



Universidade de Brasília

Instituto de Ciências Exatas
Departamento de Ciência da Computação

Estudo Comparativo de Brokers de Mensagens e Serializadores em Arquitetura do Open Finance

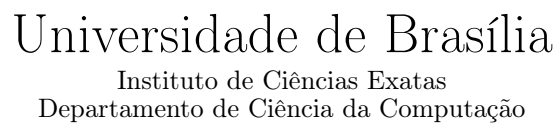
Fernando L. Araújo

Dissertação apresentada como requisito parcial para
conclusão do Mestrado em Informática

Orientador

Prof. Dr. Rodrigo Bonifacio de Almeida

Brasília
2026



Fernando L. Araújo

Prof. Dr. Rodrigo Bonifacio de Almeida (Orientador)
CIC/UnB

Prof. Dr. Cláudia Nalon
Coordenador do Programa de Pós-graduação em Informática

Brasília, 15 de Maio de 2026

Dedicatória

Dedico este trabalho, primeiramente, a Deus, cuja graça e direção iluminaram meu caminho em cada etapa desta jornada. Sua presença constante foi minha fortaleza nos momentos difíceis e minha inspiração nos momentos de conquista.

Aos meus pais, dedico esta realização como um agradecimento pelo amor incondicional, pelos valores que me transmitiram e por todo o apoio que me proporcionaram. Em especial, à minha mãe, que sempre sonhou em me ver alcançar este marco. Este trabalho é, de certa forma, a concretização de um desejo seu, e espero que ele simbolize o orgulho e a alegria que vocês sempre buscaram para mim.

Que esta conquista seja um reflexo da força, fé e dedicação que sempre recebi de Deus e da minha família, pilares fundamentais na minha vida.

Agradecimentos

Quero expressar minha mais profunda gratidão a todos que, de alguma forma, contribuíram para a realização deste trabalho de mestrado.

Em primeiro lugar, agradeço aos professores da Universidade de Brasília (UnB), cujas aulas, debates e orientações foram pilares fundamentais na minha formação acadêmica. A paixão de cada um de vocês pela educação e a dedicação em transmitir conhecimento deixaram marcas profundas no meu percurso como estudante e pesquisador. Cada discussão enriquecedora e cada desafio proposto ajudaram a moldar minha visão crítica e a expandir meu entendimento sobre o tema deste estudo.

Agradeço especialmente ao meu orientador, Rodrigo Bonifácio, por sua orientação ímpar e constante apoio ao longo desta jornada. Sua capacidade de enxergar além dos problemas e propor caminhos claros foi essencial para que este trabalho alcançasse seus objetivos. Sou imensamente grato por sua paciência, por estar sempre disponível para esclarecer dúvidas e por me desafiar a crescer academicamente. Sua mentoria não apenas direcionou esta pesquisa, mas também me inspirou a buscar excelência em cada etapa do processo.

Por fim, agradeço a todos aqueles que, direta ou indiretamente, contribuíram para a realização deste trabalho. Suas palavras de incentivo, apoio emocional e contribuições práticas foram indispensáveis para que eu pudesse superar os desafios e concluir esta etapa tão importante.

O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES), por meio do Acesso ao Portal de Periódicos.

Resumo

O interesse em otimizar microsserviços para o processamento de dados financeiros começou durante minha experiência com sistemas de alta demanda, onde atrasos e ineficiências frequentemente resultavam em gargalos operacionais. Esta dissertação se baseia nessa premissa, explorando estratégias arquiteturais para alcançar microsserviços escaláveis e resilientes no contexto do Open Finance. O estudo foca na otimização do desempenho de sistemas que lidam com grandes volumes de transações financeiras, com ênfase na redução do tempo de processamento.

Foi proposta uma arquitetura experimental integrando diferentes tecnologias de mensageria, representadas pelos brokers ActiveMQ e Apache Kafka, e diferentes mecanismos de serialização de dados, incluindo Json, Protobuf, Protostuff, Kryo e Msgpack. A arquitetura foi avaliada por meio de experimentos controlados considerando métricas de desempenho como tempo de resposta, transações por segundo (TPS), utilização de memória, consumo de CPU e consumo energético.

Os dados obtidos foram analisados por meio de técnicas estatísticas descritivas e inferenciais, permitindo validar as hipóteses formuladas sobre o impacto das diferentes combinações de brokers e serializadores no desempenho da arquitetura. Os resultados indicam que a escolha das tecnologias de mensageria e serialização exerce influência significativa sobre métricas relacionadas ao desempenho e à eficiência operacional, fornecendo evidências que podem apoiar decisões arquiteturais em sistemas distribuídos de alta demanda. Dessa forma, o estudo contribui para a compreensão dos trade-offs envolvidos na seleção dessas tecnologias e para o desenvolvimento de arquiteturas de microsserviços mais eficientes, escaláveis e sustentáveis.

Palavras-chave: Microsserviços, Open Finance, Processamento em Lote, Fila de Mensagens, Serialização, Otimização de Desempenho, Observabilidade

Abstract

The interest in optimizing microservices for financial data processing began during my experience with high-demand systems, where delays and inefficiencies frequently resulted in operational bottlenecks. This dissertation builds on that premise, exploring architectural strategies to achieve scalable and resilient microservices within the Open Finance context. The study focuses on optimizing the performance of systems handling large volumes of financial transactions, with an emphasis on reducing processing time.

An experimental architecture was proposed, integrating different messaging technologies, represented by the ActiveMQ and Apache Kafka brokers, and different data serialization mechanisms, including Json, Protobuf, Protostuff, Kryo, and Msgpack. The architecture was evaluated through controlled experiments considering performance metrics such as response time, transactions per second (TPS), memory utilization, CPU consumption, and energy consumption.

The collected data were analyzed using descriptive and inferential statistical techniques, enabling the validation of the hypotheses formulated regarding the impact of different broker and serializer combinations on the architecture's performance. The results indicate that the choice of messaging and serialization technologies significantly influences performance-related metrics and operational efficiency, providing evidence to support architectural decision-making in high-demand distributed systems. Therefore, this study contributes to a better understanding of the trade-offs involved in selecting these technologies and supports the development of more efficient, scalable, and sustainable microservices architectures.

Keywords: Microservices, Open Finance, Batch Processing, Message Queue, Serialization, Performance Optimization, Observability

Sumário

1	Introdução	1
1.1	Problema de Pesquisa	2
1.2	Objetivos	3
1.3	Metodologia	3
1.4	Organização do Documento	4
2	Fundamentação e Trabalhos Relacionados	5
2.1	Open Finance	5
2.2	Arquitetura de Microserviços	7
2.3	Representational State Transfer (REST)	8
2.4	Arquitetura Orientada a Eventos	9
2.5	Fila de Mensageria e Streaming	10
2.5.1	Green Software	10
2.6	Trabalhos Relacionados	11
2.6.1	Comparações de Arquiteturas	11
2.6.2	Comparações de Comunicação e Mensageria	12
2.6.3	Comparações de Serializadores	15
3	Metodologia Experimental	18
3.1	Metodologia Experimental	18
3.2	Definição do Problema e Justificativa	19
3.3	Planejamento Experimental	19
3.3.1	Formulação das Hipóteses	20
3.4	Arquitetura de Referência	21
3.4.1	Organização da arquitetura	22
3.4.2	Fluxo de processamento	22
3.4.3	Observabilidade	22
3.4.4	Tecnologias utilizadas	23
3.5	Variáveis do Experimento	24

3.5.1	Procedimento de Execução e Coleta de Dados	26
3.5.2	Variáveis Resposta	26
3.6	Recursos Computacionais	27
3.7	Análise dos Resultados	28
4	Resultados dos Experimentos	29
4.1	Métricas de Desempenho	29
4.1.1	Análise Estatística Descritiva	29
4.1.2	Testes Estatísticos	31
4.2	Métricas de Consumo de Energia	34
4.2.1	Análise Estatística Descritiva	34
4.2.2	Testes Estatísticos	36
4.3	Síntese dos resultados quantitativos	37
4.4	Avaliação Qualitativa por Grupo Focal	38
4.4.1	Metodologia do Grupo Focal	38
4.4.2	Resultados do Grupo Focal	39
4.4.3	Discussão dos Resultados	40
4.4.4	Limitações e Ameaças à Validade	41
5	Conclusão e Trabalhos Futuros	43
5.1	Conclusão	43
5.2	Limitações e Perspectivas Futuras	44
	Referências	46
	Apêndice	49
A	Apêndices	50
A.0.1	Apêndice A — Comparações Significativas (TukeyHSD)	50
A.0.2	Apêndice B — Diagnósticos dos Modelos (Rplots)	52
A.0.3	Apêndice C — Métricas Estatísticas (Conceitos)	57
A.0.4	Apêndice D — Reprodutibilidade e Pipeline da Análise	57
	Anexo	57
I	Anexos	58
I.0.1	Galeria Completa de Figuras	58

Lista de Figuras

3.1	Arquitetura Proposta.	21
3.2	Ranking 20 maiores receptores Setembro 2024.	25
4.1	Distribuição das métricas de desempenho por tecnologia de broker (<i>Kafka</i> vs <i>ActiveMQ</i>). A figura apresenta boxplots para uso de CPU, consumo de memória, tempo de resposta e transações por segundo, agrupados por broker.	30
4.2	Distribuição das métricas de desempenho por tecnologia de serialização. A figura apresenta boxplots para uso de CPU, consumo de memória, tempo de resposta e transações por segundo, agrupados por serializador.	31
4.3	Estatísticas descritivas do consumo de energia e emissões por <i>broker</i>	35
4.4	Estatísticas descritivas do consumo de energia e emissões por serializador.	36
A.1	Diagnóstico de modelo — saída Rplot001.. . . .	53
A.2	Diagnóstico de modelo — saída Rplot002.. . . .	53
A.3	Diagnóstico de modelo — saída Rplot003.. . . .	54
A.4	Diagnóstico de modelo — saída Rplot004.. . . .	54
A.5	Diagnóstico de modelo — saída Rplot005.. . . .	55
A.6	Diagnóstico de modelo — saída Rplot006.. . . .	55
A.7	Diagnóstico de modelo — saída Rplot007.. . . .	56
I.1	Gráfico de diagnóstico dos resíduos do modelo estatístico (verificação de normalidade e homocedasticidade).	58
I.2	Gráfico de diagnóstico dos resíduos do modelo estatístico (verificação de normalidade e homocedasticidade).	59
I.3	Gráfico de diagnóstico dos resíduos do modelo estatístico (verificação de normalidade e homocedasticidade).	59
I.4	Gráfico de diagnóstico dos resíduos do modelo estatístico (verificação de normalidade e homocedasticidade).	60
I.5	Gráfico de diagnóstico dos resíduos do modelo estatístico (verificação de normalidade e homocedasticidade).	60

I.6	Gráfico de diagnóstico dos resíduos do modelo estatístico (verificação de normalidade e homocedasticidade).	61
I.7	Gráfico de diagnóstico dos resíduos do modelo estatístico (verificação de normalidade e homocedasticidade).	61
I.8	Médias de uso de CPU por Broker e Serializador.	62
I.9	Médias de uso de memória por Broker e Serializador.	62
I.10	Médias de tempo de resposta por Broker e Serializador.	63
I.11	Médias de transações por segundo (TPS) por Broker e Serializador.	63
I.12	Boxplot da distribuição de uso de CPU por Broker e Serializador.	64
I.13	Boxplot da distribuição de uso de memória por Broker e Serializador.	64
I.14	Boxplot da distribuição do tempo de resposta por Broker e Serializador.	65
I.15	Boxplot da distribuição de transações por segundo (TPS) por Broker e Serializador.	65
I.16	Gráfico de coeficientes estimados do modelo com interação, destacando efeitos principais e termos de interação.	66
I.17	Gráfico auxiliar do pipeline de análise estatística, utilizado para inspeção intermediária dos resultados.	67
I.18	Gráfico auxiliar do pipeline de análise estatística, utilizado para inspeção intermediária dos resultados.	68
I.19	Gráfico auxiliar do pipeline de análise estatística, utilizado para inspeção intermediária dos resultados.	69
I.20	Gráfico auxiliar do pipeline de análise estatística, utilizado para inspeção intermediária dos resultados.	70
I.21	Gráfico auxiliar do pipeline de análise estatística, utilizado para inspeção intermediária dos resultados.	71
I.22	Gráfico auxiliar do pipeline de análise estatística, utilizado para inspeção intermediária dos resultados.	72
I.23	Gráfico auxiliar do pipeline de análise estatística, utilizado para inspeção intermediária dos resultados.	73
I.24	Gráfico auxiliar do pipeline de análise estatística, utilizado para inspeção intermediária dos resultados.	74
I.25	Gráfico auxiliar do pipeline de análise estatística, utilizado para inspeção intermediária dos resultados.	75
I.26	Gráfico auxiliar do pipeline de análise estatística, utilizado para inspeção intermediária dos resultados.	76
I.27	Gráfico auxiliar do pipeline de análise estatística, utilizado para inspeção intermediária dos resultados.	77

I.28	Gráfico auxiliar do pipeline de análise estatística, utilizado para inspeção intermediária dos resultados.	78
I.29	Gráfico auxiliar do pipeline de análise estatística, utilizado para inspeção intermediária dos resultados.	79
I.30	Gráfico auxiliar do pipeline de análise estatística, utilizado para inspeção intermediária dos resultados.	80
I.31	Gráfico auxiliar do pipeline de análise estatística, utilizado para inspeção intermediária dos resultados.	81
I.32	Gráfico auxiliar do pipeline de análise estatística, utilizado para inspeção intermediária dos resultados.	82
I.33	Gráfico auxiliar do pipeline de análise estatística, utilizado para inspeção intermediária dos resultados.	83
I.34	Gráfico auxiliar do pipeline de análise estatística, utilizado para inspeção intermediária dos resultados.	84
I.35	Gráfico auxiliar do pipeline de análise estatística, utilizado para inspeção intermediária dos resultados.	85
I.36	Gráfico auxiliar do pipeline de análise estatística, utilizado para inspeção intermediária dos resultados.	86
I.37	Gráfico auxiliar do pipeline de análise estatística, utilizado para inspeção intermediária dos resultados.	87
I.38	Gráfico auxiliar do pipeline de análise estatística, utilizado para inspeção intermediária dos resultados.	88
I.39	Gráfico auxiliar do pipeline de análise estatística, utilizado para inspeção intermediária dos resultados.	89
I.40	Gráfico auxiliar do pipeline de análise estatística, utilizado para inspeção intermediária dos resultados.	90
I.41	Gráfico auxiliar do pipeline de análise estatística, utilizado para inspeção intermediária dos resultados.	91
I.42	Gráfico auxiliar do pipeline de análise estatística, utilizado para inspeção intermediária dos resultados.	92
I.43	Gráfico auxiliar do pipeline de análise estatística, utilizado para inspeção intermediária dos resultados.	93
I.44	Gráfico auxiliar do pipeline de análise estatística, utilizado para inspeção intermediária dos resultados.	94

Lista de Tabelas

3.1	Versões das tecnologias utilizadas nos experimentos.	24
3.2	Combinações de Experimentos com ActiveMQ e Kafka utilizando diferentes Serializadores	24
4.1	Resultados do ART para consumo de CPU e memória.	32
4.2	Resultados do ART para tempo de resposta.	32
4.3	Resultados do ART para transações por segundo.	33
4.4	Resultados do ART para consumo de energia por operação.	36
4.5	Resultados do ART para consumo de energia por segundo.	37
A.1	Comparação de broker para tempo de resposta (ANOVA/Tukey).	50
A.2	Comparações pareadas por <i>serializer</i> para Tempo de Resposta (TukeyHSD).	50
A.3	Comparação do fator <i>Broker</i> para TPS (TukeyHSD).	51
A.4	Comparações entre serializadores para TPS (TukeyHSD).	51

Capítulo 1

Introdução

Otimizar o desempenho de arquiteturas de processamento de requisições é uma preocupação essencial em diversos domínios (como o financeiro e o de telecomunicações). Com o aumento do volume de dados e o crescimento das demandas de processamento, é fundamental buscar soluções que reduzam o tempo necessário para processar grandes volumes de requisições e informações.

Tal demanda por alta capacidade de processamento ocorre no *Open Finance*. O Open Finance é um conceito que se refere à abertura de sistemas financeiros e ao compartilhamento de dados e serviços por meio de APIs. Ele se baseia no princípio de promover a interoperabilidade e a colaboração entre diferentes instituições financeiras e provedores de serviços financeiros. O compartilhamento de dados entre instituições financeiras e provedores de serviços pode impulsionar a inovação, promover a concorrência e proporcionar uma melhor experiência ao cliente.

De acordo com Open Finance Brasil [1], desde o seu lançamento, este serviço tem se destacado como um dos maiores e mais dinâmicos ecossistemas de compartilhamento de dados financeiros no mundo. Com o suporte de mais de 800 instituições participantes, o sistema registrou mais de 2,7 bilhões de chamadas de APIs e 6,7 milhões de consentimentos ativos até o primeiro semestre de 2022. Esses números demonstram a rápida adoção e a confiança das instituições e dos consumidores no novo modelo.

O desenvolvimento do Open Finance Brasil foi realizado em fases, cada uma ampliando as funcionalidades e o escopo de compartilhamento de dados. A Fase 1 introduziu dados abertos das instituições participantes, enquanto a Fase 2 permitiu o compartilhamento de dados dos clientes mediante consentimento. A Fase 3, implementada a partir de outubro de 2021, introduziu a iniciação de transações de pagamento via Pix, destacando-se pela inovação e praticidade que trouxe aos usuários.

O relatório semestral do Open Finance Brasil [1] reflete os avanços e os desafios enfrentados, destacando a importância de uma regulamentação eficaz e de uma colaboração

continua entre todos os stakeholders. A evolução contínua do sistema, com a introdução de novas fases e funcionalidades, promete ampliar ainda mais os benefícios para consumidores e empresas, promovendo um ambiente financeiro mais inclusivo, competitivo e inovador.

Esse crescimento também aumenta o volume de requisições entre as instituições participantes, exigindo arquiteturas capazes de processar grandes quantidades de chamadas aos serviços do Open Finance.

Embora a comunicação entre as instituições ocorra por meio de APIs HTTP, o processamento interno dessas requisições pode utilizar diferentes estratégias arquiteturais para aumentar a capacidade de processamento. Entre essas estratégias estão o uso de brokers de mensageria para desacoplamento entre componentes e diferentes tecnologias de serialização para a troca de dados entre os microsserviços da aplicação.

Existem estudos que comparam brokers de mensageria ou formatos de serialização de forma isolada. No entanto, poucos trabalhos analisam essas tecnologias de forma conjunta no contexto do Open Finance, considerando cargas de trabalho baseadas nas mensagens previstas por suas especificações.

Neste trabalho, brokers e serializadores são avaliados em conjunto porque ambos fazem parte da arquitetura interna proposta para o processamento das requisições. Enquanto o broker coordena a troca de mensagens entre os componentes da arquitetura, o serializador define como essas mensagens são representadas durante esse processamento. Dessa forma, o estudo permite avaliar o comportamento das diferentes combinações dessas tecnologias em uma arquitetura desacoplada para processamento de requisições do Open Finance.

1.1 Problema de Pesquisa

Atualmente, a arquitetura utilizada pela instituição financeira MYBANK para processar requisições do Open Finance enfrenta desafios relacionados ao volume de chamadas realizadas às instituições participantes. O processamento dessas requisições pode demandar um tempo elevado, principalmente em cenários com grande quantidade de operações simultâneas. Como consequência, aumenta o tempo necessário para concluir o processamento interno dessas requisições. Esse cenário compromete tanto a experiência dos usuários quanto o desempenho geral da instituição, que lida diariamente com um grande volume de transações financeiras.

Uma alternativa para reduzir esse tempo consiste em utilizar uma arquitetura desacoplada baseada em brokers de mensageria e diferentes tecnologias de serialização para o processamento interno das requisições. Dessa forma, esta pesquisa busca avaliar essas combinações e identificar aquelas que apresentam melhor desempenho para esse cenário.

1.2 Objetivos

O objetivo principal desta pesquisa é comparar diferentes combinações de brokers de mensageria e tecnologias de serialização por meio de uma arquitetura experimental proposta neste trabalho, buscando identificar aquelas que apresentam melhor desempenho na redução do tempo de processamento das requisições no sistema Open Finance da instituição financeira MYBANK. A meta é alcançar maior eficiência no processamento de grandes volumes de dados, garantindo um desempenho otimizado da arquitetura em um ambiente de alta demanda. Para atingir o objetivo principal, os seguintes objetivos específicos foram definidos.

1. Propor e implementar uma arquitetura experimental reproduzível que permita avaliar diferentes combinações de tecnologias de serialização e implementações de filas de mensagens para o processamento das requisições.
2. Projetar um experimento computacional que permita avaliar, de forma sistemática, diferentes combinações de tecnologias de serialização e implementações de filas de mensagens.
3. Executar experimentos computacionais e avaliar quais combinações de serializadores e filas de mensagens resultam em maior capacidade de processamento no contexto do Open Finance.

1.3 Metodologia

Esta pesquisa adota uma abordagem aplicada, baseada em experimentos computacionais. A escolha dessa metodologia visa investigar empiricamente como diferentes configurações arquiteturais influenciam o desempenho do sistema Open Finance. O foco central está em analisar o impacto de escolhas tecnológicas específicas, como o uso de diferentes filas de mensagens e de serializadores, no tempo de processamento das requisições em um ambiente de alta demanda.

A metodologia utilizada será metodologia experimental, tendo os seguintes passos:

- Revisão da Literatura: Investigação teórica sobre a otimização do desempenho em arquiteturas de microserviços e de Open Finance.
- Planejamento Experimental: Definição dos parâmetros experimentais, como as filas de mensagens e as tecnologias de serialização.
- Execução dos Experimentos: Condução de experimentos que simulam diferentes cenários arquiteturais.

- Análise dos Resultados: Avaliação dos dados coletados utilizando técnicas estatísticas adequadas, incluindo análise exploratória de dados e testes de hipótese.

Mais detalhes sobre a metodologia são apresentados no Capítulo 3.

1.4 Organização do Documento

A dissertação está organizada em cinco capítulos, além de apêndices e anexos. O Capítulo 1 apresenta a introdução, contextualizando o problema de pesquisa, seus objetivos e a metodologia adotada. O Capítulo 2 reúne a fundamentação teórica e os trabalhos relacionados, abrangendo temas como Open Finance, arquitetura de microsserviços, REST, arquitetura orientada a eventos, serviços de mensageria e as principais comparações da literatura sobre brokers e serializadores. O Capítulo 3 descreve a metodologia experimental, incluindo a definição das hipóteses, o planejamento do experimento, a arquitetura de referência e os procedimentos de execução e de coleta de dados. O Capítulo 4 apresenta a avaliação quantitativa e qualitativa dos resultados, incluindo métricas de desempenho, métricas de consumo de energia, análises estatísticas, discussão das evidências e síntese interpretativa. Por fim, o Capítulo 5 reúne as conclusões gerais, as limitações e as sugestões para trabalhos futuros. Os apêndices e anexos complementam o texto com gráficos, diagnósticos estatísticos, tabelas e demais materiais de apoio utilizados na análise.

Capítulo 2

Fundamentação e Trabalhos Relacionados

Este capítulo apresenta a fundamentação teórica que serve de base para esta pesquisa. Inicialmente, são discutidos os principais conceitos relacionados ao Open Finance e às arquiteturas de microsserviços. Em seguida, são apresentados os fundamentos sobre comunicação entre serviços, arquitetura orientada a eventos, brokers de mensageria, tecnologias de serialização e Green Software. Por fim, são discutidos os trabalhos relacionados, destacando as principais abordagens encontradas na literatura e as lacunas que motivam esta pesquisa.

