and impediments associated with adopting modern Python features.

Our results show that adoption is a gradual and heterogeneous process. Quantitatively, developers often begin experimenting with new features shortly after their release, but widespread and sustained adoption typically takes several years to emerge. Features that offer lightweight, syntactic improvements tend to diffuse more rapidly, whereas constructs that introduce new paradigms or require broader structural changes exhibit longer maturation periods. Qualitatively, developers primarily associate code rejuvenation with improving code comprehension and consistency, while frequently citing compatibility constraints, maintenance costs, and learning effort as challenges that delay or limit adoption.

These findings contribute to Software Engineering research by offering a long-term empirical view of Python feature usage and by connecting longitudinal adoption patterns with themes observed in developers' pull request discussions. From a practical perspective, our results suggest considerations that may help tool builders and maintainers prioritize support for feature migration and modernization, for example by focusing on constructs with slower uptake and by accounting for commonly reported constraints such as version compatibility and perceived maintenance cost. For practitioners, this study provides empirical context that can support more informed planning of modernization efforts, including setting realistic expectations about adoption timelines and aligning feature uptake with project constraints and developer familiarity.

References

[1] Bacchelli, A., Bird, C., 2013. Expectations, outcomes, and challenges of modern code review, in: Notkin, D., Cheng, B.H.C., Pohl, K. (Eds.), 35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013, IEEE Computer Society. pp. 712–721. URL: https://doi.org/10.1109/ICSE.2013.6606617, doi:10.1109/ICSE.2013.6606617.

[2] Biswas, S., Islam, M.J., Huang, Y., Rajan, H., 2019. Boa meets python: a boa dataset of data science software in python language, in: Storey, M.D., Adams, B., Haiduc, S. (Eds.), Proceedings of the 16th International Conference on Mining Software Repositories, MSR 2019, 26-27 May 2019, Montreal, Canada, IEEE / ACM. pp. 577–581. URL: https://doi.org/10.1109/MSR.2019.00086, doi:10.1109/MSR.2019.00086.

[3] Bogachenkova, V., Nguyen, L., Ebert, F., Serebrenik, A., Castor, F., 2022. Evaluating atoms of confusion in the context of code reviews, in: IEEE International Conference on Software Maintenance and Evolution, ICSME 2022, Limassol, Cyprus, October 3-7, 2022, IEEE. pp. 404–408. URL: https://doi.org/10.1109/ICSME55016.2022.00048, doi:10.1109/ICSME55016.2022.00048.

[4] Callaú, O., Robbes, R., Tanter, É., Röthlisberger, D., 2011. How developers use the dynamic features of programming languages: the case of smalltalk, in: van Deursen, A., Xie, T., Zimmermann, T. (Eds.), Proceedings of the 8th International Working Conference on Mining Software Repositories, MSR 2011 (Co-located with ICSE), Waikiki, Honolulu, HI, USA, May 21-28, 2011, Proceedings, ACM. pp. 23–32. URL: https://doi.org/10.1145/1985441.1985448, doi:10.1145/1985441.1985448.

[5] Clarke, V., Braun, V., 2017. Thematic analysis. The journal of positive psychology 12, 297–298.

[6] Dabic, O., Aghajani, E., Bavota, G., 2021. Sampling projects in github for MSR studies, in: 18th IEEE/ACM International Conference on Mining Software Repositories, MSR 2021, IEEE. pp. 560–564.

[7] Dantas, R., Carvalho, A., Marcilio, D., Fantin, L., Silva, U., Lucas, W., Bonifácio, R., 2018. Reconciling the past and the present: An empirical study on the application of source code transformations to automatically rejuvenate java programs, in: Oliveto, R., Penta, M.D., Shepherd, D.C. (Eds.), 25th International Conference on Software Analysis, Evolution and Reengineering, SANER 2018, Campobasso, Italy, March 20-23, 2018, IEEE Computer Society. pp. 497–501. URL: https://doi.org/10.1109/SANER.2018.8330247, doi:10.1109/SANER.2018.8330247.