2.1 Open Finance

O Open Finance Brasil [2], também conhecido como sistema financeiro aberto, permite que clientes compartilhem seus dados financeiros entre instituições autorizadas pelo Banco Central. Esse modelo amplia a interoperabilidade do sistema financeiro ao possibilitar o acesso padronizado a informações mediante autorização do usuário, fortalecendo a competição e estimulando a inovação no setor.

Iniciativas semelhantes têm sido adotadas em diversos países sob o termo Open Banking, permitindo que instituições menores acessem dados de clientes e promovam serviços personalizados. No Brasil, conforme apontado por T. Aguiar [3], o Open Finance busca superar a falta de visibilidade entre instituições, permitindo o acesso seguro e consentido a dados previamente isolados em sistemas proprietários.

Diferentemente do Open Banking, o Open Finance possui escopo ampliado, abrangendo informações de câmbio, investimentos, seguros e previdência, como destacado por T. Wijnen [4]. Essa expansão possibilita que clientes utilizem seu histórico financeiro

para obter melhores condições em produtos variados e ampliem sua mobilidade entre instituições.

Além disso, o ecossistema está em evolução contínua. O Banco Central e a Susep trabalham na integração entre Open Finance e Open Insurance, permitindo a interoperabilidade entre os setores bancário e de seguros [2]. Esse movimento visa consolidar uma infraestrutura mais integrada, com benefícios diretos para consumidores e provedores de serviços.

Entre as funcionalidades já disponíveis ou em desenvolvimento, destacam-se:

Acesso a dados abertos de produtos e serviços financeiros: As instituições participantes disponibilizam informações padronizadas sobre seus canais de atendimento e características de produtos e serviços financeiros. Isso inclui dados sobre câmbio, seguros, previdência, investimentos e credenciamento. Essas informações estão em formato aberto, permitindo que desenvolvedores criem soluções que comparem diferentes ofertas financeiras, como comparadores de tarifas bancárias, tipos de contas e cartões de crédito. Isso auxilia as pessoas na escolha da opção mais adequada às suas necessidades.

Compartilhamento de dados cadastrais e transacionais: Os clientes têm a opção de compartilhar seus dados cadastrais, informações de transações em contas, cartões de crédito e produtos de crédito entre as instituições participantes. Esse compartilhamento ocorre (a) mediante autorização e (b) para finalidades específicas—sendo válido por um período determinado. Os clientes têm o controle e podem cancelar a autorização a qualquer momento. Em breve, também será possível compartilhar dados relacionados a produtos de câmbio, investimentos, seguros e previdência.

Iniciação de transação de pagamento: O Open Finance permite a realização de pagamentos, sendo o Pix a primeira modalidade disponibilizada. Outras formas de pagamento, como débito em conta, serão adicionadas ao longo do tempo. Isso tornará os pagamentos mais rápidos e convenientes para os clientes, preservando a segurança do processo.

Interoperabilidade com Open Insurance: O Open Insurance é um projeto semelhante ao Open Finance, focado no setor de seguros e supervisionado pela Susep. A interoperabilidade entre o Open Finance e o Open Insurance permitirá que os dados e informações transitem entre os dois ecossistemas, trazendo mais comodidade e benefícios para os usuários. Essa etapa está sendo desenvolvida pelo Banco Central e Susep, em conjunto com as estruturas de governança de ambos os projetos.

Encaminhamento de proposta de operação de crédito: Uma das próximas evoluções do Open Finance envolve soluções para receber propostas de operações de crédito. Isso facilitará o acesso a serviços financeiros, permitindo uma comparação mais fácil entre diferentes ofertas e a escolha de opções mais vantajosas para quem busca crédito. É importante ressaltar que o compartilhamento de dados nesses casos ocorrerá apenas com a autorização prévia e específica do cliente.

Essas funcionalidades visam ampliar a conveniência para os clientes, facilitar o acesso a produtos financeiros e incentivar a inovação. A participação no Open Finance pode ser obrigatória ou voluntária, dependendo do porte e do tipo de instituição. O processo de compartilhamento sempre ocorre mediante consentimento, e a instituição receptora deve informar claramente sua finalidade.

2.2 Arquitetura de Microsserviços

O Open Finance define as APIs que as instituições financeiras devem expor e o formato dos dados trocados entre elas, mas não impõe uma arquitetura interna. A arquitetura adotada neste trabalho é baseada em microsserviços, o que torna relevante compreender os conceitos principais desse modelo para contextualizar o experimento proposto. Fowler [5] descreve as arquiteturas de microsserviços como um sistema construído como um conjunto de serviços independentes, cada um executando em seu próprio processo e se comunicando por meio de protocolos leves, como HTTP ou mensageria assíncrona, conforme também discutido por Newman [6].

Richardson [7] descreve uma arquitetura de microsserviços como composta por vários serviços que são implementados e implantados de forma independente, permitindo que equipes de desenvolvimento trabalhem em diferentes partes do sistema simultaneamente. Cada serviço é responsável por uma única funcionalidade específica, seguindo o princípio da responsabilidade única (Single Responsibility Principle, ou SRP) [5]. Lewis e Fowler [8] discutem como essa abordagem modular facilita a manutenção, o teste e o deployment contínuo, além de permitir que cada serviço seja dimensionado independentemente para lidar com diferentes cargas de trabalho.

Fowler [5] aponta que uma das vantagens das arquiteturas de microsserviços é a possibilidade de usar tecnologias diferentes para implementar cada serviço, escolhendo a linguagem, o banco de dados e as ferramentas mais adequadas para cada contexto. Richardson [7] complementa que isso permite que as equipes adotem tecnologias especializadas e atualizem os componentes do sistema de forma independente, sem afetar o funcionamento dos outros serviços.

Dragoni et al. [9] analisaram a transição das abordagens orientadas a objetos e serviços para os microsserviços e discutem como essa arquitetura facilita a manutenção de sistemas complexos ao decompor grandes sistemas em serviços independentes que se comunicam por meio de mecanismos leves.

Em termos de desempenho, Dragoni et al. [9] mostram que a comunicação entre microsserviços via rede pode introduzir latências em comparação com chamadas em memória em arquiteturas monolíticas. Os autores também discutem a necessidade de ferramentas de suporte e padrões de design específicos para desenvolver, testar e manter arquiteturas de microsserviços.

O trabalho foi publicado em 2017, quando ainda havia incertezas sobre os benefícios de arquiteturas baseadas em microsserviços. Os autores apontam que a principal vantagem dos microsserviços é a flexibilidade e a capacidade de adaptação às mudanças, com independência operacional entre os serviços. A abordagem também apresenta desafios em termos de desempenho e segurança, decorrentes da natureza distribuída dos sistemas [9].

As arquiteturas de microsserviços também apresentam desafios, como o gerenciamento da complexidade da comunicação entre os serviços e a necessidade de implementar estratégias de descoberta de serviços e balanceamento de carga. Newman [6] aponta a importância de um bom design de contrato entre os serviços para manter uma comunicação consistente entre os componentes.

Giamattei et al. [10] propõem uma técnica chamada CAR-PT (*Causal-Reasoning-driven Performance Testing*), baseada em modelos causais para identificar e mitigar problemas de desempenho em arquiteturas de microsserviços. O CAR-PT utiliza raciocínio causal para gerar cargas de trabalho mais propensas a revelar problemas de desempenho, tratando as condições que causam desvios nas métricas de forma mais direta do que abordagens baseadas em geração de carga aleatória.

Os investigadores concluíram que o uso de modelos causais permitiu prever com maior precisão as condições que levariam a problemas de desempenho, como aumentos na latência e consumo de CPU. A técnica obteve bons resultados em testes preliminares, com potencial de aplicação em cenários reais para melhorar a confiabilidade de sistemas baseados em microsserviços.

2.3 Representational State Transfer (REST)

REST (Representational State Transfer), definido por R.T.Fielding [11], consolidou-se como um dos principais estilos arquiteturais para sistemas distribuídos. Seus princípios estabelecem diretrizes para construção de serviços web escaláveis, interoperáveis e de fácil manutenção. Nesta seção, são apresentados os fundamentos do REST conforme discutido

por R.T.Fielding [11], Booth et al.[12] e Pautasso et al. [13], contextualizando seu papel na arquitetura utilizada neste trabalho, onde atua como mecanismo de invocação das APIs responsáveis pelo acionamento do processamento e pela disponibilização de dados aos clientes.

De acordo com R. T.Fielding [11], o REST é centrado na noção de *recursos*, identificados por URIs (Uniform Resource Identifiers) e manipulados por meio de uma interface uniforme baseada em métodos HTTP, como GET, POST, PUT e DELETE. Esses métodos possibilitam a obtenção, criação, atualização e remoção de representações de recursos.

Uma característica estrutural do REST é o uso de comunicação sem estado (*stateless*). Segundo O. Farias [14], cada requisição deve conter todas as informações necessárias para ser processada, sem dependência de contexto mantido no servidor. Esse princípio reduz acoplamento, facilita a escalabilidade horizontal e simplifica a implementação dos componentes do servidor.

Um recurso pode ser disponibilizado em diferentes formatos, como Json ou XML, negociados por meio de cabeçalhos HTTP. Essa característica é diretamente relevante para este estudo, que avalia múltiplas técnicas de serialização (Json, Protobuf, Protostuff, Kryo e Msgpack) quanto ao seu impacto no desempenho.

2.4 Arquitetura Orientada a Eventos

O Open Finance define os contratos de API e os formatos de dados para a comunicação entre instituições financeiras, mas não impõe uma arquitetura interna para os sistemas que participam do ecossistema. A arquitetura proposta neste trabalho adota o modelo orientado a eventos para o processamento interno das mensagens, o que torna a escolha do broker de mensageria um fator direto no desempenho do sistema. É esse componente, o broker, que este estudo avalia.

Na arquitetura orientada a eventos, os componentes de um sistema se comunicam por meio de eventos que representam mudanças de estado. S. Ferg [15] define eventos como mudanças de estado em um sistema que desencadeiam respostas em outros componentes. Esse modelo permite que cada parte do sistema opere de forma independente, sem precisar conhecer os outros componentes com os quais se comunica.

M. Fowler [16] descreve que, nesse modelo, os componentes se comunicam por mensagens assíncronas, o que permite que cada parte do sistema seja alterada ou substituída sem impacto direto nas demais.

No experimento proposto, cada etapa do processamento é separada por uma fila de mensagens: o gerador de carga publica mensagens em uma fila, um consumidor as lê e as encaminha para processamento, e o resultado é registrado. Essa estrutura é o que torna

relevante comparar os brokers ActiveMQ e Kafka, pois são eles que gerenciam essas filas e, portanto, determinam o desempenho do fluxo como um todo.

2.5 Fila de Mensageria e Streaming

Mensageria é uma técnica usada em sistemas distribuídos para a comunicação assíncrona entre componentes ou serviços. Em vez de se comunicarem diretamente, os componentes trocam mensagens por meio de filas, sem precisar de uma conexão ativa entre emissor e receptor no momento do envio. Hohpe et al. [17] descrevem esse modelo como uma forma de desacoplar os componentes do sistema, o que permite que cada parte opere e escale de forma independente.

O streaming é uma abordagem diferente: em vez de filas discretas de mensagens, os dados fluem de forma contínua e são processados à medida que chegam, sem armazenamento intermediário. Kreps et al. [18] descrevem esse modelo como adequado para sistemas que exigem baixa latência no processamento de eventos, como monitoramento em tempo real e transações financeiras. O Kafka, objeto de estudo deste trabalho, foi projetado originalmente com base nesse modelo.

Na prática, os dois modelos não são excludentes. O ActiveMQ opera principalmente como um broker de filas de mensagens, enquanto o Kafka combina características de fila e streaming. Essa diferença de arquitetura interna é um dos fatores que influencia os resultados de desempenho comparados neste estudo.

2.5.1 Green Software

Nos últimos anos, o consumo de energia de aplicações de software passou a receber maior atenção na área de Engenharia de Software. O crescimento de aplicações distribuídas, microsserviços e infraestruturas em nuvem aumentou o interesse por soluções que reduzam não apenas o tempo de processamento, mas também o consumo energético associado à execução dos sistemas.

Segundo Lago et al. [?], a sustentabilidade pode ser tratada como um atributo de qualidade de software, devendo ser considerada durante o projeto e a avaliação de arquiteturas. Sob essa perspectiva, decisões arquiteturais, como a escolha de tecnologias e mecanismos de comunicação entre componentes, podem influenciar tanto o desempenho quanto o consumo de recursos computacionais e energéticos.

Uma revisão sistemática conduzida por Danushi et al. [19] mostra que a medição do consumo de energia tem recebido atenção crescente na literatura. Os autores identificam diferentes abordagens para medir e reduzir o consumo energético de software e apontam que decisões relacionadas à arquitetura, comunicação entre componentes e infraestrutura

podem influenciar diretamente esse consumo. A revisão também destaca a necessidade de mais estudos experimentais que avaliem essas características em cenários específicos de aplicação.

Nesta pesquisa, além das métricas tradicionais de desempenho, foram consideradas métricas de consumo de energia durante a execução dos experimentos. Essa avaliação permite verificar se combinações de brokers de mensageria e tecnologias de serialização que apresentam melhor desempenho também produzem diferenças no consumo energético da arquitetura proposta.

2.6 Trabalhos Relacionados

A presente seção tem como objetivo revisar estudos relevantes que sustentam o desenvolvimento desta dissertação, com ênfase em arquiteturas de microsserviços, padrões de design, comunicação e análise de desempenho. A seleção dos trabalhos foi guiada pelo potencial de contribuir para as discussões sobre desempenho, escalabilidade e utilização de recursos em sistemas baseados em microsserviços.

2.6.1 Comparações de Arquiteturas

Benavente et al. [20] apresentam um estudo comparativo entre arquiteturas de microsserviços e monolíticas. Os autores analisaram características essenciais, como tempo de resposta, capacidade de suporte a requisições e custos por milhão de requisições, utilizando ferramentas como JMeter, Amazon Web Services, Java, Python, Play, Jax-RS, Spring Boot e Angular. Os autores sugerem que a arquitetura de microsserviços é vantajosa para sistemas grandes e complexos, pois simplifica a base de código, oferece maior flexibilidade e estabilidade e facilita a manutenção e escalabilidade. Em termos de desempenho, os testes mostraram que microsserviços têm tempos de resposta mais baixos e suportam um número maior de requisições por minuto em comparação com a arquitetura monolítica. Além disso, a implementação baseada em microsserviços em ambientes de nuvem, como AWS Lambda, resulta em menores custos de infraestrutura por milhão de requisições. No entanto, a escolha da arquitetura depende dos requisitos específicos do sistema e do ambiente de desenvolvimento.

Em um estudo realizado por Singh et al.[21], os autores concluem que a arquitetura orientada a eventos oferece várias vantagens, como baixo acoplamento e capacidade de escalar com mais facilidade, o que é crucial para sistemas distribuídos. A latência em sistemas de mensagens orientados a eventos foi medida em diferentes casos de projetos de software, sugerindo que tais sistemas podem responder a eventos quase em tempo real. O trabalho também apresenta uma visão geral das arquiteturas orientadas a eventos versus

orientadas a mensagens, destacando que a primeira é mais eficiente para sistemas que exigem respostas em tempo real e alta escalabilidade.

Outro estudo conduzido por Rahmatulloh et al. [22] investiga a implementação da arquitetura orientada a eventos (EDA) em sistemas baseados em microsserviços com o objetivo de melhorar o desempenho e a escalabilidade. O estudo compara a EDA com a arquitetura baseada em API (API-Driven Architecture), utilizando contêineres orquestrados pelo Kubernetes e a comunicação intermediada por NATS Streaming como o barramento de eventos.

Os autores concluíram que a EDA proporciona um tempo de resposta 19,18% mais rápido e uma taxa de erro 34,40% menor em comparação com a arquitetura baseada em APIs. No entanto, a EDA requer maior uso de recursos de CPU, com um aumento de 8,52% no consumo em relação à arquitetura baseada em API. A arquitetura orientada a eventos mostrou-se mais eficaz em lidar com grandes volumes de requisições simultâneas, oferecendo maior escalabilidade e desempenho, especialmente em cenários em que a latência e a robustez são críticas.

Mais recentemente, Jackson et al. [23] realizaram um estudo empírico comparando diferentes tecnologias de streaming e protocolos de serialização. Os autores desenvolveram um framework extensível para benchmark e avaliaram mais de uma centena de combinações entre tecnologias de comunicação e formatos de serialização, considerando métricas como latência, throughput, tempo de serialização, tempo de desserialização e tamanho das mensagens. Os resultados evidenciam que o desempenho depende da combinação entre a tecnologia de comunicação e o protocolo de serialização adotados, destacando a existência de diferentes *trade-offs* entre latência, vazão e tamanho das mensagens. Embora o estudo realize uma avaliação abrangente dessas tecnologias, os experimentos foram conduzidos utilizando cargas de trabalho voltadas para aplicações científicas e de aprendizado de máquina, diferindo do contexto do Open Finance, no qual as mensagens seguem especificações padronizadas e requisitos próprios do domínio financeiro.

2.6.2 Comparações de Comunicação e Mensageria

Um estudo conduzido por Fu et al. [24] comparou o desempenho de diferentes sistemas de filas de mensagens, incluindo Kafka e MQ. Os resultados mostraram que o Kafka possui uma maior taxa de transferência. Ou seja, para usuários que precisam realizar coleta e análise de dados em larga escala, o Kafka se destaca como a escolha ideal. Devido à sua alta taxa de transferência, o Kafka é capaz de suportar a coleta contínua de dados em streaming, servindo como fonte de dados para diversas ferramentas de análise, como Flink, Spark e Storm. Kafka é especialmente adequado para o rastreamento de atividades em websites, coleta e monitoramento de dados em fluxo contínuo e fusão de logs. Para

otimizar o desempenho, de acordo com os autores, recomenda-se que o número de consumidores no grupo de consumidores seja igual ao número de partições, e que o número apropriado de partições seja configurado em cada Broker.

Em cenários onde a baixa latência é crucial para a qualidade do serviço, especialmente em transações online, Fu et al. [24] sugerem que o RocketMQ é a opção preferida. Sua alta confiabilidade e baixa latência fazem do RocketMQ uma solução adequada para aplicações como transações financeiras, recargas, envio de mensagens em tempo real e análises instantâneas. O RocketMQ é capaz de gerenciar inúmeros tópicos e acumulação de mensagens, tornando-o ideal para cenários de negócios complexos. O estudo de Fu et al. é o trabalho mais próximo desta dissertação em termos de comparação de brokers. A diferença principal é que os autores não avaliaram o impacto de diferentes formatos de serialização no desempenho, nem conduziram os experimentos no contexto do Open Finance, onde o tamanho e o volume das mensagens seguem padrões regulatórios específicos.

O artigo de Dobbelaere et al. [25] apresenta uma análise comparativa entre dois sistemas populares de publicação/assinatura (pub/sub), Apache Kafka e RabbitMQ, que são amplamente utilizados para a construção de sistemas distribuídos escaláveis e desacoplados. A comparação é baseada tanto em aspectos qualitativos quanto quantitativos, utilizando um framework comum para examinar as funcionalidades centrais desses sistemas. O estudo aborda aspectos como desempenho, escalabilidade, latência e throughput, além de destacar as características distintas de cada sistema. São enumerados casos de uso onde cada sistema se sobressai, e fornecido um guia para ajudar na escolha da melhor arquitetura de acordo com os requisitos específicos. Os autores chegaram as seguintes conclusões:

Latência Ambas as tecnologias são capazes de fornecer resultados com baixa latência (cerca de 10 ms de média/mediana). Para o RabbitMQ, a diferença entre os modos de entrega “at most once” e “at least once” é insignificante. Para o Kafka, a latência no modo “at least once” é aproximadamente duas vezes maior que no modo “at most once”, podendo aumentar significativamente se houver leitura de disco.

Throughput No setup mais básico, o throughput do RabbitMQ supera o do Kafka. No entanto, ao aumentar o número de partições no Kafka, seu desempenho melhora significativamente, demonstrando sua excelente escalabilidade. Aumentar o número de produtores/canais no RabbitMQ melhora o desempenho moderadamente.

Escalabilidade Ambos podem escalar distribuindo fluxos por múltiplos nós. No RabbitMQ, isso requer lógica adicional, enquanto no Kafka é uma funcionalidade nativa.

Impacto da Replicação A replicação reduz drasticamente o desempenho de ambos os sistemas, afetando o RabbitMQ em 50% e o Kafka em 75%.

Finalmente, de acordo com os autores, a escolha entre RabbitMQ e Kafka deve considerar múltiplas dimensões do problema, e uma combinação de ambos os sistemas pode ser a melhor solução para alguns cenários.

Um estudo comparativo semelhante ao de Fu et al. [24] foi conduzido por Chy et al. [26]. Nesse trabalho, os autores mostram que o Kafka, por exemplo, apresentou alto throughput e boa escalabilidade, embora com maior consumo de recursos. O Pulsar destacou-se pela eficiência no uso de CPU e memória, ainda que com latências mais elevadas em percentis superiores. O Artemis exibiu um desempenho equilibrado, especialmente para mensagens de menor tamanho, enquanto o RocketMQ apresentou limitações no processamento de mensagens maiores.

Os resultados do estudo indicam que a escolha da tecnologia de mensageria mais adequada depende fortemente dos requisitos específicos do sistema e do caso de uso. Fatores como tamanho das mensagens, volume de tráfego, requisitos de latência e disponibilidade de recursos computacionais devem ser considerados na tomada de decisão. Isso motiva a nossa pesquisa, uma vez que as características do Open Finance são diferentes das usadas nos estudos anteriores.

O estudo de A. Akbulut [27] comparou REST e RabbitMQ como mecanismos de comunicação em sistemas baseados em microsserviços. Utilizando a infraestrutura da AWS, os autores conduziram testes de carga para avaliar a performance em termos de throughput e tempo médio de processamento. Os resultados mostraram que RabbitMQ apresentou melhor desempenho em cenários de alta concorrência, mantendo consistência mesmo com o aumento da complexidade do sistema. Por outro lado, REST demonstrou limitações em cenários de alta latência, especialmente quando integrado a bancos de dados relacionais. Este trabalho é relevante para esta dissertação, pois fornece uma base para comparação entre comunicação síncrona e assíncrona. Contudo, ele não aborda a interação entre diferentes formatos de serialização e filas de mensagens, um aspecto que esta pesquisa busca explorar.

De acordo com Bux et al. [28], a comparação entre serviços web RESTful síncronos e a mensageria RabbitMQ assíncrona demonstra que, embora a comunicação RESTful seja amplamente utilizada devido à sua simplicidade, ela pode ser prejudicada pelo aumento da latência à medida que o número de serviços cresce. Esse estudo destaca que RabbitMQ, por ser um sistema de mensageria assíncrona, pode oferecer uma solução mais estável e eficiente em ambientes de alta carga de usuários, ao permitir que as requisições sejam processadas em paralelo e de forma desacoplada. No contexto do Open Finance, onde o volume de transações financeiras é considerável, a implementação de filas de mensagens como ActiveMQ ou Apache Kafka está diretamente relacionada a essa necessidade de reduzir o tempo de resposta e aumentar a capacidade de processamento em momentos de

pico de demanda.

Um outro estudo feito por Kumar et al. [29] trata da implementação e avaliação de ordenação global de mensagens utilizando clusters distribuídos de Kafka. A ordenação de mensagens é crucial em sistemas distribuídos para garantir que as mensagens sejam processadas na ordem correta, o que é especialmente importante em aplicações financeiras.

Em um estudo realizado por Guo et al. [30] utilizando o Spring Cloud, o sistema incorpora ferramentas para gestão de configuração, descoberta de serviços, balanceamento de carga e fila de mensagens RabbitMQ para controle de requisições. Testes de estresse realizados com Apache JMeter demonstraram que o sistema é altamente eficiente em lidar com grandes volumes de dados e acessos concorrentes, apresentando boas capacidades de escalabilidade e tolerância a falhas.

2.6.3 Comparações de Serializadores

J. C. Viotti [31] apresenta uma análise detalhada das especificações de serialização binária compatíveis com Json, incluindo Apache Avro, Microsoft Bond, Protocol Buffers e Apache Thrift. Os autores discutem as características, as técnicas de codificação e os casos de uso de cada uma dessas tecnologias. Protocol Buffers e Apache Avro são destacados por sua eficiência e robustez, especialmente em ambientes de *big data* e em sistemas distribuídos, onde a compactação eficiente e o suporte à evolução de esquemas são relevantes. Formatos como Json e BSON são preferidos pela simplicidade e ampla adoção, sendo mais indicados para aplicações que requerem flexibilidade e menor complexidade na integração. Os autores concluem que não existe uma solução universalmente ideal: a escolha depende dos requisitos específicos da aplicação, como desempenho, eficiência de espaço e suporte a evolução de esquemas.

Um estudo realizado por Álvaro et al. [32] compara o PSON com outros formatos populares de serialização, como Json, MessagePack, Protocol Buffers e XML. Os autores realizaram testes em duas plataformas de hardware distintas: Arduino UNO e ESP32-WROVER-B, que representam cenários de dispositivos de baixa e de alta capacidade de processamento, respectivamente. Os resultados demonstraram que o PSON apresenta um desempenho superior em termos de velocidade de serialização e desserialização, especialmente no ESP32. Comparado ao MessagePack, outro formato sem esquema similar ao PSON, o PSON foi 33% mais rápido na codificação e 55% mais rápido na decodificação em média.

Um artigo de K. Maeda [33] compara doze bibliotecas de serialização de objetos nos formatos XML, Json e binário. O estudo avalia o desempenho dessas bibliotecas medindo o tamanho dos arquivos serializados e o tempo de processamento durante a execução. O

objetivo é identificar as vantagens e desvantagens de cada formato e de cada biblioteca no contexto de aplicações distribuídas.

Os resultados mostram que não há uma solução única que se destaque como a melhor em todas as situações. A escolha da biblioteca de serialização depende do contexto específico de uso. As bibliotecas baseadas em binário, como Protocol Buffers e Avro, geralmente produzem arquivos menores e oferecem melhor desempenho no tempo de execução. No entanto, bibliotecas como Json e XML são mais fáceis de ler e entender, além de serem mais adequadas para ambientes heterogêneos com múltiplas linguagens de programação. A avaliação detalhada demonstrou que cada biblioteca possui pontos fortes e fracos distintos, dependendo dos requisitos do sistema e do ambiente de desenvolvimento.

Já o artigo A. Sumaray [34] compara quatro formatos de serialização de dados: XML, Json, Thrift e Protobuf. A pesquisa se concentra em eficiência de serialização em dispositivos móveis, levando em conta a velocidade de serialização, o tamanho dos dados e a usabilidade dos formatos. Testes foram realizados em dispositivos Android, utilizando objetos *text-heavy* e *number-heavy* para medir o desempenho das bibliotecas de serialização.

Os estudos apontaram que XML é o mais lento e menos eficiente em termos de tamanho dos dados, sendo o menos recomendado para ambientes móveis. Json é mais rápido que XML e tem um tamanho menor, tornando-se uma opção intermediária, especialmente útil devido à sua legibilidade e ampla adoção. Protobuf e Thrift são significativamente mais rápidos e geram arquivos menores do que XML e Json. Entre eles, Protobuf tem ligeira vantagem em termos de velocidade de deserialização, enquanto Thrift oferece maior suporte a múltiplas linguagens de programação.

Em outro estudo realizado por M. A. Pradana [35] sobre o uso Flatbuffers em comparação com outros formatos (Json, XML e CSV) para a comunicação no modelo Publish/Subscribe utilizando o protocolo MQTT. Os resultados indicaram que o Flatbuffers tem um valor de payload relativamente pequeno, sendo eficiente em termos de armazenamento, requerendo apenas 0.996108949 bytes por caractere. Em termos de latência, o Flatbuffers apresentou a melhor performance no processamento de dados, com um valor de 0.5002 ms, o que se deve ao fato de não realizar a desserialização quando os dados são recebidos pelo assinante. No entanto, a latência de entrega dos dados não foi tão competitiva, sendo maior que a do Json e XML.

Apesar do desempenho superior no processamento de dados, o throughput do Flatbuffers não superou o dos formatos Json e XML. Portanto, o estudo conclui que o formato Flatbuffers é muito adequado para armazenamento de dados, mas não é o ideal para o envio de dados. Sugere-se que o desempenho do Flatbuffers poderia ser melhor explorado ao utilizar protocolos de entrega distintos do MQTT.

Esta dissertação tem como objetivo comparar tecnologias de mensageria, mas amplia o escopo ao incluir não apenas ActiveMQ e Kafka, mas também uma análise de diferentes serializadores (Json, Protobuf, Kryo, etc.) no contexto do Open Finance. Enquanto o trabalho de Dobbelaere et al. [25] foca na comparação entre dois sistemas de filas, nossa abordagem avalia o impacto combinado de brokers e serializadores em métricas como uso de memória, uso de CPU e TPS.

Os estudos analisados fornecem uma base sólida para discussão, a presente pesquisa difere ao integrar padrões de microsserviços, comunicação e formatos de serialização. Além disso, ao validar as hipóteses em cenários de Open Finance, a dissertação busca contribuir para a literatura ao propor soluções otimizadas e aplicáveis a contextos reais e com volumes transacionais significativos.

Capítulo 3

Metodologia Experimental

Este capítulo descreve a metodologia adotada para condução dos experimentos desta pesquisa. Inicialmente, são apresentadas as hipóteses e o planejamento experimental. Em seguida, são descritos a arquitetura proposta, o ambiente utilizado para execução dos experimentos, as métricas coletadas e os procedimentos de coleta de dados. Por fim, são apresentados o planejamento da análise estatística e os procedimentos utilizados para garantir a reprodutibilidade dos experimentos.

3.1 Metodologia Experimental

De acordo com Jedlitschka et al. [36] e Juristo et al. [37], a Metodologia Experimental é um conjunto de procedimentos e técnicas utilizadas para planejar, implementar e analisar experimentos científicos. Essa metodologia tem como objetivo principal investigar relações de causa e efeito entre variáveis, permitindo testar hipóteses e obter conclusões confiáveis sobre fenômenos e processos.

Na Metodologia Experimental, os experimentos seguem uma série de etapas bem definidas. Inicialmente, as hipóteses de pesquisa são formuladas com base em teorias existentes, observações anteriores ou conhecimentos prévios do pesquisador. Em seguida, o experimento é planejado, onde são definidos os aspectos práticos, como as variáveis a serem manipuladas e medidas, a seleção dos participantes ou amostras, os grupos experimentais e de controle, os procedimentos experimentais e as métricas de avaliação do desempenho.

Com o planejamento estabelecido, o experimento é implementado conforme as diretrizes definidas. Durante essa fase, os dados são coletados por meio de observações, questionários, medições ou outros métodos apropriados. A coleta de dados é realizada de forma cuidadosa, garantindo a precisão e a consistência dos resultados.

Após a coleta de dados, a análise estatística é aplicada para identificar relações, tendências e diferenças significativas entre as variáveis. Métodos estatísticos adequados são utilizados para determinar a validade estatística dos resultados e para verificar se as hipóteses de pesquisa são suportadas pelos dados coletados.

Com base na análise dos dados, os resultados são interpretados e discutidos em relação ao contexto teórico e prático. As implicações dos resultados são identificadas, e conclusões são tiradas sobre as relações de causa e efeito entre as variáveis investigadas. As limitações do estudo também são discutidas, juntamente com possíveis direções futuras da pesquisa.

3.2 Definição do Problema e Justificativa

O problema central abordado nesta pesquisa é a necessidade de otimizar o desempenho da arquitetura Open Finance, que lida com o processamento de grandes volumes de requisições de dados financeiros. Atualmente, o elevado tempo de processamento dessas requisições compromete diretamente a eficiência do sistema e prejudica a experiência dos usuários.

A justificativa para esta pesquisa reside na importância de explorar alternativas de implementação que permitam a redução do tempo de resposta das requisições para outras instituições financeiras, melhorando, assim, a escalabilidade e a resiliência do sistema. Considerando o crescimento substancial de dados e a crescente demanda por serviços financeiros em tempo real, é essencial que as escolhas tecnológicas de implementação de serviços do Open Finance sejam capazes de lidar com a crescente necessidade de processamento de grandes volumes de requisições e de dados. O sucesso desse aprimoramento pode ser determinante para garantir que a instituição financeira MYBANK se mantenha competitiva, oferecendo serviços mais rápidos e confiáveis para seus clientes.

Diante desse cenário, a pesquisa explora alternativas arquiteturais com foco em mensageria e serialização para otimizar o processamento de requisições. A comparação entre diferentes abordagens visa identificar as soluções mais eficientes.

3.3 Planejamento Experimental

Durante o planejamento experimental são definidas as hipóteses e os aspectos práticos da condução do experimento. Isso inclui a escolha das variáveis a serem controladas e medidas, a definição dos grupos experimentais e de controle, a determinação dos procedimentos experimentais e a definição das métricas de avaliação do desempenho.

Para atingir os objetivos desta pesquisa, o estudo foi conduzido em duas etapas complementares. A primeira consiste em uma avaliação quantitativa baseada nos experimentos

computacionais descritos neste capítulo. Após a conclusão dessa etapa, foi realizada uma avaliação qualitativa por meio de um grupo focal com profissionais da área, com o objetivo de discutir a aplicabilidade dos resultados e complementar a interpretação das evidências obtidas nos experimentos.

3.3.1 Formulação das Hipóteses

O desempenho de arquiteturas baseadas em microsserviços pode ser impactado por diversos fatores. Para este experimento, foram escolhidas duas vertentes principais: os diferentes brokers de mensagens e as tecnologias de serialização. Dado que ActiveMQ e Kafka são soluções amplamente adotadas em sistemas distribuídos, esta pesquisa busca investigar como diferentes tecnologias de brokers e serialização (Json, Protobuf, Protostuff, Kryo e Msgpack) influenciam o desempenho desses sistemas.

A escolha dos brokers foi motivada pela intenção de explorar diferentes abordagens de mensageria amplamente reconhecidas no mercado. No entanto, o foco do experimento não está nas características exclusivas do Kafka ou ActiveMQ, podendo ter incluído outros brokers, como RabbitMQ ou qualquer outra solução de Message Queue (MQ). O objetivo principal é analisar como diferentes soluções de mensageria, em combinação com diversos serializadores, influenciam o desempenho da arquitetura.

Embora as chamadas externas ainda precisem ser realizadas utilizando Json, visto que o protocolo de comunicação com instituições externas não pode ser modificado, as tecnologias de serialização utilizadas internamente podem ser ajustadas para otimizar o desempenho.

No contexto desta pesquisa, foram formuladas as seguintes hipóteses:

- Hipótese 1: O desempenho da infraestrutura utilizada no experimento—medido em termos de tempo de resposta, transações por segundo (TPS), consumo de memória e uso de CPU—varia de acordo com cada combinação entre tecnologias de serialização (Json, Protobuf, Protostuff, Kryo, Msgpack) e a plataforma de mensageria ActiveMQ.
- Hipótese 2: O consumo de energia durante a execução dos experimentos—medido ao longo das diferentes configurações avaliadas—varia de acordo com cada combinação entre tecnologias de serialização (Json, Protobuf, Protostuff, Kryo, Msgpack) e a plataforma de mensageria ActiveMQ.

A comparação será conduzida em um ambiente de alta demanda, simulando cenário da busca de dados de contas do Open Finance. Embora o cenário real envolva volumes muito grandes de dados, as hipóteses serão testadas com um percentual representativo,

garantindo que os resultados permaneçam aplicáveis e reflitam condições reais de processamento.

Ao validar essas hipóteses, espera-se identificar não apenas o broker de mensagens mais eficiente, mas também como a escolha do formato de serialização pode otimizar o desempenho em termos de tempo de resposta, TPS, utilização de recursos (memória e CPU) e consumo de energia. Esta investigação fornecerá dados valiosos para a implementação de arquiteturas mais escaláveis e eficientes em sistemas de microsserviços.

3.4 Arquitetura de Referência

Para validar as hipóteses desta pesquisa, foi desenvolvida uma arquitetura de referência para execução dos experimentos. A arquitetura reúne os componentes responsáveis pela geração das requisições, processamento das mensagens, coleta de métricas e monitoramento da execução. A Figura 3.1 apresenta uma visão geral da arquitetura utilizada.

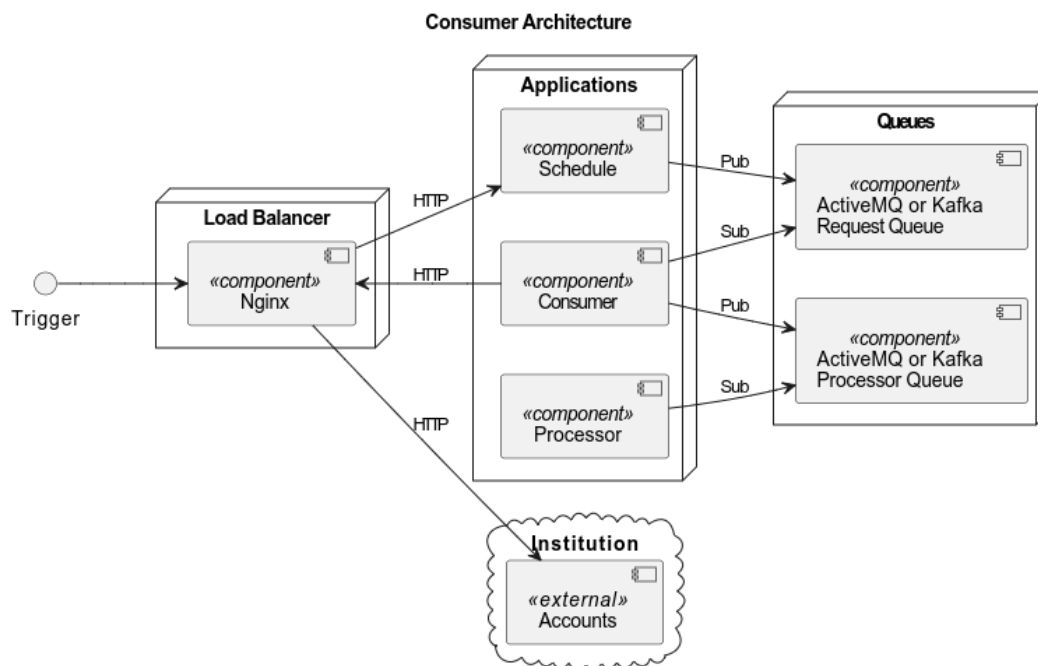


Figura 3.1: Arquitetura Proposta.

3.4.1 Organização da arquitetura

A arquitetura é composta pelos microserviços *Schedule*, *Consumer*, *Processor* e *Accounts*, além de um mecanismo de balanceamento de carga, um sistema de mensageria e componentes de observabilidade.

O balanceamento de carga foi implementado com o Nginx para distribuir as requisições entre múltiplas instâncias do serviço *Accounts*. Durante os testes preliminares, uma única instância desse serviço não foi suficiente para atender ao volume de requisições gerado pelo componente *Consumer*. Por esse motivo, o serviço *Accounts* foi executado com quatro instâncias em todos os experimentos. Os demais microserviços permaneceram com apenas uma instância, embora também estivessem acessíveis por meio do mesmo balanceador de carga.

Para avaliar o impacto do broker de mensageria, a arquitetura foi executada em duas configurações. Na primeira configuração, todas as filas utilizaram ActiveMQ. Na segunda, todas utilizaram Apache Kafka.

3.4.2 Fluxo de processamento

O componente *Schedule* inicia cada experimento. Inicialmente, ele recebe a quantidade total de requisições definida para a execução e gera lista de URLs a ser chamadas pelo *Consumer*. Essas URLs são organizadas em lotes de até 10 mil. Cada lote é publicado na fila de requisições **REQUEST-QUEUE**. Quando a quantidade total de URLs não é múltipla de 10 mil, o último lote contém apenas as URLs restantes.

O componente *Consumer* consome os lotes publicados na fila **REQUEST-QUEUE**. Após recuperar cada lote, utiliza o componente Flux do Spring Boot para executar entre 200 e 300 requisições HTTP simultâneas. Essas requisições são encaminhadas ao serviço *Accounts*, que simula o comportamento do endpoint de transações de conta definido pela especificação do Open Finance.

Após receber as respostas, o *Consumer* publica cada resultado individualmente na fila **PROCESSOR-QUEUE**. O componente *Processor* consome essas mensagens e realiza a etapa final do processamento. Embora esse componente permita operações como transformação, validação e agregação, durante os experimentos ele foi utilizado apenas para registrar as mensagens recebidas e concluir o fluxo de processamento.

3.4.3 Observabilidade

A arquitetura utiliza OpenTelemetry, Prometheus e Grafana para acompanhar a execução dos experimentos.

O OpenTelemetry foi utilizado para propagar o contexto de rastreamento entre os microsserviços. O script de benchmark envia um cabeçalho `traceparent` seguindo o padrão W3C Trace Context. O agente Java propaga automaticamente esse identificador pelas chamadas HTTP e pelas mensagens trocadas entre os componentes. Algumas métricas expostas pela aplicação utilizam esse identificador para associar cada execução ao respectivo *traceId*. Essas informações são consultadas diretamente no Prometheus para identificar quando uma execução experimental foi concluída.

O Prometheus realiza a coleta periódica das métricas expostas pelas aplicações por meio dos endpoints do Spring Boot Actuator. Entre as métricas coletadas estão tempo de resposta, utilização de CPU, consumo de memória da JVM e outras informações utilizadas durante os experimentos.

O Grafana foi utilizado apenas para visualizar essas métricas durante a execução dos experimentos. Sua utilização teve caráter de apoio, permitindo acompanhar os dados coletados pelo Prometheus de forma gráfica.

Para coletar as métricas de consumo de energia nos experimentos, foi utilizada a biblioteca CodeCarbon [38]. Trata-se de uma ferramenta de código aberto que estima e registra o consumo de energia e as emissões de dióxido de carbono da execução de software. A ferramenta converte o consumo elétrico em emissões equivalentes de CO₂ (CO₂eq), usando dados da composição energética regional para estimar o impacto ambiental. No Brasil, o coeficiente de intensidade de carbono adotado pelo CodeCarbon é de 98,348 gramas de CO₂ equivalente por quilowatt-hora, com base em dados de 2023 [39]. Internamente, o CodeCarbon utiliza o Intel RAPL (*Running Average Power Limit*) para medir o consumo dos processadores, o que significa que seus resultados se aplicam a ambientes com processadores Intel. No contexto desta pesquisa, o CodeCarbon foi utilizado para medir o consumo de energia durante a execução dos experimentos com os diferentes brokers e serializadores.

3.4.4 Tecnologias utilizadas

Os experimentos foram implementados em Java utilizando o framework Spring Boot. A escolha dessa plataforma permite integração direta com os componentes de mensageria utilizados no estudo, além de simplificar o desenvolvimento dos microsserviços.

Os serviços foram executados em contêineres Docker, permitindo reproduzir o mesmo ambiente em todas as execuções dos experimentos.

Os experimentos foram executados fevereiro de 2026 utilizando as versões apresentadas na Tabela 3.1.

Além de permitir a execução dos experimentos desta pesquisa, a arquitetura proposta, os scripts de automação, os procedimentos de coleta de métricas e os artefatos disponibili-

Tabela 3.1: Versões das tecnologias utilizadas nos experimentos.

Tecnologia	Versão
ActiveMQ Artemis	2.28.0
Apache Kafka (Confluent)	6.0.14
Apache ZooKeeper (Confluent)	6.0.14
Jackson (JSON)	2.15.0
Protocol Buffers	3.24.4
Protostuff	1.8.0
Kryo	5.6.0
MessagePack	0.9.8

zados no repositório formam um ambiente experimental reproduzível. Esse ambiente pode ser utilizado em estudos futuros para avaliar outras tecnologias de mensageria, serialização ou configurações arquiteturais no contexto do Open Finance.

3.5 Variáveis do Experimento

Neste estudo, investigamos duas variáveis de controle que podem afetar o desempenho das plataformas: o serviço de mensageria (com as opções ActiveMQ e Apache Kafka) e as tecnologias de serialização (incluindo Json, Protobuf, Protostuff, Kryo e Msgpack). O intuito é avaliar a eficácia de cada combinação das variáveis de controle em diversos cenários de carga. A tabela a seguir apresenta as combinações a serem testadas:

Plataforma de Mensageria	Serializador
ActiveMQ	Json
ActiveMQ	Protobuf
ActiveMQ	Protostuff
ActiveMQ	Kryo
ActiveMQ	Msgpack
Kafka	Json
Kafka	Protobuf
Kafka	Protostuff
Kafka	Kryo
Kafka	Msgpack

Tabela 3.2: Combinações de Experimentos com ActiveMQ e Kafka utilizando diferentes Serializadores

Para a variável de carga, iremos tomar como base o gráfico apresentado na Figura 3.2, que mostra o volume de requisições feitas por instituições atuando como receptoras de dados. Ou seja, analisando informações de clientes de outras instituições financeiras, é

possível observar os principais destaques em termos de volume de chamadas de APIs realizadas. Este gráfico apresenta dados referentes ao mês de setembro de 2024.

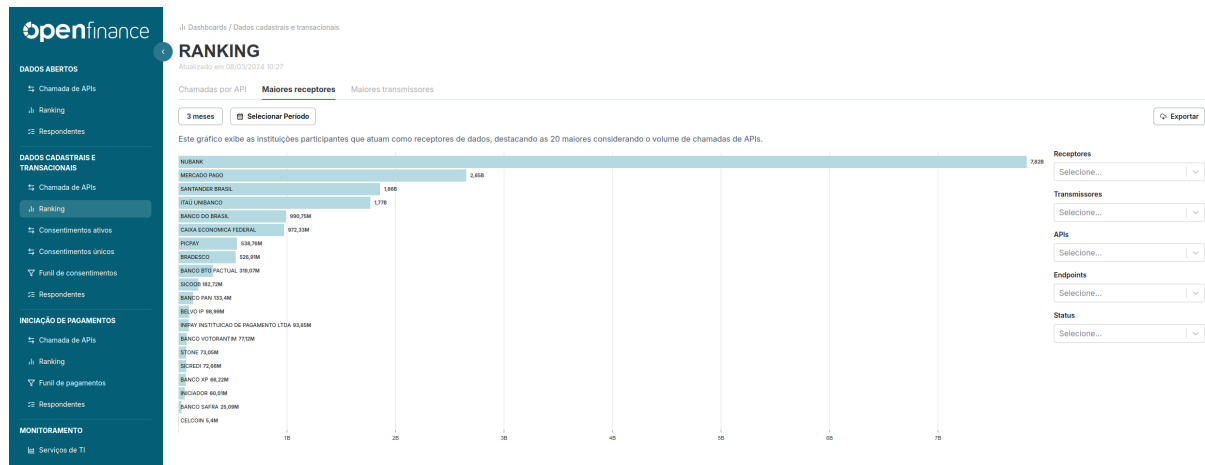


Figura 3.2: Ranking 20 maiores receptores Setembro 2024.

Entre as instituições destacadas, observa-se que o Nubank lidera o ranking com um expressivo volume de 7,82 bilhões de requisições, seguido por instituições como Mercado Pago, com 2,65 bilhões, e o Santander Brasil, com 1,86 bilhão. Também estão presentes nomes como BTG Pactual, que apresenta 319,07 milhões de chamadas no período, e outras instituições como Banco Pan e PicPay, com volumes variados.