[8] Dietrich, J., Jezek, K., Brada, P., 2016. What java developers know about compatibility, and why this matters. Empir. Softw. Eng. 21, 1371–1396. URL: https://doi.org/10.1007/s10664-015-9389-1, doi:10.1007/S10664-015-9389-1.

[9] Foundation, P.S., . Python enhancement proposals (peps). https://peps.python.org/.

[10] Foundation, P.S., 2008. What's new in python 3.0. https://docs.python.org/3/whatsnew/3.0.html.

[11] Foundation, P.S., 2009. What's new in python 3.1. https://docs.python.org/3/whatsnew/3.1.html.

[12] Foundation, P.S., 2011. What's new in python 3.2. https://docs.python.org/3/whatsnew/3.2.html.

[13] Foundation, P.S., 2012. What's new in python 3.3. https://docs.python.org/3/whatsnew/3.3.html.

[14] Foundation, P.S., 2014. What's new in python 3.4. https://docs.python.org/3/whatsnew/3.4.html.

[15] Foundation, P.S., 2015. What's new in python 3.5. https://docs.python.org/3/whatsnew/3.5.html.

[16] Foundation, P.S., 2016. What's new in python 3.6. https://docs.python.org/3/whatsnew/3.6.html.

[17] Foundation, P.S., 2018. What's new in python 3.7. https://docs.python.org/3/whatsnew/3.7.html.

[18] Foundation, P.S., 2019. What's new in python 3.8. https://docs.python.org/3/whatsnew/3.8.html.

[19] Foundation, P.S., 2020. What's new in python 3.9. https://docs.python.org/3/whatsnew/3.9.html.

[20] Foundation, P.S., 2021. What's new in python 3.10. https://docs.python.org/3/whatsnew/3.10.html.

[21] Foundation, P.S., 2022. What's new in python 3.11. https://docs.python.org/3/whatsnew/3.11.html.