Para exemplificar a análise, tomaremos o caso do BTG Pactual, que realizou aproximadamente 319,07 milhões de requisições ao longo do mês de setembro de 2024. Dividindo esse valor por 30 (número de dias do mês), chegamos à estimativa de que a instituição realizou, em média, cerca de 10,63 milhões de requisições por dia.

A partir desse volume médio diário, definimos o volume de execução para os testes realizados, utilizando percentuais desse valor. Como referências:

- 10% do volume diário: cerca de 1 milhão de requisições/dia;
- 50% do volume diário: aproximadamente 5 milhões de requisições/dia.
- 100% do volume diário: aproximadamente 10 milhões de requisições/dia.

Esses volumes serviram de base para os experimentos. Esse tipo de análise é crucial para compreender como o aumento ou a redução no volume de requisições pode impactar as métricas analisadas.

3.5.1 Procedimento de Execução e Coleta de Dados

Diante disso, para realizar os experimentos de comparação entre diferentes combinações de brokers de mensagens e serializadores, foi desenvolvido um script em Bash que automatiza todo o processo, desde a execução das requisições até a coleta das métricas.

O script realiza múltiplas iterações para cada broker (ActiveMQ e Kafka) e serializador (Json, Protobuf, Protostuff, Kryo e Msgpack), garantindo a reprodutibilidade dos experimentos. A cada execução, o script dispara requisições ao serviço de scheduler, que atua apenas como um disparador, enquanto o processamento ainda ocorre mesmo após a devolução da resposta. Para gerenciar essa dinâmica, o script verifica a quantidade de execuções solicitadas em relação às concluídas registradas no Prometheus.

Quando ambos os números coincidem, o script considera o processamento finalizado. Logo após a execução do processamento, são coletadas métricas, como uso de CPU, memória, tempo de resposta, transações por segundo (TPS) e consumo de energia; por fim, os dados são organizados em um arquivo CSV.

3.5.2 Variáveis Resposta

A cada ciclo de execução do experimento, as seguintes métricas são computadas:

- Tempo de Resposta (Response Time): Tempo de Resposta (Response Time): corresponde ao intervalo entre o envio de uma execução ao componente *Schedule* e a conclusão do processamento de todas as URLs pertencentes àquela execução. O início da medição ocorre no disparo da API do *Schedule*. O término é identificado quando a quantidade de URLs processadas é igual à quantidade de URLs geradas para a execução, conforme registrado no Prometheus. O valor obtido é convertido para segundos.
- Taxa de Transações por Segundo (TPS - Transactions Per Second): A métrica de TPS é calculada medindo a quantidade de transações concluídas em um segundo durante a execução de uma requisição. Para obter essa métrica, o script recupera o tempo de resposta conforme explicado acima; e, em seguida, divide o número total da volumetria solicitada. Esse cálculo gera a taxa média de transações por segundo, fornecendo uma visão da capacidade de processamento do sistema ao longo do experimento.
- Consumo de Memória RAM: A métrica de consumo de memória é coletada por meio de uma consulta ao Prometheus, que utiliza uma query para obter o consumo agregado de memória (em bytes) ao longo de um período específico. A consulta é realizada após a execução de cada requisição experimental, considerando os tempos

de início e término da operação. A métrica é, então, convertida para megabytes (MB), facilitando a interpretação e a análise do uso de memória. Esse processo é acionado pelo script, que coleta os dados do Prometheus e calcula o valor médio de memória consumida durante o experimento. Esse valor, quando convertido para MB, permite uma avaliação clara da demanda de memória para cada experimento realizado. O valor obtido representa o consumo agregado registrado para a execução e permite comparar o uso de memória entre os experimentos realizados.

- **Utilização de CPU:** A métrica de utilização de CPU também é coletada por meio de uma consulta ao Prometheus, que utiliza uma query para obter o consumo agregado de CPU ao longo de um período específico. Após a execução da requisição, o script consulta o Prometheus e retorna o valor correspondente à execução analisada. A métrica de CPU é convertida em milicores para facilitar a análise (onde 1000 milicores correspondem a um vCPU). Esse processo permite mensurar o consumo de processamento do sistema ao longo do experimento, possibilitando a avaliação da demanda de CPU em diferentes condições. Esse valor permite comparar a utilização de CPU entre os diferentes experimentos.
- **Consumo de Energia:** A métrica de consumo de energia é coletada por meio da biblioteca CodeCarbon [38], que estima o consumo elétrico e as emissões de CO₂ equivalente (CO₂eq) durante a execução do software. A ferramenta utiliza o Intel RAPL (*Running Average Power Limit*) para medir o consumo dos processadores e converte o consumo elétrico em emissões de CO₂eq com base na composição energética regional. A coleta é realizada pelo script de orquestração, que inicia o tracker em segundo plano antes de cada combinação de broker e serializador, mantém a medição ativa durante toda a execução do experimento e o encerra ao término, registrando os dados de consumo em quilowatt-hora (kWh) e as emissões correspondentes em gramas de CO₂eq em um arquivo.

3.6 Recursos Computacionais

Para a realização dos experimentos, foi selecionada uma infraestrutura em nuvem utilizando a Amazon Web Services (AWS), especificamente uma instância EC2 m5.2xlarge. Esta instância opera com o Amazon Linux 2, uma distribuição Linux mantida pela AWS, e utiliza o Docker Compose para a orquestração e inicialização das aplicações e dos componentes de infraestrutura necessários. A instância é equipada com 32GB de memória RAM, 8 vCPUs com frequência base de até 3.1 GHz, e armazenamento EBS (Elastic Block Store) com volumes GP3 configurados com 1TB, garantindo performance consistente de

I/O. Além disso, foi configurada uma partição de swap de 32GB para suportar cenários com alta demanda de memória, garantindo maior estabilidade durante a execução dos experimentos.

Quando a coleta de métricas de consumo de energia foi incorporada ao conjunto de variáveis resposta, todos os experimentos foram refeitos em uma instância `n2-highmem-8` do Google Cloud Platform (GCP), na região `us-central1` (Iowa, EUA). Essa instância tem 8 vCPUs Intel Xeon a 2,20GHz e 64 GB de memória RAM, executando Linux (kernel 6.1.0), com a mesma configuração de Docker Compose adotada na primeira rodada. Os resultados apresentados neste trabalho referem-se a essa segunda execução.

A utilização de infraestrutura em nuvem facilita a reprodutibilidade dos experimentos, pois outros pesquisadores podem recriar o mesmo ambiente computacional a partir das configurações descritas.

3.7 Análise dos Resultados

Os dados coletados nos experimentos foram analisados com técnicas estatísticas descritivas, incluindo média aritmética e regressão linear. A média é usada para capturar a tendência central dos dados e resumir os tempos de processamento de requisições para cada configuração avaliada. A regressão linear é usada para avaliar relações entre as variáveis analisadas, como o impacto de diferentes configurações no desempenho do sistema.

Os resultados são apresentados em tabelas e figuras para facilitar a interpretação, destacando as configurações que obtiveram melhor desempenho em termos de tempo de processamento.

Capítulo 4

Resultados dos Experimentos

Este capítulo apresenta os resultados dos experimentos conduzidos para avaliar o desempenho de diferentes serializadores Java, bem como as filas de mensagens **ActiveMQ** e **Kafka**. Os resultados deste experimento são importantes para compreender as nuances de desempenho entre diferentes tecnologias de serialização e sistemas de mensagens, fornecendo dados valiosos para os objetivos desta pesquisa. Inicialmente são apresentados os resultados quantitativos dos experimentos, incluindo análises descritivas e testes de hipótese para as métricas de desempenho e de consumo de energia. Em seguida, são apresentados os resultados da avaliação qualitativa realizada por meio de um grupo focal, conduzido com profissionais da área, com o objetivo de complementar a interpretação dos resultados experimentais.

4.1 Métricas de Desempenho

Essa seção apresenta os resultados dos experimentos para as métricas de desempenho. Iniciamos a seção com os resultados de uma análise descritiva e em seguida, reportamos os resultados de testes estatísticos.

4.1.1 Análise Estatística Descritiva

Conforme discutido no capítulo anterior, foram computadas métricas de desempenho—consumo de CPU, consumo de memória, tempo de resposta e transações por segundo—bem como métricas de consumo de energia. As Figuras 4.1 e 4.2 apresentam um resumo dos resultados das métricas de desempenho, considerando os serviços de mensageria (*brokers*) e as técnicas de serialização exploradas neste estudo. Para fins de reprodutibilidade, destaca-se que foram realizadas três execuções para cada combinação de *broker* e de técnica de serialização, considerando diferentes volumes de transações (i.e., 1M, 5M e 10M de

transações). Cada observação apresentada nos boxplots corresponde aos valores obtidos ao final de uma execução do experimento. No caso das métricas de CPU e memória, cada valor representa o consumo agregado registrado durante a respectiva execução. Por fim, uma medição foi removida do conjunto de dados por apresentar um tempo de resposta claramente caracterizado como *outlier*, possivelmente decorrente de uma instabilidade momentânea na infraestrutura de execução dos experimentos.

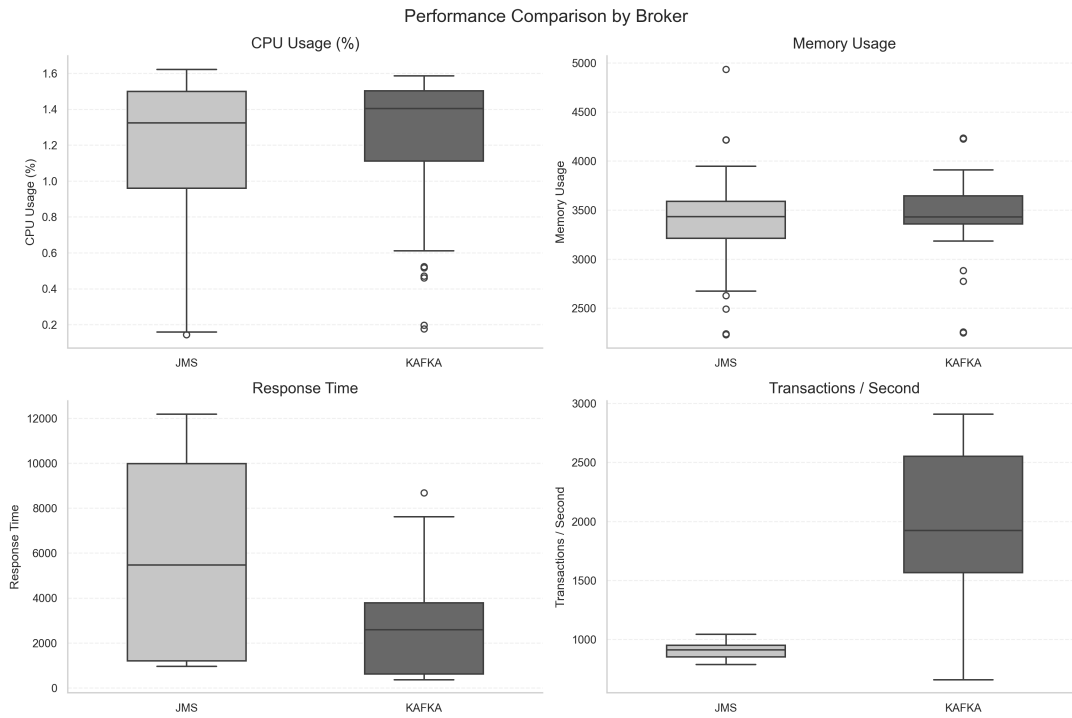


Figura 4.1: Distribuição das métricas de desempenho por tecnologia de broker (**Kafka** vs **ActiveMQ**). A figura apresenta boxplots para uso de CPU, consumo de memória, tempo de resposta e transações por segundo, agrupados por broker.

Considerando a Figura 4.1, observa-se que as distribuições das métricas de consumo de CPU e de memória são semelhantes entre os *brokers* avaliados. Por outro lado, o **Kafka** apresenta melhor desempenho que o **ActiveMQ** em termos de tempo de resposta e de volume de transações por segundo. A mediana do número de transações por segundo foi de 912,57 para o **ActiveMQ** e 1923,44 para o **Kafka**. Em relação ao tempo de resposta, a mediana foi de 5.479,00 para o **ActiveMQ** e 2.599,50 para o **Kafka**.

Considerando a Figura 4.2, observa-se que Kryo e MsgPack apresentam, visualmente, menores medianas para consumo de CPU e de memória. Protostuff e Protobuf apresentam menores tempos de resposta e maiores valores de transações por segundo.

A avaliação integrada dos brokers **Kafka** e **ActiveMQ**, em conjunto com os serializadores Protostuff, Protobuf, MsgPack, Kryo e Json, indica que o desempenho dos serializadores varia de forma consistente entre os dois ambientes, embora com magnitudes distintas. No

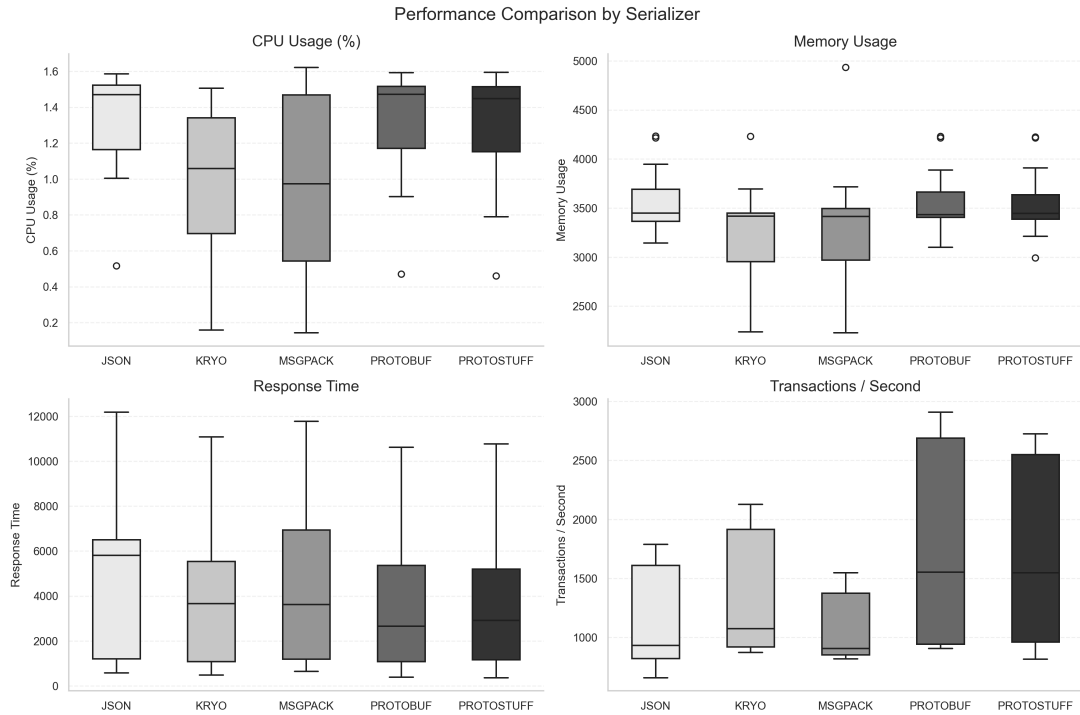


Figura 4.2: Distribuição das métricas de desempenho por tecnologia de serialização. A figura apresenta boxplots para uso de CPU, consumo de memória, tempo de resposta e transações por segundo, agrupados por serializador.

caso do **ActiveMQ**, as diferenças entre serializadores são relativamente modestas, com Protobuf e Protostuff apresentando as maiores medianas de transações por segundo (941,27 e 951,29, respectivamente), além dos menores tempos de resposta medianos (5402,0 e 5256,0). Por outro lado, JSON e MsgPack apresentam desempenho inferior nesse cenário, tanto em termos de vazão quanto de latência.

Nas configurações de execução com **Kafka**, as diferenças entre serializadores tornam-se mais evidentes. Protobuf e Protostuff destacam-se de forma significativa, alcançando medianas de transações por segundo substancialmente mais elevadas (2698,33 e 2558,85, respectivamente), ao mesmo tempo em que apresentam os menores tempos de resposta medianos (1852,0 e 1954,0). Em contraste, JSON e MsgPack exibem menor desempenho relativo, com maiores tempos de resposta e menor vazão. Além disso, observa-se um aumento na variabilidade das medições no KAFKA, especialmente para serializadores como JSON e KRYO, indicando maior sensibilidade às condições de execução.

4.1.2 Testes Estatísticos

Inicialmente, foram avaliados os pressupostos necessários para a aplicação de ANOVA, em particular a normalidade dos resíduos e a homogeneidade das variâncias. Os testes

de Shapiro–Wilk indicaram violação da hipótese de normalidade para todas as variáveis resposta, mesmo após a aplicação de transformações logarítmicas. Por outro lado, o teste de Levene confirmou a homogeneidade das variâncias entre os grupos.

Diante desse cenário, optou-se por complementar a análise com a técnica de *Aligned Rank Transform* (ART), que permite avaliar efeitos fatoriais e suas interações sem assumir normalidade dos dados. A adequação do procedimento foi verificada por meio dos diagnósticos do ART, que indicaram somas das respostas alinhadas próximas de zero e valores de F próximos de zero para efeitos não associados, conforme esperado.

Na análise pelo ART, foram considerados três fatores: o *Broker* (Kafka e ActiveMQ), o *Serializador* (Json, Kryo, MsgPack, Protobuf e Protostuff) e o *Volume de Execução* (1M, 5M e 10M de requisições), além das interações entre esses fatores.

A Tabela 4.1 apresenta os principais resultados da análise para consumo de CPU e memória. Observa-se que o consumo de CPU é significativamente influenciado pelo fator *Serializador* ($p < 0.01$), bem como pela interação entre *Serializador* e *Volume de Execução* (1M, 5M e 10M de requisições). Em contraste, o consumo de memória não apresentou diferenças estatisticamente significativas em relação a nenhum dos fatores ou interações considerados.

Tabela 4.1: Resultados do ART para consumo de CPU e memória.

Variável	Fator	F	p -valor	η_p^2
Consumo de CPU	Serializador	4.63	< 0.01	médio
Consumo de CPU	Serializador:Volume de Execução	3.32	< 0.01	médio
Consumo de Memória	–	–	> 0.25	desprezível

Em particular, a ausência de efeitos significativos para a memória sugere que ganhos de desempenho não estão associados a um aumento no consumo desse recurso.

A Tabela 4.2 apresenta os resultados da análise para o tempo de resposta. Observa-se que todos os fatores principais e as interações são estatisticamente significativos ($p < 0.001$), indicando uma forte dependência da latência em relação às variáveis analisadas.

Tabela 4.2: Resultados do ART para tempo de resposta.

Fator	F	p -valor	η_p^2
Broker	137.86	< 0.001	grande
Serializador	29.74	< 0.001	médio
Volume de Execução	238.70	< 0.001	muito grande
Broker:Serializador	9.71	< 0.001	médio
Serializador:Volume de Execução	10.41	< 0.001	médio
Interação tripla	5.22	< 0.001	pequeno/médio

Destaca-se o efeito dominante do volume de execução, o que indica que a latência é fortemente influenciada pela carga de trabalho. Além disso, a presença de interações significativas evidencia que o impacto dos serializadores depende tanto do *broker* quanto do volume de execução.

A Tabela 4.3 apresenta os resultados da métrica de vazão. Observa-se que o *Broker* e o *Serializador* apresentam efeitos altamente significativos ($p < 0.001$), bem como a interação entre esses fatores.

Tabela 4.3: Resultados do ART para transações por segundo.

Fator	F	p -valor	η_p^2
Broker	181.98	< 0.001	muito grande
Serializador	59.41	< 0.001	grande
Volume de Execução	3.70	< 0.05	pequeno
Broker:Serializador	48.03	< 0.001	grande
Serializador:Volume de Execução	2.39	< 0.05	pequeno
Interação tripla	2.11	< 0.05	pequeno

Os resultados indicam que o desempenho em termos de vazão é fortemente determinado pelo sistema de mensageria, sendo o efeito do *Broker* dominante. O fator *Serializador* também exerce influência significativa, especialmente em interação com o *Broker*, reforçando a importância de considerar esses fatores de forma conjunta.

Os testes revelaram diferenças significativas entre técnicas de serialização, particularmente no contexto do **Kafka**. Protobuf e Protostuff apresentaram desempenho superior em termos de vazão e tempo de resposta quando comparados a alternativas como JSON e MsgPack ($p < 0.001$). Em contraste, no ambiente ActiveMQ, as diferenças entre os serializadores são menos significativas. Esses resultados evidenciam a presença de efeitos de interação, indicando que a escolha da tecnologia de serialização deve ser considerada em conjunto com o sistema de mensageria e o volume de transações.

Além disso, os resultados mostram que métricas de desempenho, como tempo de resposta e vazão, são altamente sensíveis aos fatores analisados e às suas interações, enquanto métricas de consumo de recursos apresentam menor sensibilidade às configurações avaliadas. Em particular, o desempenho do sistema é determinado principalmente pela escolha do *broker* e pela carga de trabalho, enquanto que o impacto das técnicas de serialização, embora significativo, depende da configuração de execução.

4.2 Métricas de Consumo de Energia

O estudo sobre o consumo de energia e as emissões envolveu estatísticas descritivas e testes estatísticos, considerando tanto valores totais quanto métricas normalizadas. Em particular, analisamos o consumo total de energia, emissões de CO₂, tempo de execução, bem como métricas derivadas, como consumo de energia e emissões por unidade de tempo e por operação. Esta seção descreve tanto os resultados da análise descritiva quanto dos testes de hipótese.

4.2.1 Análise Estatística Descritiva

No nível de *broker*, as estatísticas descritivas indicam que execuções utilizando o **ActiveMQ** apresentam maior consumo total de energia e emissões quando comparadas ao **Kafka**. O consumo médio de energia no **ActiveMQ** (0.0248) é superior ao observado no **Kafka** (0.0161), sendo esse comportamento acompanhado por maiores emissões médias de CO₂ (0.0106 no **ActiveMQ** e 0.0069 no **Kafka**).

Ao considerar métricas normalizadas (ver Figura 4.3), como consumo de energia por segundo, observa-se que o **Kafka** apresenta valores mais elevados do que o **ActiveMQ**, indicando maior intensidade energética por unidade de tempo. Em contrapartida, o **ActiveMQ** apresenta menor consumo por segundo, sugerindo maior eficiência energética, ainda que com maior tempo de execução. Dessa forma, a aparente vantagem do **Kafka** em termos de consumo total de energia pode ser atribuída ao seu menor tempo de execução, e não a diferenças substanciais na eficiência energética por unidade de tempo.

Diferentemente da análise por *broker*, as diferenças entre serializadores permanecem evidentes mesmo após a normalização das métricas. De modo geral, os serializadores **Protobuf** e **Protostuff** tendem a apresentar valores médios de consumo de energia e emissões ligeiramente menores, tanto por unidade de tempo quanto por operação, indicando maior eficiência energética.

Observa-se, ainda, que o serializador **JSON** apresenta, de forma consistente, maiores valores de consumo de energia por operação, conforme evidenciado nos boxplots da Figura 4.4. Esse resultado sugere um maior custo energético por unidade de trabalho quando comparado a alternativas como **Protobuf** e **Protostuff**.

Por outro lado, ao considerar o consumo de energia por segundo, essa tendência não se mantém de forma consistente. Os boxplots indicam que o **JSON** não se destaca como o serializador mais intensivo energeticamente nessa métrica, apresentando valores comparáveis aos demais serializadores. Esse resultado sugere que o maior custo do **JSON** está mais associado à quantidade de trabalho realizado do que à intensidade energética ao longo do tempo.

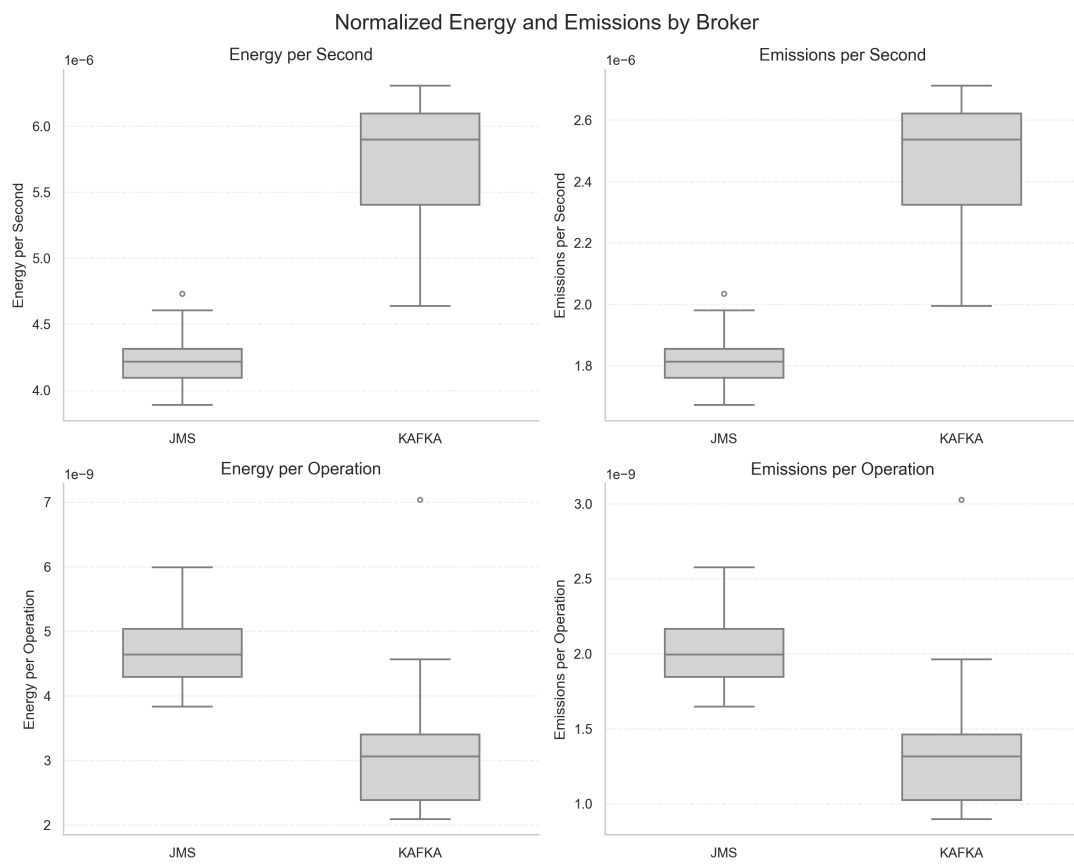


Figura 4.3: Estatísticas descritivas do consumo de energia e emissões por *broker*.

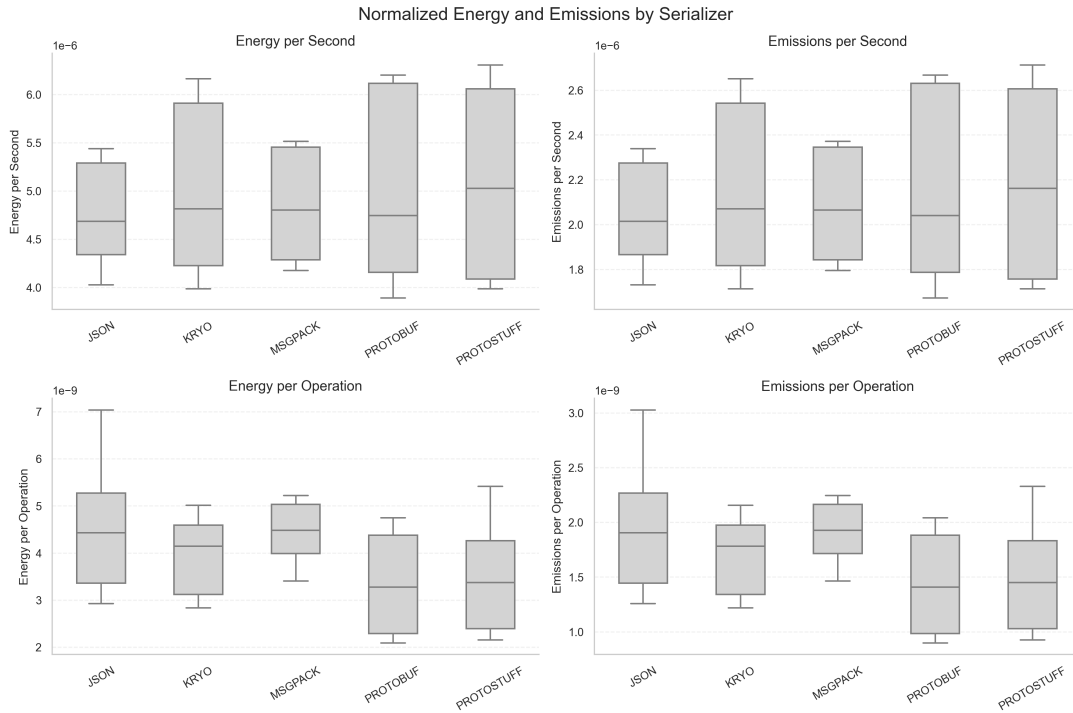


Figura 4.4: Estatísticas descritivas do consumo de energia e emissões por serializador.

De forma geral, os resultados indicam que o impacto do serializador na eficiência energética não é uniforme entre as métricas analisadas. Enquanto algumas abordagens apresentam maior custo energético por operação, as diferenças em termos de intensidade energética por unidade de tempo são menos evidentes.

4.2.2 Testes Estatísticos

De maneira análoga à análise de desempenho, a suposição de normalidade foi violada mesmo após transformação logarítmica. Assim, adotou-se a técnica de *Aligned Rank Transform* (ART) para os testes de hipótese. A adequação do procedimento foi confirmada pelas somas das colunas alinhadas, próximas de zero para ambas as métricas (ordem de grandeza 10^{-14}).

Tabela 4.4: Resultados do ART para consumo de energia por operação.

Fator	F	p -valor	η_p^2
Broker	158.17	< 0.001	muito grande
Serializer	42.91	< 0.001	muito grande
Broker:Serializer	7.79	< 0.001	médio

A Tabela 4.4 mostra que tanto o *broker* quanto a tecnologia de serialização influenciam significativamente o consumo de energia por operação. A interação entre os fatores

($F(4, 78) = 7.79$, $p < 0.001$) indica que a eficiência relativa dos serializadores depende do ambiente de execução.

As comparações pós-hoc (ajuste de Holm) revelam dois grupos estatisticamente indistinguíveis: **Protobuf** e **Protostuff** ($p = 0.551$), e **JSON** e **MsgPack** ($p = 0.114$). As demais diferenças são significativas ($p < 0.001$), com maior consumo para **JSON** e **MsgPack**, e menor para **Protobuf** e **Protostuff**.

Tabela 4.5: Resultados do ART para consumo de energia por segundo.

Fator	F	p -valor	η_p^2
Broker	239.37	< 0.001	muito grande
Serializer	13.71	< 0.001	grande
Broker:Serializer	29.06	< 0.001	grande

A Tabela 4.5 evidencia um efeito ainda mais pronunciado do *broker* sobre o consumo de potência ($\eta_p^2 = 0.754$), indicando que **Kafka** opera, em média, com maior potência que **ActiveMQ**. A interação significativa ($F(4, 78) = 29.06$, $p < 0.001$) reforça que o comportamento dos serializadores varia conforme o broker.

Para energia por segundo, observa-se um padrão distinto: **Kryo**, **Protobuf** e **Protostuff** formam um grupo sem diferenças significativas ($p = 1$), assim como **JSON** e **MsgPack** ($p = 1$). Entretanto, há separação clara entre os grupos: **JSON** e **MsgPack** apresentam menor consumo de potência que os demais ($p < 0.001$). Isso sugere que seu maior custo energético por operação decorre de maior tempo de execução, e não de maior potência instantânea.

As emissões de CO_2 , por serem reescalonamentos lineares das métricas de energia com base no coeficiente de intensidade de carbono do CodeCarbon, apresentam resultados equivalentes e, por isso, não são reportadas separadamente.

4.3 Síntese dos resultados quantitativos

Os experimentos mostram que a escolha do broker de mensageria tem impacto muito maior no desempenho do sistema do que a escolha do serializador. O **Kafka** apresentou desempenho superior ao **ActiveMQ** em todas as combinações testadas: a mediana de TPS foi de 1923,44 transações por segundo no **Kafka** contra 912,57 no **ActiveMQ**, e a mediana de tempo de resposta foi de 2599,50 ms no **Kafka** contra 5479,00 ms no **ActiveMQ**. Os testes estatísticos confirmaram essa diferença com $p < 0,001$ e tamanho de efeito grande para ambas as métricas. O fator broker explica aproximadamente 95% da variância no tempo de resposta e 81% da variância no TPS.

Entre os serializadores, Protobuf e Protostuff apresentaram os melhores resultados em tempo de resposta e TPS, especialmente quando combinados com o Kafka. O Json teve o pior desempenho em todas as combinações. O Kryo ficou em posição intermediária, mas com um ponto relevante: ao contrário do Protobuf, que exige a criação de arquivos de schema, o Kryo serializa objetos Java diretamente, o que reduz o esforço de implementação. CPU e memória não apresentaram diferenças estatisticamente significativas entre as combinações testadas, o que indica que esses recursos não são fatores decisivos na escolha entre as tecnologias avaliadas.

Do ponto de vista prático, a combinação Kafka com Protobuf ou Protostuff é a que oferece maior ganho de desempenho em latência e vazão. O MyBank é um exemplo de ambiente onde esses resultados seriam aplicáveis. Os processos internos do sistema, como filas de processamento de consentimentos, notificações de eventos e ingestão de dados de outras instituições, poderiam se beneficiar diretamente da adoção do Kafka como broker e de formatos binários para a comunicação interna. O Kryo seria uma alternativa razoável ao Protobuf quando o esforço de criar e manter arquivos de schema não se justifica pelo volume de transações ou pelo prazo do projeto. O Json continuaria obrigatório nas interfaces externas por exigência regulatória do Open Finance, mas poderia ser substituído por formatos binários em toda a camada interna, onde os ganhos medidos neste estudo se aplicariam diretamente.

4.4 Avaliação Qualitativa por Grupo Focal

Esta seção apresenta os resultados da avaliação qualitativa realizada por meio de um grupo focal. O objetivo foi complementar os resultados quantitativos com percepções de profissionais com experiência no domínio do Open Finance e em arquiteturas distribuídas.

4.4.1 Metodologia do Grupo Focal

O grupo focal foi realizado de forma síncrona, com duração aproximada de 60 minutos, e contou com quatro participantes, incluindo o pesquisador. Os participantes foram selecionados por conveniência, priorizando profissionais com vivência em sistemas financeiros de larga escala, experiência com mensageria e serialização de dados, e familiaridade com o Open Finance.

Os participantes possuíam experiência profissional em desenvolvimento de software e arquitetura de sistemas distribuídos. O grupo foi composto por um engenheiro de software com sete anos de experiência, uma coordenadora e um arquiteto de soluções, ambos com mais de dez anos de atuação em projetos de desenvolvimento de software. O pesquisador

também participou da sessão na função de moderador e de apresentador dos resultados experimentais.

A sessão foi gravada, transcrita e analisada por meio de análise temática [40]. O método consiste em identificar e organizar padrões recorrentes no discurso dos participantes. As etapas foram: leitura exploratória da transcrição, codificação das falas relevantes, agrupamento em categorias temáticas e identificação de percepções representativas.

O roteiro foi semiestruturado e incluiu a apresentação do *evidence briefing* com os resultados experimentais, discussão sobre clareza e compreensão dos dados, análise da relevância prática das tecnologias testadas, e identificação de barreiras e oportunidades para aplicação em instituições financeiras.

4.4.2 Resultados do Grupo Focal

A análise temática organizou as falas em três categorias, apresentadas a seguir.

Clareza dos resultados e interpretação das métricas

Os participantes consideraram os resultados claros e bem apresentados. A distinção entre as métricas com e sem diferença prática foi percebida com facilidade. Em particular, os participantes observaram que CPU e memória não exercem impacto relevante na escolha do broker ou do serializador, e que tempo de resposta e TPS são as métricas determinantes para essa decisão. Um participante apontou que não tinha expectativa prévia sobre o comportamento de CPU e memória, e que os experimentos confirmaram essa percepção.

Surgiu também a sugestão de incluir um diagrama de componentes no relatório de evidências para tornar explícito o fluxo do experimento, da geração de requisições até o processamento final.

Escolha tecnológica: desempenho e complexidade de implementação

Os participantes reconheceram o Kafka como o broker de melhor desempenho e o Protobuf como o serializador mais rápido, em linha com os resultados quantitativos. A diferença de desempenho do Kafka em relação ao ActiveMQ foi percebida como expressiva, especialmente no volume de transações processadas por segundo. O Json foi identificado como o serializador de pior desempenho, embora seja o mais comum no mercado. Os formatos binários foram vistos como adequados para uso interno nos sistemas.

A discussão sobre complexidade de implementação foi recorrente. Os participantes compararam o Protobuf, que exige a criação de arquivos de especificação, com o Kryo, que trabalha diretamente com objetos Java sem configuração adicional. Foi observado que, se a dificuldade de implementação for alta e o ganho de desempenho não for proporcional,

a adoção de uma tecnologia pode não ser viável no dia a dia. O Kryo, por exemplo, oferece ganhos reais em relação ao Json com menor esforço de adoção do que o Protobuf.

A decisão de executar os testes com as configurações padrão de cada ferramenta também foi avaliada positivamente. Os participantes reconheceram essa escolha como uma forma justa de comparar as tecnologias, sem favorecer nenhuma delas com ajustes específicos.

Contexto regulatório, aplicabilidade e sugestões

Os participantes reconheceram que o Open Finance exige o uso de Json nas interfaces externas entre instituições, o que limita o uso de formatos binários à comunicação interna dos sistemas. A aplicabilidade dos resultados foi avaliada de forma positiva, com a ressalva de que mudanças entre camadas de integração representam um desafio prático para adoção em sistemas existentes. Por fim, um participante sugeriu testar outras tecnologias disponíveis em ambientes corporativos, como o IBM MQ, o que poderia gerar resultados distintos dos obtidos.

De forma geral, as três categorias temáticas confirmam os resultados quantitativos e apontam que a escolha tecnológica no Open Finance envolve, além do desempenho, fatores como complexidade de implementação, restrições regulatórias e viabilidade de adoção em sistemas existentes.

4.4.3 Discussão dos Resultados

Os resultados do grupo focal são coerentes com os resultados quantitativos apresentados. Os participantes confirmaram que o Kafka apresenta desempenho superior ao ActiveMQ nas métricas de tempo de resposta e TPS, e que formatos binários como Protobuf e Kryo superam o Json nessas mesmas métricas. As métricas de CPU e memória foram reconhecidas como pouco relevantes para a decisão tecnológica, o que também é consistente com os dados experimentais.

Um ponto levantado no grupo focal que vai além dos experimentos é o papel da complexidade de implementação. Os participantes indicaram que a dificuldade de adotar uma tecnologia pode ser decisiva, mesmo quando o ganho de desempenho é claro. Isso indica que estudos futuros podem incluir métricas de esforço de adoção, como curva de aprendizado e impacto no ciclo de desenvolvimento.

Outro ponto recorrente foi a restrição regulatória do Open Finance, que exige Json nas interfaces externas. Essa restrição não invalida os resultados, mas indica que os ganhos de desempenho identificados se aplicam principalmente ao núcleo interno dos sistemas.

A decisão de usar as configurações padrão de cada broker foi bem recebida pelos participantes, que a reconheceram como uma forma justa de comparar as tecnologias. Esse ponto é discutido na literatura sobre experimentos controlados em engenharia de software [41], que recomenda o uso de configurações padrão para garantir comparabilidade entre os tratamentos.

4.4.4 Limitações e Ameaças à Validade

Esta seção discute as restrições do experimento e os riscos que podem afetar a interpretação dos resultados. A análise segue o modelo de Wohlin et al. [42], dividida em limitações do estudo e ameaças à validade.

Limitações do Estudo

As principais limitações foram:

- **Infraestrutura e tecnologias escolhidas:** O estudo avaliou apenas dois brokers (ActiveMQ e Kafka) e cinco serializadores (Json, Protobuf, Protostuff, Kryo e MessagePack), deixando de fora outras tecnologias como RabbitMQ e Apache Pulsar. A execução de cenários com até 10 milhões de requisições gerou custos de infraestrutura e tempo que limitaram o número de variações testadas.
- **Uso de configurações padrão:** Os testes foram executados com as configurações padrão (*defaults*) de cada ferramenta, o que garante comparabilidade, mas não permite avaliar como cada tecnologia se comportaria com ajustes específicos de desempenho (*tuning*).
- **Restrição regulatória do Open Finance:** O formato Json é obrigatório nas interfaces externas entre instituições. Isso limita a aplicação direta de formatos binários como o Protobuf à comunicação interna dos sistemas.
- **Tamanho da amostra na etapa qualitativa:** O grupo focal contou com quatro participantes. As percepções coletadas foram consistentes com os dados quantitativos, mas o número reduzido de participantes não permite generalizações para toda a indústria.

Ameaças à Validade

As ameaças à validade são organizadas nas quatro categorias do modelo de Wohlin et al. [42]:

- **Validade de Conclusão:** A variação irregular dos dados na métrica de TPS pode afetar a precisão das conclusões estatísticas. Para CPU e memória, a alta variância residual reduz o poder do teste, o que aumenta o risco de não detectar diferenças reais entre as tecnologias (erro do Tipo II).
- **Validade Interna:** Comportamentos da JVM como *warm-up* e *Garbage Collection*, e variações de rede, podem influenciar os resultados. Para reduzir esse risco, os testes foram repetidos múltiplas vezes em condições padronizadas.
- **Validade de Constructo:** As métricas escolhidas (Tempo de Resposta, TPS, CPU e Memória) cobrem os aspectos centrais de desempenho, mas deixam de fora métricas complementares como latência (p95/p99), consumo de disco (I/O) e efeito de envio em lotes (*batch*).
- **Validade Externa:** Os resultados estão vinculados às versões e configurações das ferramentas testadas e ao tamanho das mensagens do modelo Open Finance. Ambientes com topologias diferentes, como Kafka com mais partições ou com replicação ativa, podem apresentar resultados distintos.

Capítulo 5

Conclusão e Trabalhos Futuros

Este capítulo apresenta as considerações finais desta pesquisa, sintetizando os principais achados. Além disso, são exploradas possibilidades de expansão e aprofundamento do estudo por meio de trabalhos futuros, que visam ampliar o escopo e a aplicabilidade das contribuições aqui apresentadas.

5.1 Conclusão

A presente pesquisa destaca a importância estratégica da escolha de brokers e serializadores em arquiteturas de microsserviços, especialmente no contexto de sistemas de alta performance, como aqueles aplicados ao Open Finance. A análise demonstrou que o desempenho de um sistema não depende exclusivamente de componentes individuais, mas da interação entre eles, reforçando a necessidade de uma abordagem integrada no design de soluções tecnológicas.

Os resultados evidenciaram que diferentes combinações de brokers e serializadores podem atender a demandas distintas, sejam elas relacionadas à eficiência computacional ou ao desempenho operacional. No entanto, a variabilidade das métricas e a complexidade das interações apontam para um equilíbrio delicado entre o consumo de recursos e a performance. Essa dualidade sublinha a importância de escolhas informadas, baseadas nas prioridades específicas do sistema, como vazão transacional, tempos de resposta ou restrições de hardware.

Outra contribuição desta pesquisa foi a proposta de uma arquitetura experimental para avaliação conjunta de brokers de mensageria e tecnologias de serialização no contexto do Open Finance. A arquitetura foi implementada de forma reproduzível, acompanhada por scripts de execução, coleta automática das métricas e um pacote de replicação que permite reproduzir os experimentos e utilizar o ambiente em avaliações futuras. Além da

comparação entre as tecnologias avaliadas, o ambiente possibilita a coleta de métricas de desempenho e de consumo de energia utilizando o mesmo procedimento experimental.

Por fim, este trabalho reforça que a eficiência de arquiteturas modernas depende de uma análise criteriosa de suas múltiplas dimensões, evidenciando a necessidade de alinhamento constante entre os objetivos de negócios e os recursos tecnológicos disponíveis.

5.2 Limitações e Perspectivas Futuras

Como todo estudo experimental, esta pesquisa apresenta limitações que precisam ser reconhecidas, mas que ao mesmo tempo abrem caminhos relevantes para novas investigações. Entre as principais limitações, destacam-se:

- **Erros-padrão e intervalos de confiança:** a análise de TPS utilizou testes convencionais, sem empregar variantes robustas (como HC3), o que pode influenciar a precisão dos intervalos de confiança.
- **Amostragem restrita:** para métricas de CPU e memória, a quantidade de execuções foi limitada, e não houve *blocking* sistemático da ordem de execução, o que pode ter introduzido ruído experimental.
- **Configurações fixas dos brokers:** parâmetros como `acks`, compressão, tamanho de lote/`linger`, `prefetch`, número de partições, consumidores e replicação foram mantidos em valores padrão, restringindo a avaliação do impacto de configurações alternativas.
- **Variação limitada de payload:** foram testados apenas cenários básicos de mensagens, sem explorar tamanhos diferentes, esquemas binários vs. textuais, ou evolução de esquemas.
- **Conjunto reduzido de serializadores:** a comparação concentrou-se em Json, Protobuf, Protostuff, Kryo e MsgPack, não abrangendo alternativas como Avro, FlatBuffers ou Cap'n Proto.
- **Foco em médias:** a avaliação se baseou majoritariamente em médias agregadas, sem detalhar percentis de latência (p95/p99) ou estabilidade temporal em séries de longo prazo.
- **Reprodutibilidade parcial:** embora os experimentos tenham sido descritos, o pipeline não foi completamente automatizado nem acompanhado de publicação de scripts e artefatos fixados em versão.

Essas limitações, entretanto, delineiam uma agenda promissora de trabalhos futuros, que podem ampliar a robustez dos resultados e a aplicabilidade prática no contexto de Open Finance. Entre as principais perspectivas, destacam-se:

- **Exploração de parâmetros avançados dos brokers**, incluindo replicação, particionamento, compressão e políticas de retenção, sob diferentes cargas de trabalho.
- **Integração com protocolos de rede diversos**, como HTTP/3, gRPC e WebSockets, para avaliar efeitos de interoperabilidade e desempenho em cenários distribuídos.
- **Inclusão de novos serializadores**, como Avro, FlatBuffers e Cap'n Proto, explorando ainda mecanismos de *schema registry* e compatibilidade entre versões.
- **Aplicação de percentis de latência (p95/p99)** e análise de **estabilidade temporal**, de modo a capturar variações mais finas de desempenho.
- **Uso de aprendizado de máquina** para estratégias de otimização automática de parâmetros de brokers e serializadores.
- **Avaliação de arquiteturas alternativas**, como modelos *serverless*, em comparação com microsserviços tradicionais, para workloads específicos do Open Finance.

Portanto, ao mesmo tempo em que este estudo avança no entendimento do impacto das escolhas de brokers e serializadores em arquiteturas de microsserviços aplicadas ao Open Finance, ele também abre espaço para novas pesquisas que aprofundem a robustez metodológica, ampliem os cenários avaliados e fortaleçam a adoção de soluções mais eficientes e sustentáveis em sistemas financeiros distribuídos.