[22] Foundation, P.S., 2023. What's new in python 3.12. https://docs.python.org/3/whatsnew/3.12.html.
[23] Foundation, P.S., 2024a. History of python. https://docs.python.org/3/license.html#.
[24] Foundation, P.S., 2024b. Python (programming language). https://alalqab.com/en/Python_(programming_language).
[25] Grazia, L.D., Pradel, M., 2022. The evolution of type annotations in python: an empirical study, in: Roychoudhury, A., Cadar, C., Kim, M. (Eds.), Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022, ACM. pp. 209–220. URL: https://doi.org/10.1145/3540250.3549114, doi:10.1145/3540250.3549114.
[26] Hollander, M., Wolfe, D.A., Chicken, E., 2013. Nonparametric statistical methods. John Wiley & Sons.
[27] Hu, M., Zhang, Y., 2020. The python/c API: evolution, usage statistics, and bug patterns, in: Kontogiannis, K., Khomh, F., Chatzigeorgiou, A., Fokaefs, M., Zhou, M. (Eds.), 27th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2020, London, ON, Canada, February 18-21, 2020, IEEE. pp. 532–536. URL: https://doi.org/10.1109/SANER48275.2020.9054835, doi:10.1109/SANER48275.2020.9054835.
[28] Kapil, S., 2019. Clean Python: Elegant Coding in Python. First ed., Apress, New York, NY.
[29] Khatchadourian, R., Masuhara, H., 2018. Proactive empirical assessment of new language feature adoption via automated refactoring: The case of java 8 default methods. Art Sci. Eng. Program. 2, 6. URL: https://doi.org/10.22152/programming-journal.org/2018/2/6, doi:10.22152/PROGRAMMING-JOURNAL.ORG/2018/2/6.
[30] Kohn, T., van Rossum, G., II, G.B.B., Talin, Levkivskyi, I., 2020. Dynamic pattern matching with python, in: Flat, M. (Ed.), DLS 2020: Proceedings of the 16th ACM SIGPLAN International Symposium on Dynamic Languages, Virtual Event, USA, November 17, 2020, ACM. pp. 85–98. URL: https://doi.org/10.1145/3426422.3426983, doi:10.1145/3426422.3426983.
[31] Lin, H.Y., Thongtanunam, P., 2023. Towards automated code reviews: Does learning code structure help?, in: Zhang, T., Xia, X., Novielli, N. (Eds.), IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2023, Taipa, Macao, March 21-24, 2023, IEEE. pp. 703–707. URL: https://doi.org/10.1109/SANER56733.2023.00075, doi:10.1109/SANER56733.2023.00075.
[32] Lindstrom, G., 2005. Programming with python. IT Prof. 7, 10–16. URL: https://doi.org/10.1109/MITP.2005.120, doi:10.1109/MITP.2005.120.
[33] Lucas, W., Carvalho, F., Nunes, R.C., Bonifácio, R., Saraiva, J., Accioly, P.R.G., 2024. Embracing modern C++ features: An empirical assessment on the KDE community. J. Softw. Evol. Process. 36. URL: https://doi.org/10.1002/smr.2605, doi:10.1002/SMR.2605.
[34] Lucas, W., Nunes, R.C., Bonifácio, R., Carvalho, F., Lima, R., Silva, M., Torres, A., Accioly, P.R.G., Monteiro, E., Saraiva, J., 2025. Understanding the adoption of modern javascript features: An empirical study on open-source systems. Empir. Softw. Eng. 30, 107. URL: https://doi.org/10.1007/s10664-025-10663-9, doi:10.1007/S10664-025-10663-9.
[35] Luo, X., Xue, Y., Xing, Z., Sun, J., 2022. PRCBERT: prompt learning for requirement classification using bert-based pretrained language models, in: 37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022, Rochester, MI, USA, October 10-14, 2022, ACM. pp. 75:1–75:13. URL: https://doi.org/10.1145/3551349.3560417, doi:10.1145/3551349.3560417.
[36] Malloy, B.A., Power, J.F., 2017. Quantifying the transition from python 2 to 3: An empirical study of python applications, in: Bener, A., Turhan, B., Biffl, S. (Eds.), 2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM 2017, Toronto, ON, Canada, November 9-10, 2017, IEEE Computer Society. pp. 314–323. URL: https://doi.org/10.1109/ESEM.2017.45, doi:10.1109/ESEM.2017.45.
[37] Malloy, B.A., Power, J.F., 2019. An empirical analysis of the transition from python 2 to python 3. Empir. Softw. Eng. 24, 751–778. URL: https://doi.org/10.1007/s10664-018-9637-2, doi:10.1007/S10664-018-9637-2.
[38] Mostafa, S., Rodriguez, R., Wang, X., 2017. Experience paper: a study on behavioral backward incompatibilities of java software libraries, in: Bultan, T., Sen, K. (Eds.), Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, Santa Barbara, CA, USA, July 10 - 14, 2017, ACM. pp. 215–225. URL: https://doi.org/10.1145/3092703.3092721, doi:10.1145/3092703.3092721.
[39] Oliveira, D., 2021. Recommending code understandability improvements based on code reviews, in: 36th IEEE/ACM International Conference on Automated Software Engineering, ASE 2021 - Workshops, Melbourne, Australia, November 15-19, 2021, IEEE. pp. 131–132. URL: https://doi.org/10.1109/ASEW52652.2021.00035, doi:10.1109/ASEW52652.2021.00035.
[40] Parnin, C., Bird, C., Murphy-Hill, E.R., 2011. Java generics adoption: how new features are introduced, championed, or ignored, in: van Deursen, A., Xie, T., Zimmermann, T. (Eds.), Proceedings of the 8th International Working Conference on Mining Software Repositories, MSR 2011 (Co-located with ICSE), Waikiki, Honolulu, HI, USA, May 21-28, 2011, Proceedings, ACM. pp. 3–12. URL: https://doi.org/10.1145/1985441.1985446, doi:10.1145/1985441.1985446.
[41] Peng, Y., Zhang, Y., Hu, M., 2021. An empirical study for common language features used in python projects, in: 28th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2021, Honolulu, HI, USA, March 9-12, 2021, IEEE. pp. 24–35. URL: https://doi.org/10.1109/SANER50967.2021.00012, doi:10.1109/SANER50967.2021.00012.
[42] Pirkelbauer, P., Dechev, D., Stroustrup, B., 2010. Source code rejuvenation is not refactoring, in: van Leeuwen, J., Muscholl, A., Peleg, D., Pokorný, J., Rumpe, B. (Eds.), SOFSEM 2010: Theory and Practice of Computer Science, 36th Conference on Current Trends in Theory and Practice of Computer Science, Spindleruv Mlýn, Czech Republic, January 23-29, 2010. Proceedings, Springer. pp. 639–650. URL: https://doi.org/10.1007/978-3-642-11266-9_53, doi:10.1007/978-3-642-11266-9_53.
[43] Rak-amnouykit, I., McCrevan, D., Milanova, A.L., Hirzel, M., Dolby, J., 2020. Python 3 types in the wild: a tale of two type systems, in: Flat, M. (Ed.), DLS 2020: Proceedings of the 16th ACM SIGPLAN International Symposium on Dynamic Languages, Virtual Event, USA, November 17, 2020, ACM. pp. 57–70. URL: https://doi.org/10.1145/3426422.3426981, doi:10.1145/3426422.3426981.
[44] Review, D.b., 2026. "it makes the code clearer": Why developers adopt modern python features in open source projects. URL: https://doi.org/10.5281/zenodo.18462376, doi:10.5281/zenodo.18462376.
[45] Ronchetti, E.M., Huber, P.J., 2009. Robust statistics. John Wiley & Sons Hoboken, NJ, USA.