Referências

- [1] Brasil, Open Finance: *Relatório semestral 2022*. <https://openfinancebrasil.org.br/relatorio-semestral-agosto-2022/>, 2022. 1
- [2] Brasil, Open Finance: *Open finance: o que é, benefícios e como funciona*. <https://openfinancebrasil.org.br>, 2021. 5, 6
- [3] Aguiar, Tiago: *Open banking and its impacts on the future of financial system*. The Paypers, 2022. <https://thepappers.com/thought-leader-insights/open-banking-and-its-impacts-on-the-future-of-financial-system--1254270>. 5
- [4] Wijnen, Tom: *From open banking to open finance: Not if, but when does it happen?* The Paypers, 2023. <https://thepappers.com/thought-leader-insights/from-open-banking-to-open-finance-not-if-but-when-does-it-happen--1260341>. 5
- [5] Fowler, Martin: *Microservices: a definition of this new architectural term*. <https://martinfowler.com/articles/microservices.html>, 2023. 7
- [6] Newman, Sam: *Building Microservices: Designing Fine-Grained Systems*. O'Reilly Media, 2015. 7, 8
- [7] Richardson, Chris: *Microservice patterns: With examples in Java*. Manning Publications, 2018. 7
- [8] Lewis, James e Martin Fowler: *Microservices: a practitioner's guide*. O'Reilly Media. 7
- [9] Dragoni, Nicola, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin e Larisa Safina: *Microservices: yesterday, today, and tomorrow*. arXiv preprint arXiv:1606.04036, 2017. 8
- [10] Giamattei, Luca, Antonio Guerriero, Ivano Malavolta, Cristian Mascia, Roberto Pietrantuono e Stefano Russo: *Identifying performance issues in microservice architectures through causal reasoning*. Em *5th ACM/IEEE International Conference on Automation of Software Test (AST)*, páginas 149–153, Lisbon, Portugal, 2024. ACM. 8
- [11] Fielding, Roy Thomas: *Architectural styles and the design of network-based software architectures*. https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm, 2000. 8, 9

- [12] al., David Booth et: *Web services architecture*. <https://www.w3.org/TR/2004/NOTE-ws-arch-20040211/#relwwwrest>, 2004. 9
- [13] Pautasso, Cesare, Erik Wilde e Rosa Alarcón (editores): *REST: From Research to Practice*. Springer, 2011. 9
- [14] Farias, Otávio: *Restful grounding: A comprehensive study*. <https://github.com/otaviofff/restful-grounding/blob/master/papers/official-paper-icwi.pdf>, 2009. 9
- [15] Ferg, Stephen: *Event-driven programming: Introduction, tutorial, history*, 2006. https://fsw01.bcc.cuny.edu/kerry.ojakian/TeachingPages/Old_Courses_UP/CSI32_Fall2019/CourseMaterials/Ferg_event_driven_programming.pdf. 9
- [16] Fowler, Martin: *Event collaboration*, 2014. <https://www.martinfowler.com/eaadDev/EventCollaboration.html>. 9
- [17] Hohpe, Gregor e Bobby Woolf: *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley, 2004. 10
- [18] Kreps, Jay, Neha Narkhede e Jun Rao: *Kafka: A distributed messaging system for log processing*. Em *Proceedings of the ACM Workshop on Networking Meets Databases*, 2011. 10
- [19] Danushi, Ornela, Stefano Forti e Jacopo Soldani: *Environmentally sustainable software design and development: A systematic literature review*. arXiv preprint, arXiv:2407.19901, 2024. <https://arxiv.org/abs/2407.19901>. 10
- [20] Benavente, Victor, Ciro Rodriguez, Luigi Yantas, Ricardo Inquilla, Isabel Moscol e Yuri Pomachagua: *Comparative analysis of microservices and monolithic architecture*. Em *14th IEEE International Conference on Computational Intelligence and Communication Networks (CICN)*, páginas 177–184, Lima, Peru, 2022. IEEE. 11
- [21] Singh, Amarjeet, Shalini Aggarwal, Vinay Singh e Alok Aggarwal: *Event driven architecture for message streaming data driven microservices systems residing in distributed version control system*. Em *International Conference on Innovations in Science and Technology for Sustainable Development (ICISTSD)*, páginas 308–312, Dehradun, India, 2022. IEEE. 11
- [22] Rahmatulloh, Alam, Fuji Nugraha, Rohmat Gunawan e Irfan Darmawan: *Event-driven architecture to improve performance and scalability in microservices-based systems*. Em *International Conference on Advancement in Data Science, E-learning and Information Systems (ICADEIS)*, páginas 1–6, Tasikmalaya, Indonesia, 2022. IEEE. 12
- [23] Jackson, Samuel, Nathan Cummings e Saiful Khan: *Streaming technologies and serialization protocols: Empirical performance analysis*. arXiv preprint arXiv:2407.13494, 2024. <https://arxiv.org/abs/2407.13494>. 12

- [24] Fu, Guo, Yanfeng Zhang e Ge Yu: *A fair comparison of message queuing systems*. IEEE Access, 9:421–432, 2021. <https://ieeexplore.ieee.org/abstract/document/9303425>, Received December 2, 2020, accepted December 15, 2020, date of publication December 22, 2020, date of current version January 4, 2021. 12, 13, 14
- [25] Dobbelaere, Philippe e Kyumars Sheykh Esmaili: *Kafka versus rabbitmq: A comparative study of two industry reference publish subscribe implementations*. Em *Proceedings of DEBS '17*, páginas 1–12. ACM, 2017. 13, 17
- [26] Chy, Md Showkat Hossain, Muhammad Ashfakur Rahman Arju, Sri Manjusha Tella e Tomas Cerny: *Comparative evaluation of java virtual machine-based message queue services: A study on kafka, artemis, pulsar, and rocketmq*. Electronics, 12(23):4792, 2023. <https://www.mdpi.com/1996-1073/12/23/4792>, Received: 18 October 2023; Revised: 22 November 2023; Accepted: 23 November 2023; Published: 27 November 2023. 14
- [27] Akbulut, Akhan e Harry G. Perros: *Performance analysis of microservice design patterns*. IEEE Internet Computing, 23(6):19–27, 2019. 14
- [28] Bux, Richard e Govind Sreekar Shenoy: *Performance analysis of restful web services and rabbitmq for microservices based systems on cloud environment*. Em *2024 3rd International Conference for Innovation in Technology (INOCON)*, páginas 1–10, Karnataka, India, 2024. IEEE. 14
- [29] Kumar, Shashank, Aryan Jadon e Sachin Sharma: *Global message ordering using distributed kafka clusters*. Em *Proceedings of the 15th International Conference on Innovations in Information Technology (IIT)*. IEEE, 2023. <https://ieeexplore.ieee.org/abstract/document/10366422>, Received July 12, 2024. 15
- [30] Guo, Kaifeng e Rongjian Gao: *Distributed healthcare information system based on high-concurrency microservice architecture*. Em *4th International Conference on Information Science, Parallel and Distributed Systems (ISPDS)*, páginas 44–47, Fujian, China, 2023. IEEE. 15
- [31] Viotti, Juan Cruz e Mital Kinderkhedia: *A survey of json-compatible binary serialization specifications*. Preprint. Under review, 2022. <https://www.jviotti.com>. 15
- [32] Luis Álvaro, Pablo Casares, Juan J. Cuadrado-Gallego e Miguel A. Patricio: *Pson: A serialization format for iot sensor networks*. Sensors, 21(13):4559, 2021. <https://doi.org/10.3390/s21134559>, Received: 27 May 2021; Accepted: 30 June 2021; Published: 2 July 2021. 15
- [33] Maeda, Kazuaki: *Performance evaluation of object serialization libraries in xml, json and binary formats*. Em *IEEE Conference on Smart Computing and Systems Engineering*, páginas 177–182, Aichi, Japan, 2012. IEEE. 15

- [34] Sumaray, Audie e S. Kami Makki: *A comparison of data serialization formats for optimal efficiency on a mobile platform*. Em *International Conference on Ubiquitous Information Management and Communication (ICUIMC)*, páginas 1–8, Kuala Lumpur, Malaysia, 2012. ACM. 16
- [35] Pradana, Muhammad Adna, Andrian Rakhmatsyah e Aulia Arif Wardana: *Flat-buffers implementation on mqtt publish/subscribe communication as data delivery format*. Em *Proceedings of the International Conference on Electrical Engineering, Computer Science and Informatics (EECSI)*, páginas 142–146, Bandung, Indonesia, 2019. IEEE. 16
- [36] Jedlitschka, Andreas, Marcus Ciolkowski e Dietmar Pfahl: *Reporting experiments in software engineering*. Guide to Advanced Empirical Software Engineering, páginas 201–228, 2008. 18
- [37] Juristo, Natalia e Ana M. Moreno: *Basics of Software Engineering Experimentation*. Springer Science & Business Media, 2001. 18
- [38] Courty, Benoit, Victor Schmidt *et al.*: *mlco2/codecarbon: v2.4.1*, 2024. <https://doi.org/10.5281/zenodo.11171501>. 23, 27
- [39] Guimarães, Allysson e Rodrigo Andrade: *Comparing the environmental footprint of web stacks in the realworld apps*. Em *SBCARS*, 2025. 23
- [40] Braun, Virginia e Victoria Clarke: *Using thematic analysis in psychology*. *Qualitative Research in Psychology*, 3(2):77–101, 2006. 39
- [41] Wohlin, Claes, Per Runeson, Martin Host, Magnus C. Ohlsson, Bjorn Regnell e Anders Wesslen: *Experimentation in software engineering*. Springer Science & Business Media, 2012. 41
- [42] Wohlin, Claes, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell e Anders Wesslén: *Experimentation in Software Engineering*. Springer Science & Business Media, 2012. 41

Apêndice A

Apêndices

A.0.1 Apêndice A — Comparações Significativas (TukeyHSD)

Apenas comparações com significância estatística (conforme saídas fornecidas).

Tempo de Resposta

Broker

Tabela A.1: Comparação de broker para tempo de resposta (ANOVA/Tukey).

Comparação	diff	IC95%	p_{adj}	Interpretação
KAFKA – ACTIVE_MQ	−5309,33	[−5477,58, −5141,09]	< 0,001	Kafka menor (mais rápido)

Serializer

Tabela A.2: Comparações pareadas por *serializer* para Tempo de Resposta (TukeyHSD).

Comparação	diff	IC95%	p_{adj}	Interpretação
KRYO – JSON	−994,33	[−1375,94, −612,73]	< 0,001	Kryo menor que JSON
PROTOBUF – JSON	−1197,33	[−1578,94, −815,73]	< 0,001	Protobuf me- nor que JSON

Tabela A.2: Comparações pareadas por *serializer* para Tempo de Resposta (continuação)

Comparação	diff	IC95%	p_{adj}	Interpretação
PROTOSTUFF – JSON	−1169,17	[−1550,77, −787,56]	< 0,001	Protostuff menor que JSON
MSGPACK – KRYO	891,67	[510,06, 1273,27]	< 0,001	MsgPack maior que Kryo
PROTOBUF – MSGPACK	−1094,67	[−1476,27, −713,06]	< 0,001	Protobuf menor que MsgPack
PROTOSTUFF – MSGPACK	−1066,50	[−1448,11, −684,89]	< 0,001	Protostuff menor que MsgPack

Transações por Segundo (TPS)

Broker

Tabela A.3: Comparação do fator *Broker* para TPS (TukeyHSD).

Comparação	diff	IC95%	p_{adj}	Interpretação
KAFKA – ACTIVE_MQ	2533,18	[2452,34, 2614,02]	< 0,001	Kafka maior (mais TPS)

Serializer

Tabela A.4: Comparações entre serializadores para TPS (TukeyHSD).

Comparação	diff	IC95%	p_{adj}	Interpretação
KRYO – JSON	781,94	[598,59, 965,30]	< 0,001	Kryo maior que JSON

Tabela A.4: Comparações entre serializadores para TPS
(continuação)

Comparação	diff	IC95%	p_{adj}	Interpretação
PROTOBUF – JSON	1020,60	[837,25, 1203,96]	$< 0,001$	Protobuf maior que JSON
PROTOSTUFF – JSON	1002,18	[818,82, 1185,53]	$< 0,001$	Protostuff maior que JSON
MSGPACK – KRYO	–717,98	[–901,33, –534,62]	$< 0,001$	MsgPack me- nor que Kryo
PROTOBUF – KRYO	238,66	[55,30, 422,01]	0,007	Protobuf maior que Kryo
PROTOSTUFF – KRYO	220,23	[36,88, 403,59]	0,014	Protostuff maior que Kryo
PROTOBUF – MSGPACK	956,63	[773,28, 1139,99]	$< 0,001$	Protobuf maior que MsgPack
PROTOSTUFF – MSGPACK	938,21	[754,86, 1121,56]	$< 0,001$	Protostuff maior que MsgPack

CPU e Memória Não foram observadas comparações significativas nos fatores Broker/Serializer (consistentes com ANOVA, $p > 0,05$).

A.0.2 Apêndice B — Diagnósticos dos Modelos (Rplots)

As figuras abaixo são as saídas padrão de diagnóstico do R (`plot.lm` e gráficos auxiliares). Elas incluem, tipicamente, (i) resíduos versus ajustados, (ii) Q–Q plot dos resíduos, (iii) escala–localização e (iv) alavancas/*Cook’s distance*. Mantemos os arquivos conforme exportados para transparência.

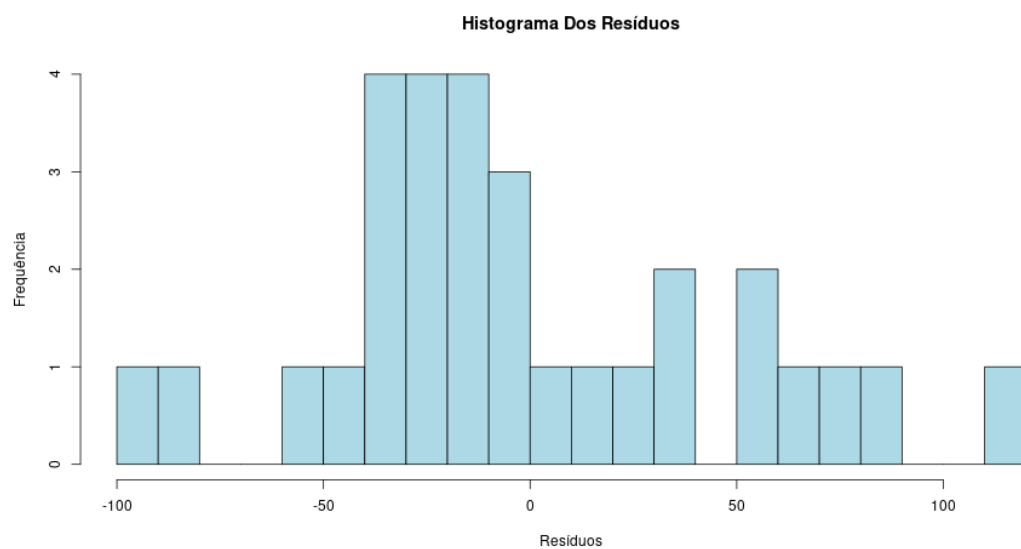


Figura A.1: Diagnóstico de modelo — saída Rplot001..

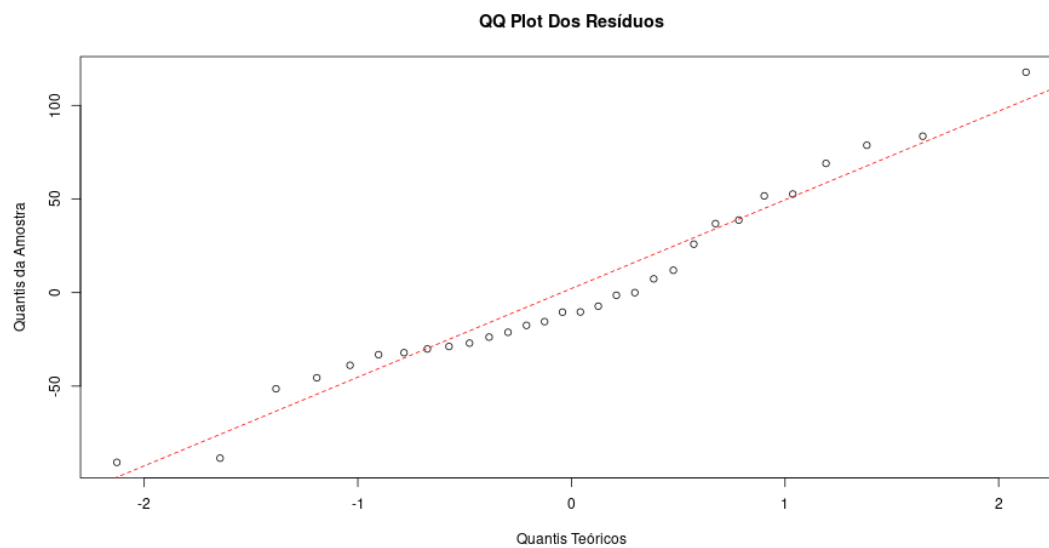


Figura A.2: Diagnóstico de modelo — saída Rplot002..

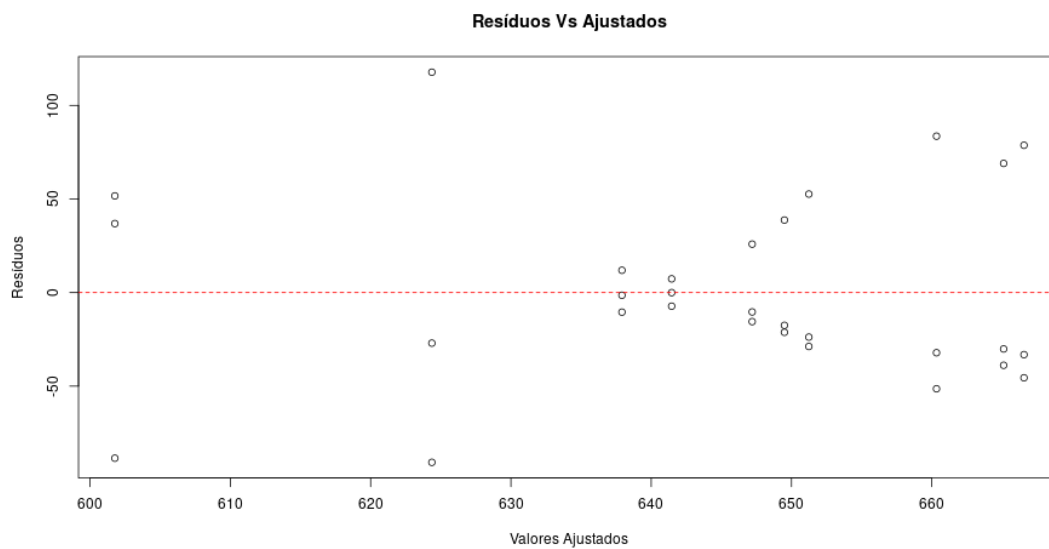


Figura A.3: Diagnóstico de modelo — saída Rplot003..

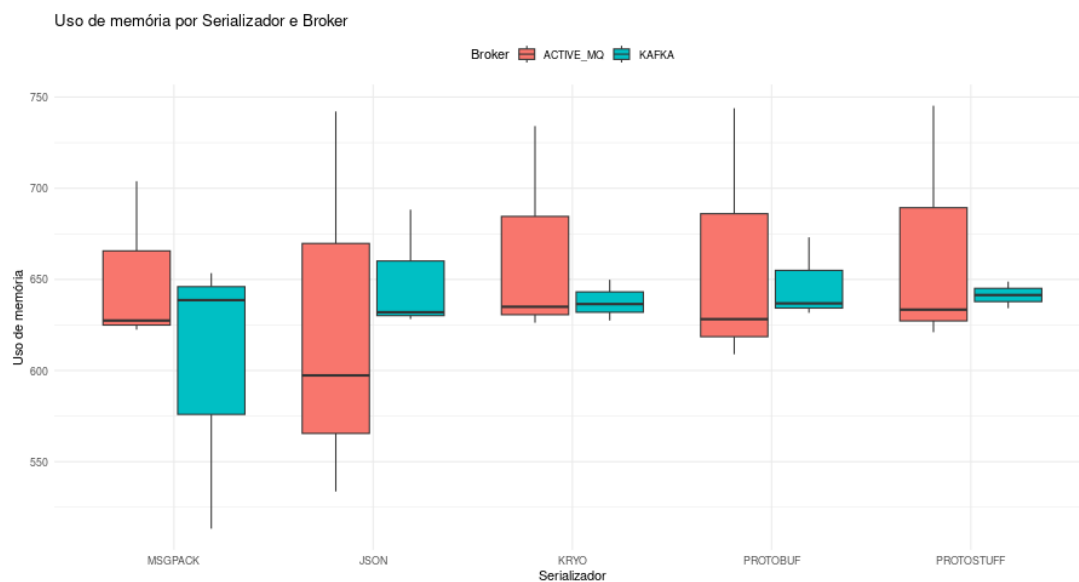


Figura A.4: Diagnóstico de modelo — saída Rplot004..

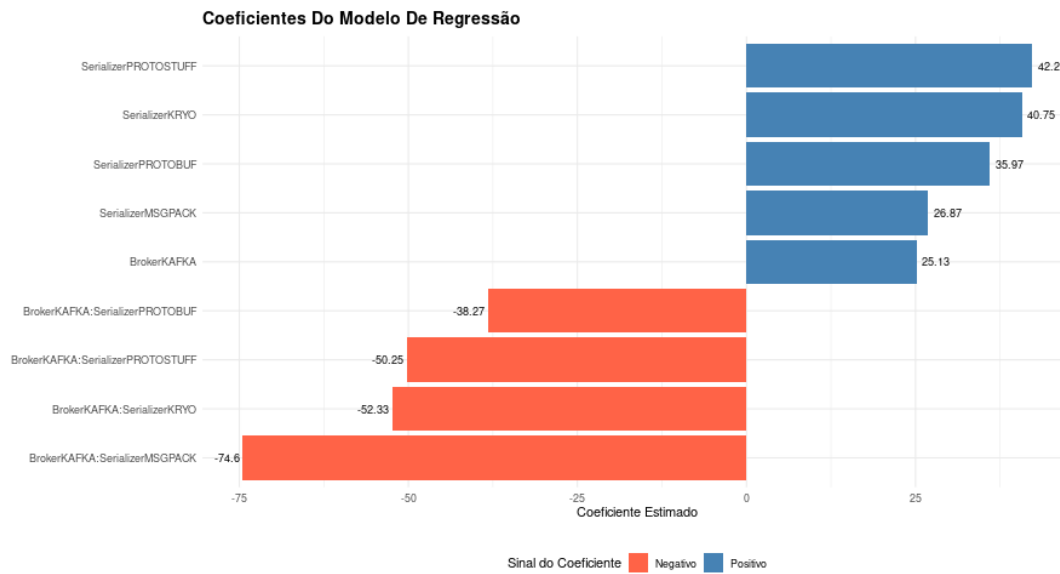


Figura A.5: Diagnóstico de modelo — saída Rplot005..