[46] Rosenkranz, S., Staegemann, D., Volk, M., Turowski, K., 2024. Explaining the business-technological age of legacy information systems. IEEE Access 12, 84579–84611. URL: https://doi.org/10.1109/ACCESS.2024.3414377, doi:10.1109/ACCESS.2024.3414377.

[47] Rózsa, B., Antal, G., Ferenc, R., 2022. Don't DIY: automatically transform legacy python code to support structural pattern matching, in: 22nd IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2021, Limassol, Cyprus, October 3, 2022, IEEE. pp. 164–169. URL: https://doi.org/10.1109/SCAM55253.2022.00024, doi:10.1109/SCAM55253.2022.00024.

[48] Sadowski, C., Söderberg, E., Church, L., Sipko, M., Bacchelli, A., 2018. Modern code review: a case study at google, in: Paulisch, F., Bosch, J. (Eds.), Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2018, Gothenburg, Sweden, May 27 - June 03, 2018, ACM. pp. 181–190. URL: https://doi.org/10.1145/3183519.3183525, doi:10.1145/3183519.3183525.

[49] Saldana, J., 2012. The Coding Manual for Qualitative Researchers. English short title catalogue Eighteenth Century collection, SAGE Publications. URL: https://books.google.com.br/books?id=kUms8QrE_SAC.

[50] Scarsbrook, J.D., Utting, M., Ko, R.K.L., 2023. Typescript's evolution: An analysis of feature adoption over time, in: 20th IEEE/ACM International Conference on Mining Software Repositories, MSR 2023, Melbourne, Australia, May 15-16, 2023, IEEE. pp. 109–114. URL: https://doi.org/10.1109/MSR59073.2023.00027, doi:10.1109/MSR59073.2023.00027.

[51] Stuart, M., 1984. Understanding robust and exploratory data analysis.

[52] Sun, X., Li, X., Li, J., Wu, F., Guo, S., Zhang, T., Wang, G., 2023. Text classification via large language models, in: Bouamor, H., Pino, J., Bali, K. (Eds.), Findings of the Association for Computational Linguistics: EMNLP 2023, Singapore, December 6-10, 2023, Association for Computational Linguistics. pp. 8990–9005. URL: https://doi.org/10.18653/v1/2023.findings-emnlp.603, doi:10.18653/V1/2023.FINDINGS-EMNLP.603.

[53] Truong, C., Oudre, L., Vayatis, N., 2020. Selective review of offline change point detection methods. Signal Processing 167, 107299. URL: https://www.sciencedirect.com/science/article/pii/S0165168419303494, doi:https://doi.org/10.1016/j.sigpro.2019.107299.