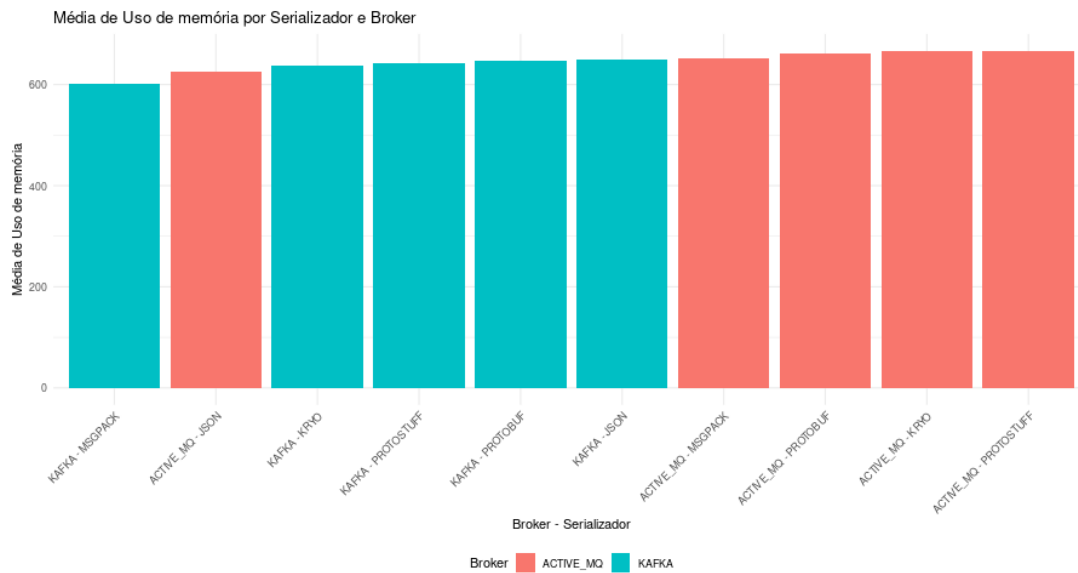


Figura A.6: Diagnóstico de modelo — saída Rplot006..

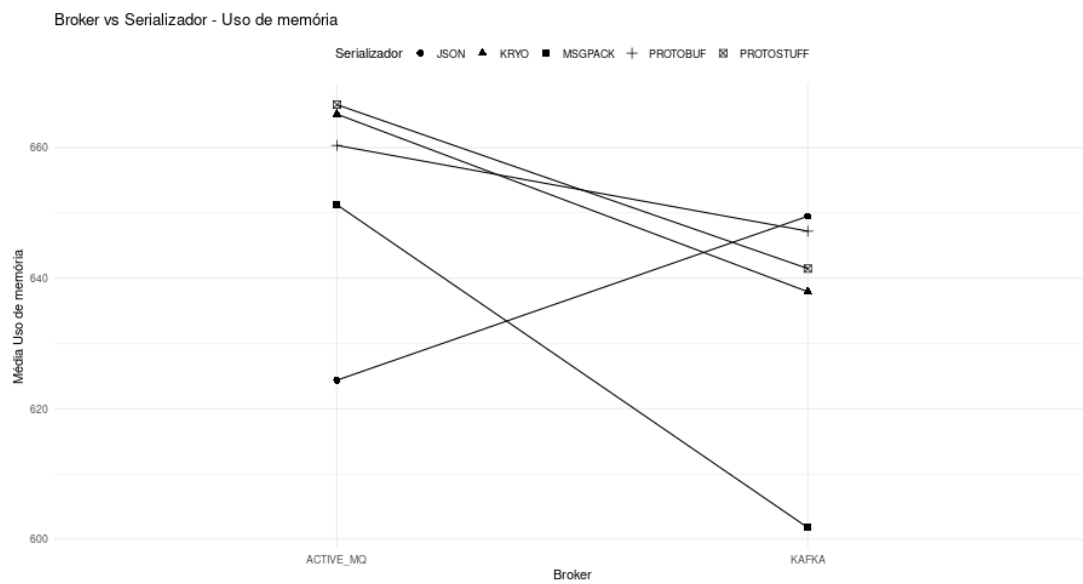


Figura A.7: Diagnóstico de modelo — saída Rplot007..

A.0.3 Apêndice C — Métricas Estatísticas (Conceitos)

ANOVA fatorial com interação: decompõe a variância do desfecho entre fatores (Broker, Serializador), interação e resíduo; a interação indica dependência do efeito de um fator em relação ao outro. **TukeyHSD:** compara pares de níveis controlando erro tipo I. **Shapiro–Wilk:** avalia normalidade dos resíduos. **Breusch–Pagan:** testa homocedasticidade. **R^2 :** proporção de variância explicada pelo modelo. **η^2 :** fração da soma de quadrados atribuída a cada componente; interpretamos como tamanho de efeito.

A.0.4 Apêndice D — Reprodutibilidade e Pipeline da Análise

A execução em R seguiu a função `executar_analise(data, melhor_maior = FALSE)`, que: (i) ajusta modelos (`aov`); (ii) interpreta coeficientes; (iii) avalia interação (comparação de modelos); (iv) analisa resíduos; (v) reporta R^2 ; (vi) gera predições com `predict()` (para métricas de minimização); (vii) executa TukeyHSD, Shapiro–Wilk e Breusch–Pagan; e (viii) exporta gráficos (barplots, boxplots, diagnósticos). Para TPS (onde maior é melhor), os resultados apresentados no corpo do texto baseiam-se em ANOVA/Tukey, evitando o artefato de minimização das predições automáticas.

Anexo I

Anexos

I.0.1 Galeria Completa de Figuras

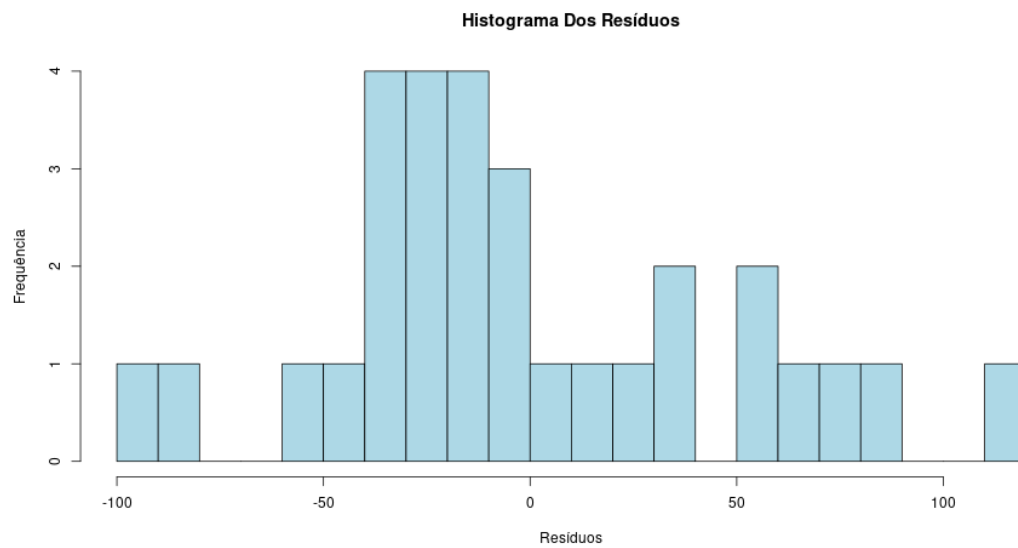


Figura I.1: Gráfico de diagnóstico dos resíduos do modelo estatístico (verificação de normalidade e homocedasticidade).

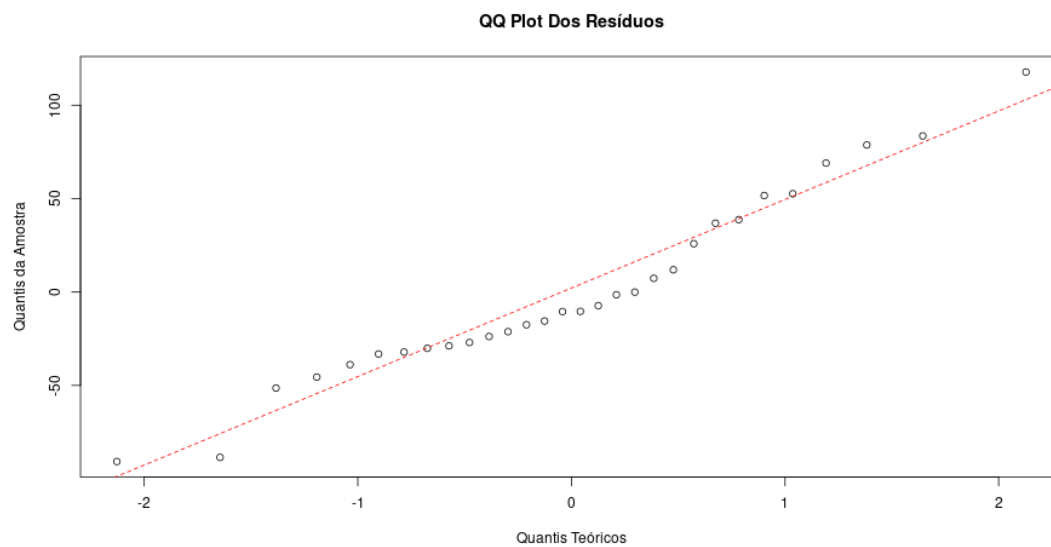


Figura I.2: Gráfico de diagnóstico dos resíduos do modelo estatístico (verificação de normalidade e homocedasticidade).

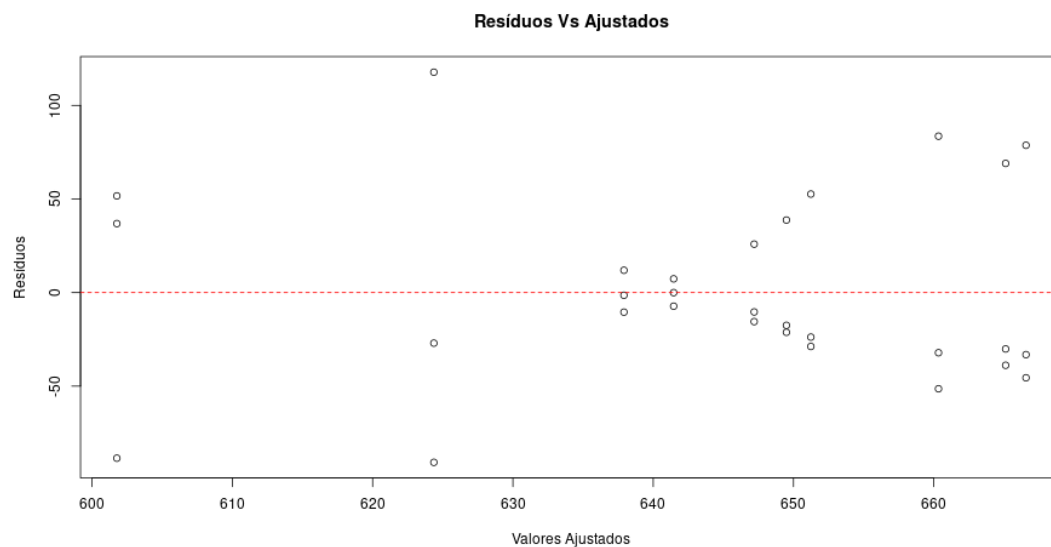


Figura I.3: Gráfico de diagnóstico dos resíduos do modelo estatístico (verificação de normalidade e homocedasticidade).

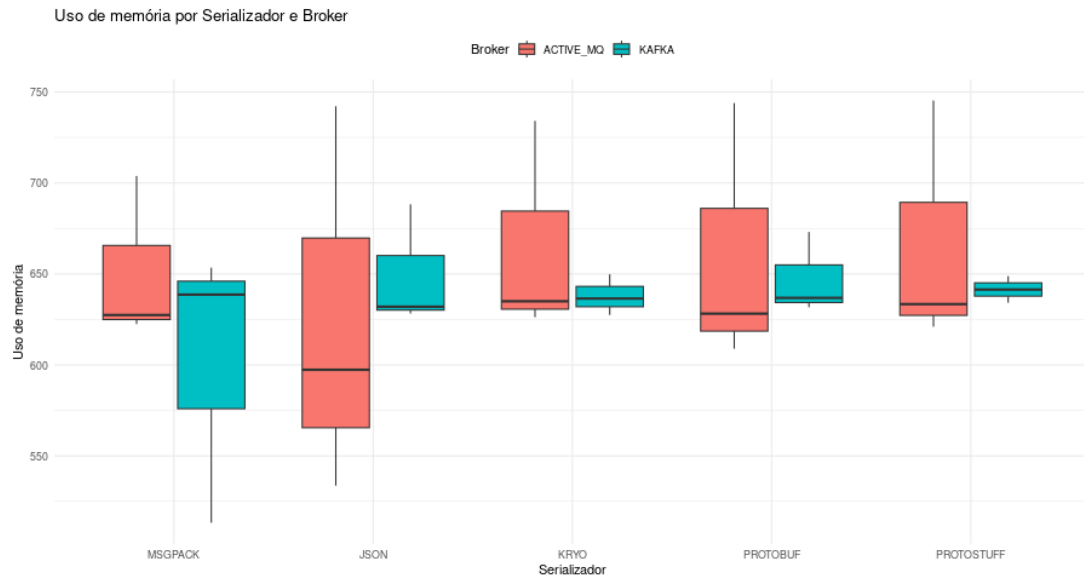


Figura I.4: Gráfico de diagnóstico dos resíduos do modelo estatístico (verificação de normalidade e homocedasticidade).

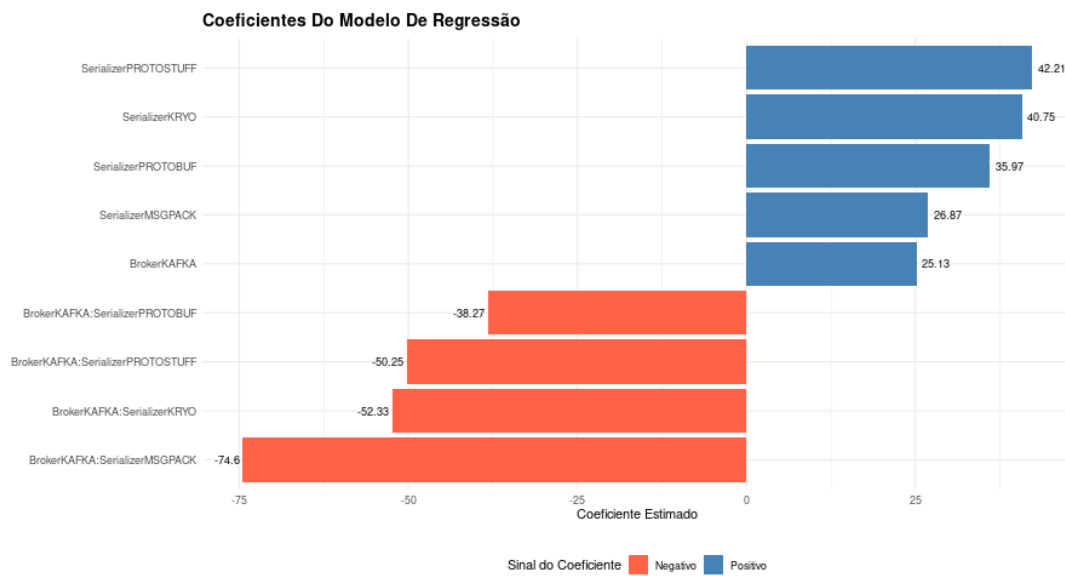


Figura I.5: Gráfico de diagnóstico dos resíduos do modelo estatístico (verificação de normalidade e homocedasticidade).

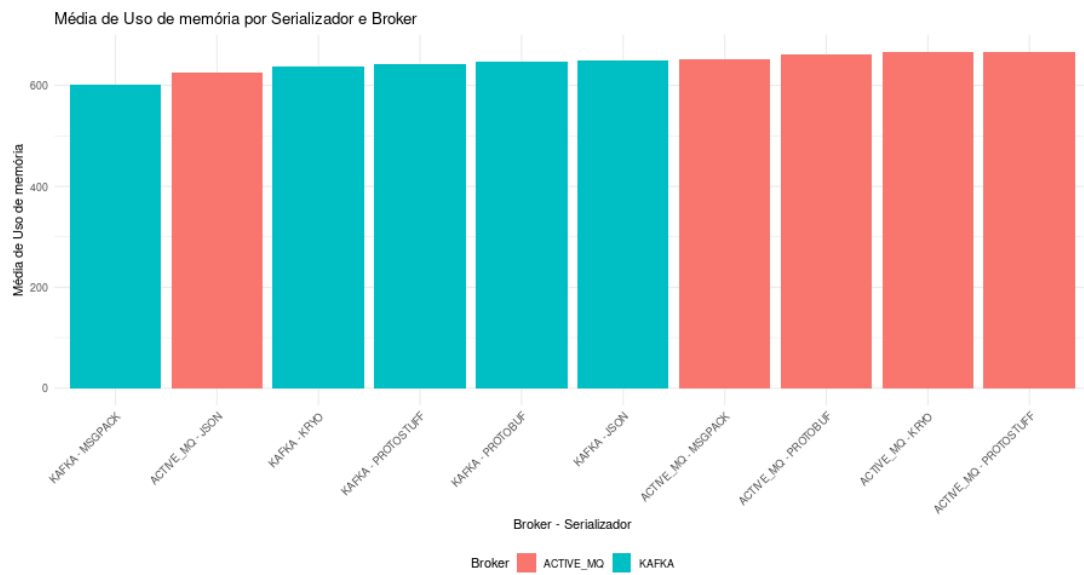


Figura I.6: Gráfico de diagnóstico dos resíduos do modelo estatístico (verificação de normalidade e homocedasticidade).

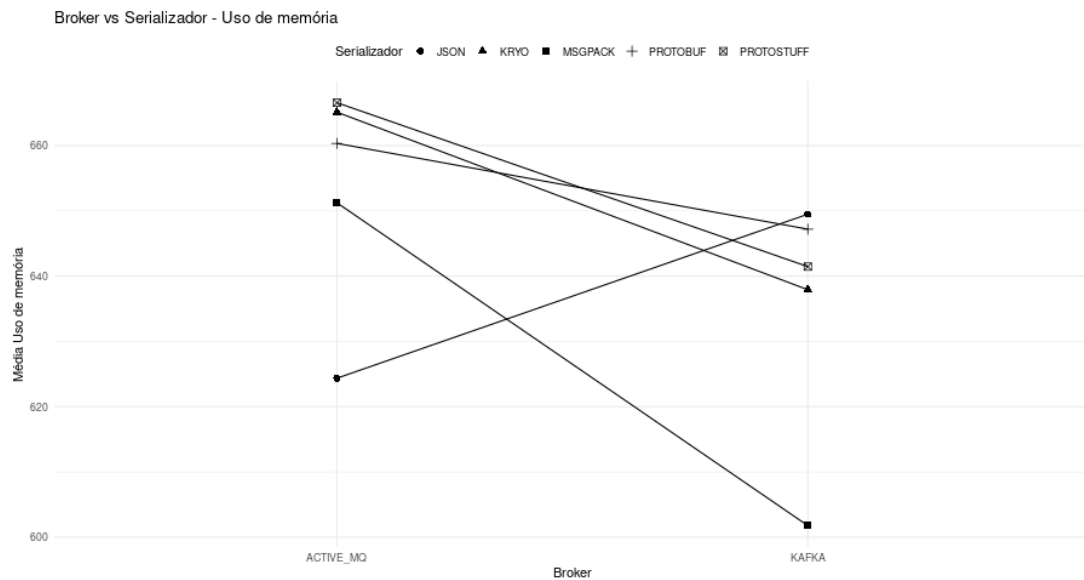


Figura I.7: Gráfico de diagnóstico dos resíduos do modelo estatístico (verificação de normalidade e homocedasticidade).

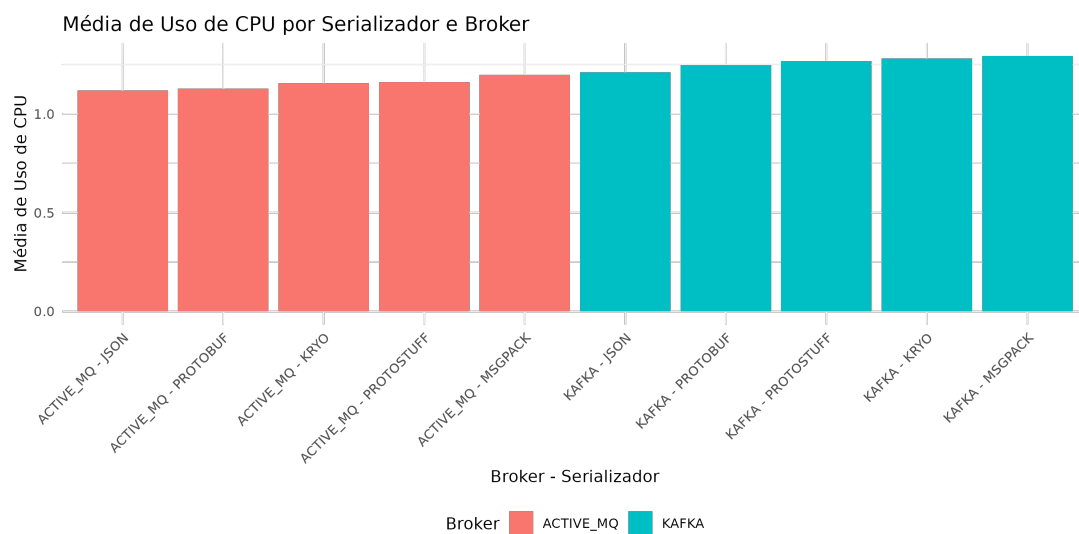


Figura I.8: Médias de uso de CPU por Broker e Serializador.

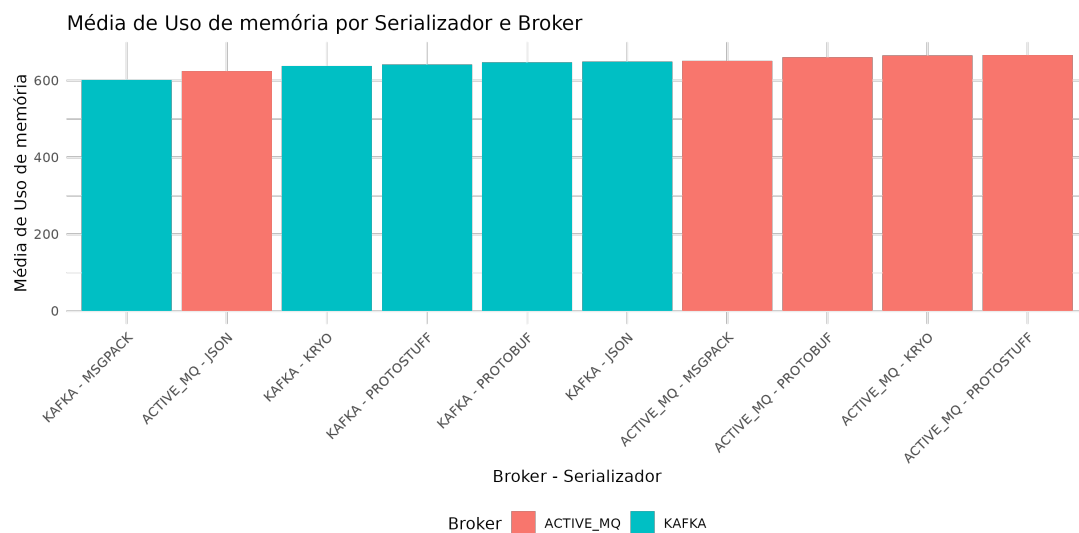


Figura I.9: Médias de uso de memória por Broker e Serializador.