[54] Vándor, N., Antal, G., Hegedüs, P., Ferenc, R., 2024. On the usefulness of python structural pattern matching: An empirical study, in: IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2024, Rovaniemi, Finland, March 12-15, 2024, IEEE. pp. 501–511. URL: https://doi.org/10.1109/SANER60148.2024.00058, doi:10.1109/SANER60148.2024.00058.

[55] Zhang, Z., Xing, Z., Xia, X., Xu, X., Zhu, L., 2022. Making python code idiomatic by automatic refactoring non-idiomatic python code with pythonic idioms, in: Roychoudhury, A., Cadar, C., Kim, M. (Eds.), Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022, ACM. pp. 696–708. URL: https://doi.org/10.1145/3540250.3549143, doi:10.1145/3540250.3549143.

[56] Zhao, Y., Serebrenik, A., Zhou, Y., Filkov, V., Vasilescu, B., 2017. The impact of continuous integration on other software development practices: a large-scale empirical study, in: Rosu, G., Penta, M.D., Nguyen, T.N. (Eds.), Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017, IEEE Computer Society. pp. 60–71. URL: https://doi.org/10.1109/ASE.2017.8115619, doi:10.1109/ASE.2017.8115619.

Appendix D

Appendix D — Participants from Rounds 1 and 3

Table D.1: Overview of participant demographics, reporting years of professional experience, role, and organizational sector.

Participant	**Years of Experience**	**Professional Role**	**Organizational Sector**
P01	12 years	Senior Developer	Public Prosecutor's Office
P02	17 years	Software Architect	Consultancy
P03	18 years	Software Architect	Enterprise software
P04	6 years	Developer	Software tooling
P05	5 years	Developer	Enterprise software
P06	12 years	Senior Developer	Public Prosecutor's Office
P07	10 years	Tech lead	Enterprise software
P08	6 years	Developer	Military Institute of Engineering
P09	5 years	Developer Consultant	Software tooling
P10	6 years	Developer	Enterprise software
P11	17 years	Senior Developer	Enterprise software
P12	17 years	Senior Developer Researcher	Institute of Research Software tooling
P13	4 years	Developer	Military Institute of Engineering
P14	20 years	Senior Developer	Enterprise software
P15	11 years	Tech lead Co-Founder	Enterprise software
P16	30 years	Senior Developer Founder	Enterprise software
P17	9 years	Tech lead	Enterprise software
P18	5 years	Developer	Public Prosecutor's Office
P19	8 years	Senior Developer	Public Prosecutor's Office
P20	25 years	Senior Developer	Enterprise software
P21	15 years	Senior Developer	Banking Health
P22	30 years	Professor Senior Developer	Enterprise software Academic
P23	6 years	Researcher Developer	Enterprise software Academic
P24	14 years	Senior Developer	Enterprise software
P25	7 years	Senior Developer	Enterprise software
P26	6 years	Developer	Enterprise software
P27	8 years	Senior Developer	Enterprise software
P28	6 years	Developer	Enterprise software
P29	8 years	Senior Developer Tech lead	Enterprise software

Appendix E

Appendix E — Six C's Hierarchical Structure Summary

This appendix shows a hierarchical summary of the Six C's structure that was made from the study's qualitative and quantitative analysis. The table sorts the identified elements into the six C's categories: Causes, Conditions, Contingencies, and Consequences. It also shows the relationships between subcategories, concepts, and codes, as well as how often they occur.

The frequencies shown are the number of times each code appears in the dataset that was analyzed. These are then added together at the concept and subcategory levels. These counts show how important each part of the new theory is compared to the others, but they don't mean that the results can be generalized statistically.

It is essential to recognize that certain concepts manifest in various subcategories, indicating their complex function within the socio-technical dynamics of source code rejuvenation. So, the table should be seen as a structured way to show how codes, ideas, and groups are related in the grounded analysis, not as a strict hierarchy or a way to separate them.

Table E.1: Partial hierarchical distribution of codes across the Six C's structure in Source Code Rejuvenation (SCR); the complete version is available in [7].

Six C's	Subcategory	Subcat.(#)	Concept	Concept(#)	Code	Freq.
Causes	Drivers for SCR	39	System Obsolescence	14	Legacy system becoming obsolete	14
Causes	Drivers for SCR	39	Language Evolution Enforcement	10	Language version support discontinuity	10
Causes	Drivers for SCR	39	Dependency and Architectural Simplification	6	Removing unnecessary dependencies	6
Causes	Drivers for SCR	39	Dependency and Framework Pressure	5	Framework updates forcing changes	5
Causes	Drivers for SCR	39	Community-Driven Adoption	4	Influence from community practices	4
Conditions	Challenges & Impediments	157	Effort, Cost & Practical Constraints	38	High effort required for SCR	38
Conditions	Disablers	48	Organizational & Managerial Constraints	48	Management restrictions and priorities	48
Conditions	Challenges & Impediments	157	Compatibility & Legacy Constraints	35	Backward compatibility requirements	35
Conditions	Challenges & Impediments	157	External Dependencies & Environment Constraints	24	Dependency limitations	24
Conditions	Conditionally Factors	46	Availability of Automated and Regression Testing	23	Lack of testing infrastructure	23
Conditions	Challenges & Impediments	157	Unpredictability and Interaction Complexity in Language Evolution	20	Breaking and silent changes	20
Conditions	Challenges & Impediments	157	Risk, Uncertainty & Fear Factors	20	Fear of introducing bugs	20
Conditions	Tooling Support for SCR	45	Tooling Constraints and Control Mechanisms	20	Lack of trust in tools	20
Conditions	Conditionally Factors	46	Human & Knowledge Factors	18	Lack of expertise	18

Continued on next page

Six C's	Subcategory	Subcat.(#)	Concept	Concept(#)	Code	Freq.
Conditions	Challenges & Impediments	157	Technical Debt & System Quality Constraints	11	Accumulated technical debt	11
Conditions	Enablers	5	Language & Architectural Enablers	5	Modern architecture support	5
Conditions	Challenges & Impediments	157	Standards, Governance & Process Constraints	5	Organizational processes	5
Conditions	Disablers	48	Perceived Relevance & Motivation	3	Low perceived value of SCR	3
Conditions	Conditionally Factors	46	Contextual & Domain Factors	2	Domain-specific constraints	2
Consequences	Code Quality Improvements	205	Code Readability and Expressiveness	125	Improved readability	125
Consequences	Code Quality Improvements	205	Reliability and Correctness	33	Reduced bugs and errors	33
Consequences	Code Quality Improvements	205	Performance and Efficiency	25	Performance improvements	25
Consequences	Code Quality Improvements	205	Maintainability and Testability impacts	22	Easier maintenance	22
Consequences	Development Efficiency Gains	21	Development Productivity improvements	21	Increased productivity	21
Consequences	Challenges & Impediments	157	Degradation effects	4	Negative impact of no SCR	4
Contingencies	Execution and Operational Strategies	64	Execution Strategy and Effort Management	61	Incremental updates	61
Contingencies	Strategic Decision-Making	83	Decision-Making and Strategic Evaluation	52	Strategic evaluation of SCR	52
Contingencies	Strategic Decision-Making	83	Adoption Strategy and Timing	31	Timing decisions	31

Continued on next page

Six C's	Subcategory	Subcat.(#)	Concept	Concept(#)	Code	Freq.
Contingencies	Tooling Support for SCR	45	Selective and Human-Controlled Use of Tool Support	25	Manual validation of tools	25
Contingencies	Human and Social Strategies	18	Knowledge, Experience, and Team Dynamics	18	Team expertise influence	18
Contingencies	Execution and Operational Strategies	64	Continuous Technical Debt Management	3	Continuous refactoring	3
Contingencies	Handling Technical Constraints	3	Managing Technical Constraints and Compatibility	3	Managing constraints	3
Contingencies	CI/CD Strategies	2	CI/CD Infrastructure and Continuous Validation	2	Automated validation pipelines	2