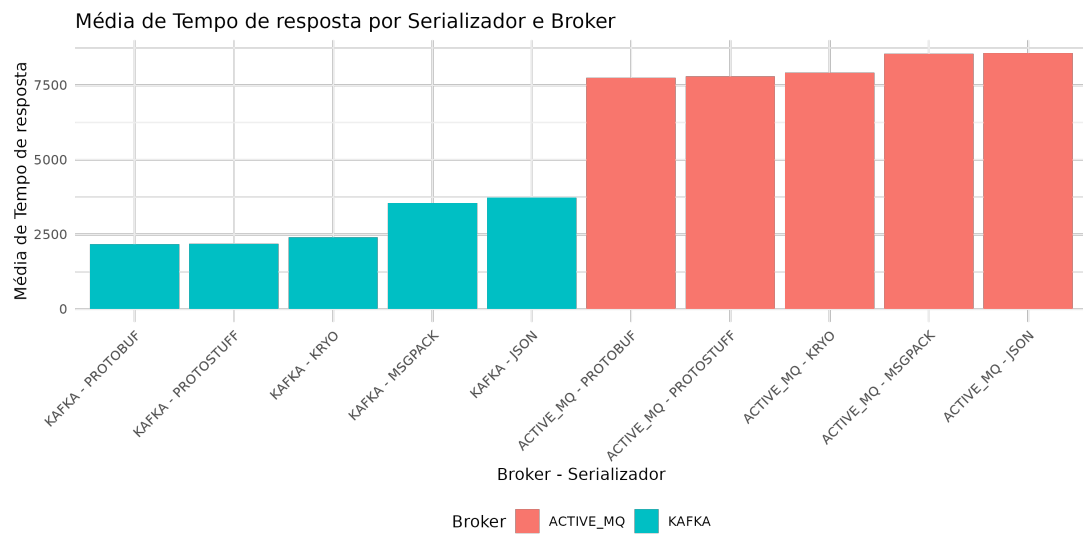


Figura I.10: Médias de tempo de resposta por Broker e Serializador.

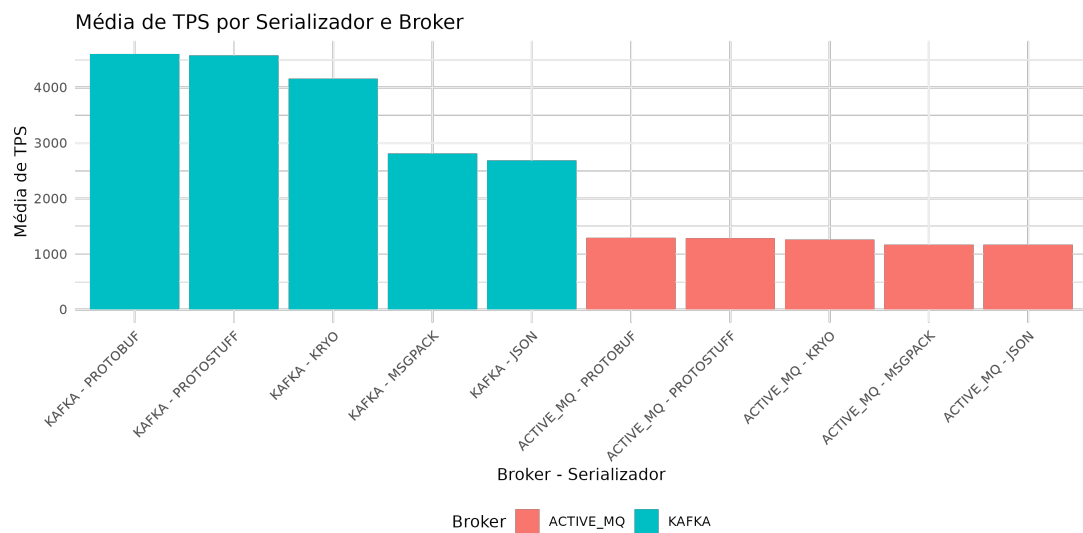


Figura I.11: Médias de transações por segundo (TPS) por Broker e Serializador.

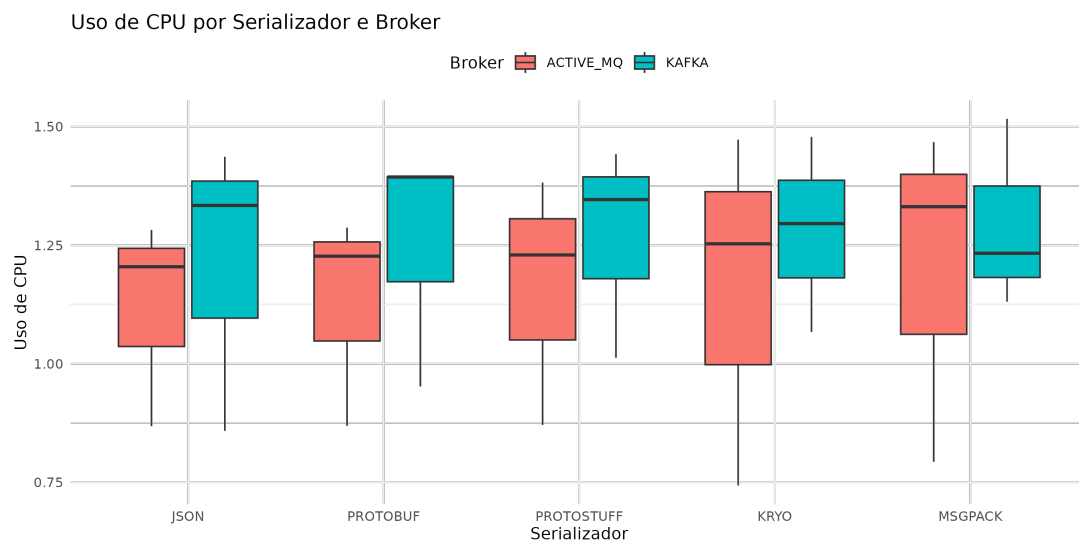


Figura I.12: Boxplot da distribuição de uso de CPU por Broker e Serializador.

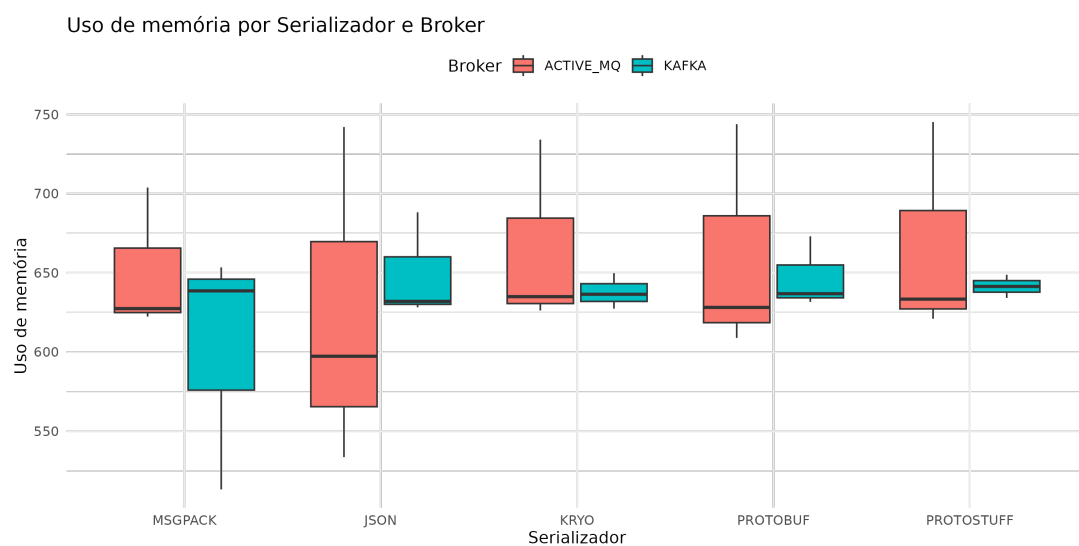


Figura I.13: Boxplot da distribuição de uso de memória por Broker e Serializador.

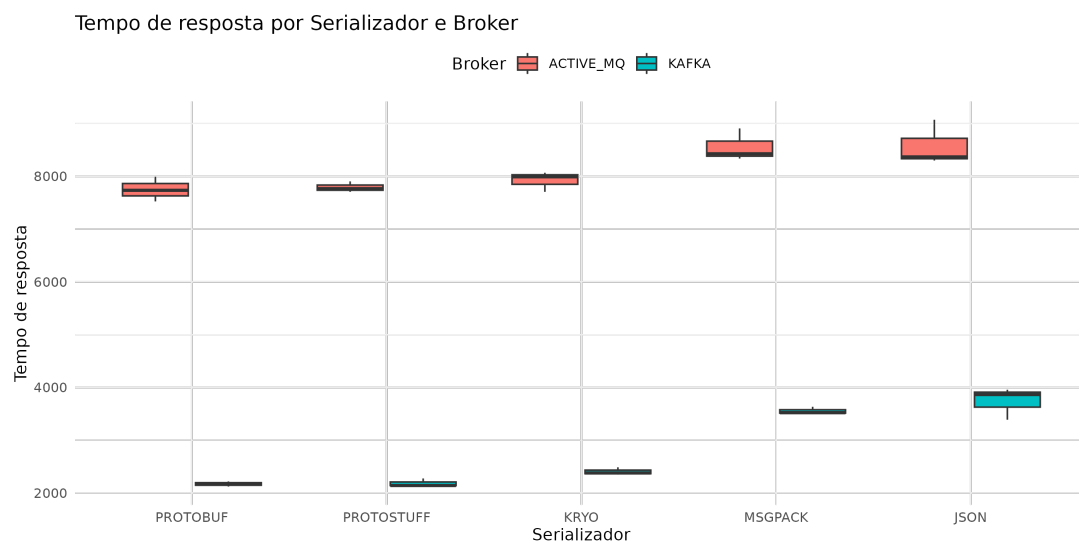


Figura I.14: Boxplot da distribuição do tempo de resposta por Broker e Serializador.

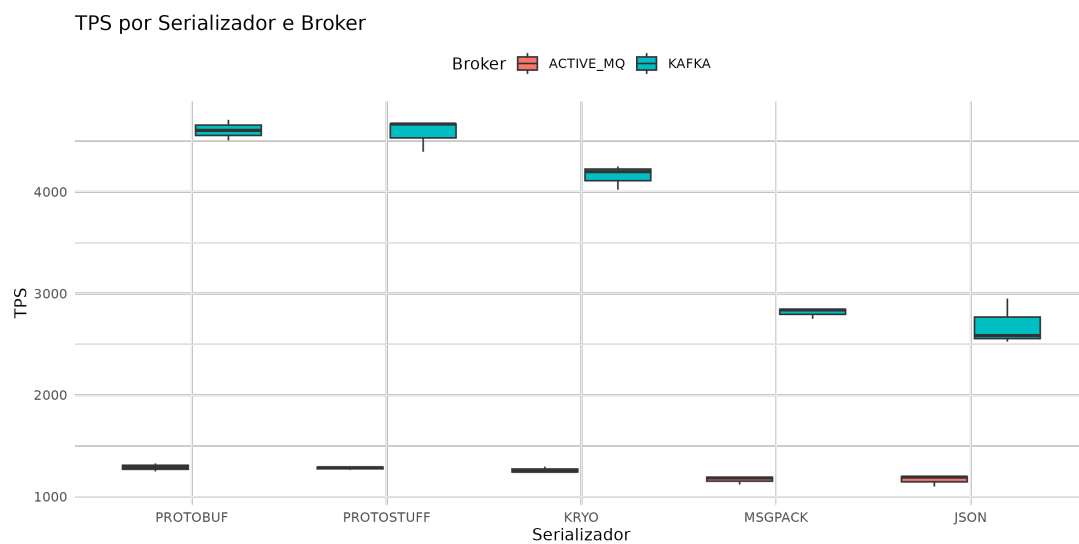


Figura I.15: Boxplot da distribuição de transações por segundo (TPS) por Broker e Serializador.

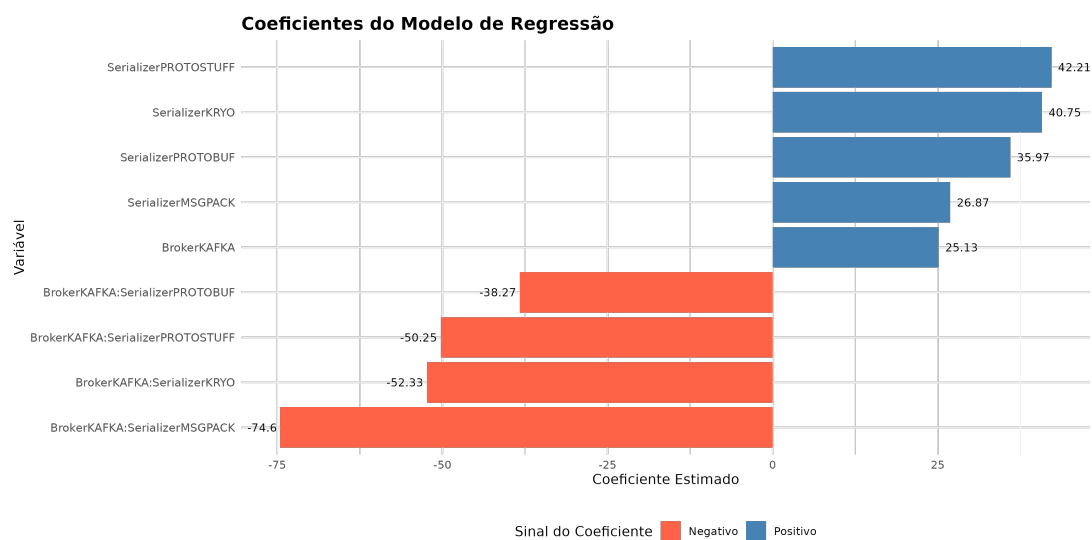


Figura I.16: Gráfico de coeficientes estimados do modelo com interação, destacando efeitos principais e termos de interação.

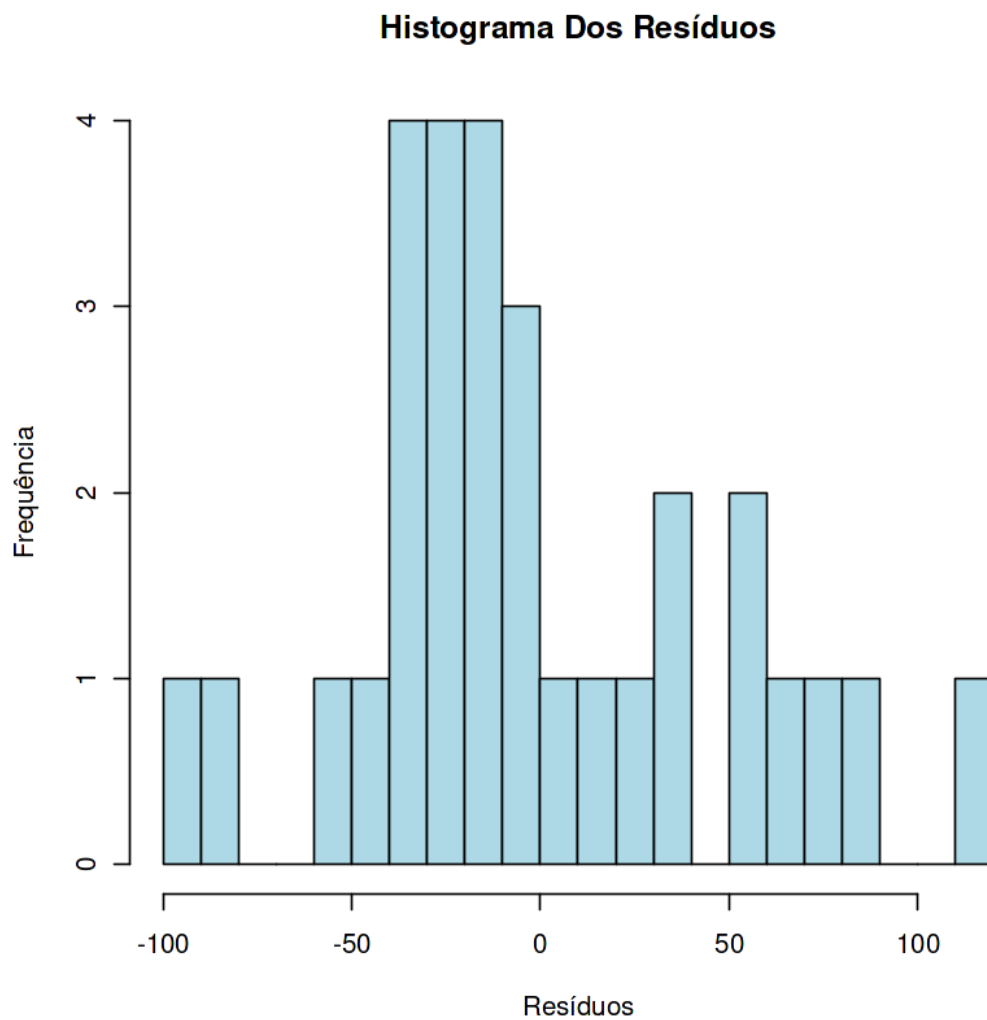


Figura I.17: Gráfico auxiliar do pipeline de análise estatística, utilizado para inspeção intermediária dos resultados.

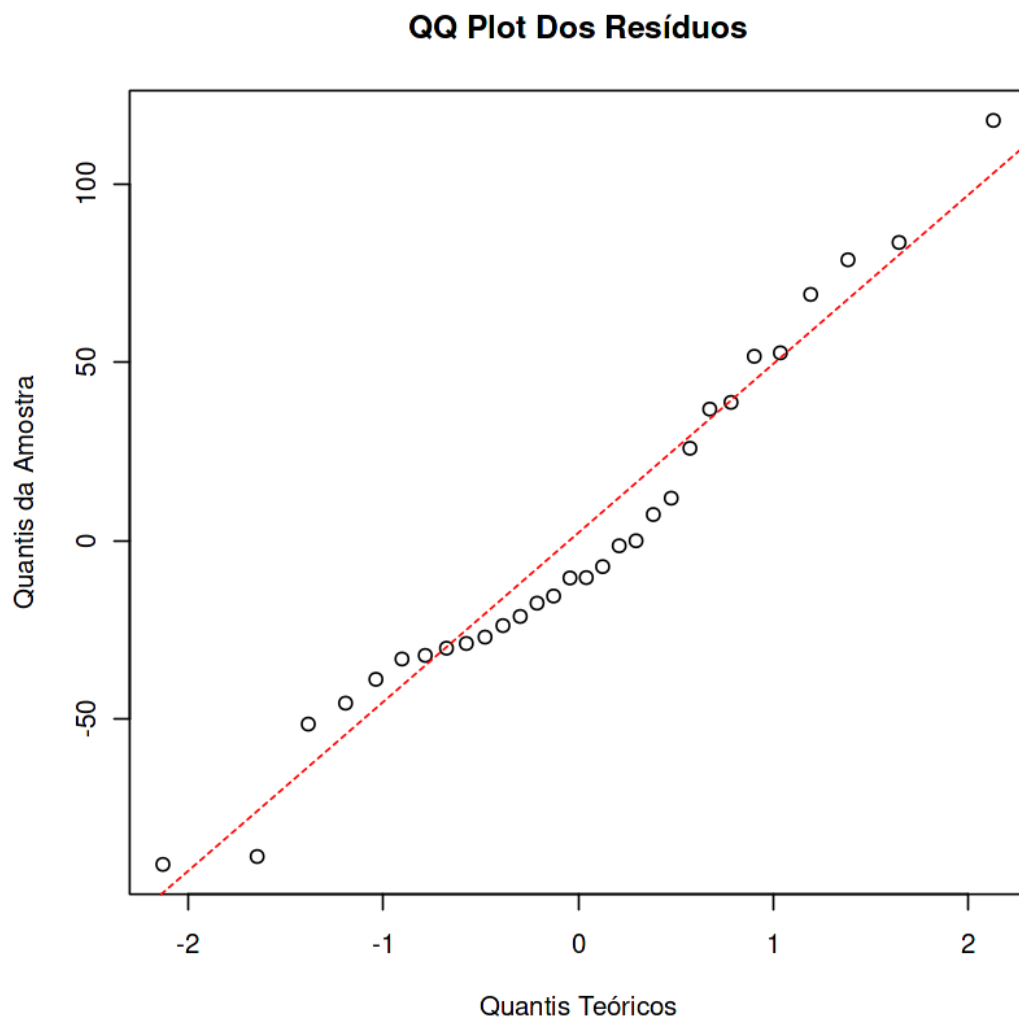


Figura I.18: Gráfico auxiliar do pipeline de análise estatística, utilizado para inspeção intermediária dos resultados.

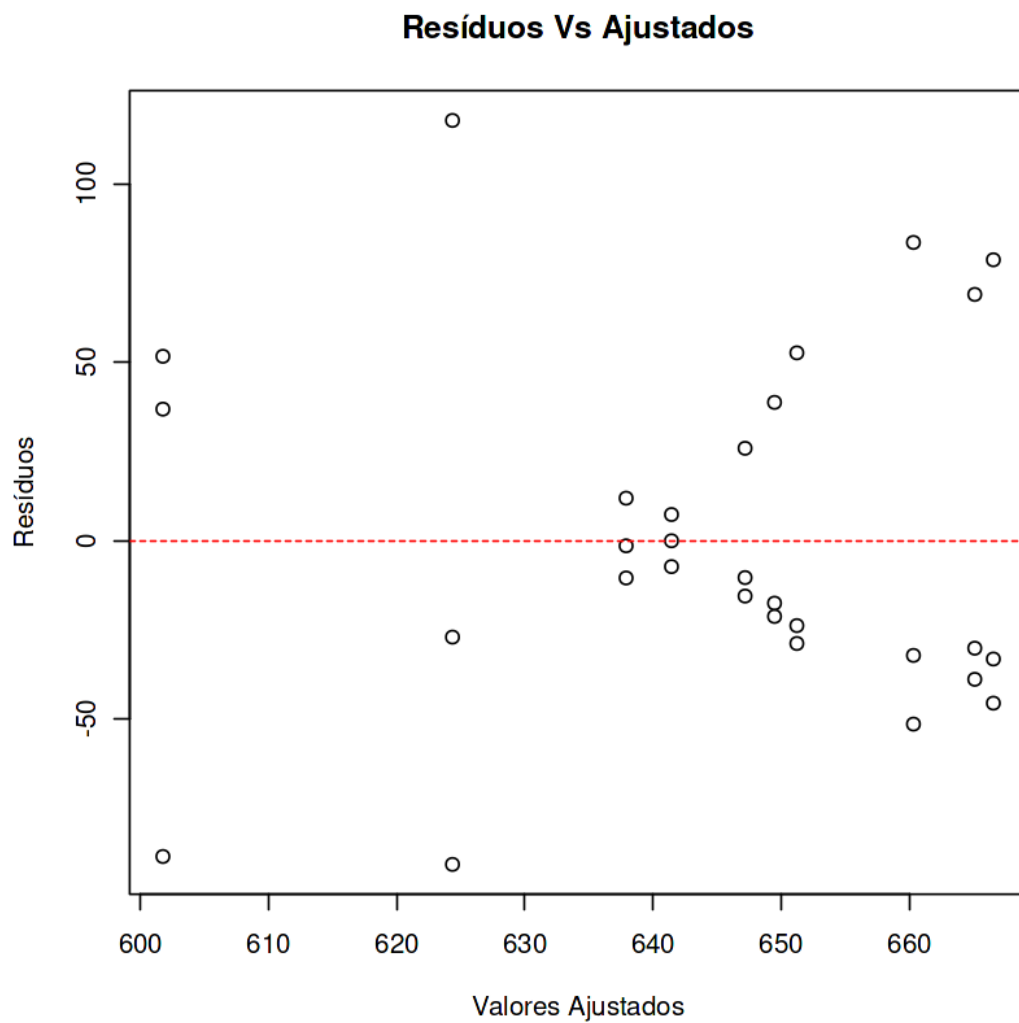


Figura I.19: Gráfico auxiliar do pipeline de análise estatística, utilizado para inspeção intermediária dos resultados.

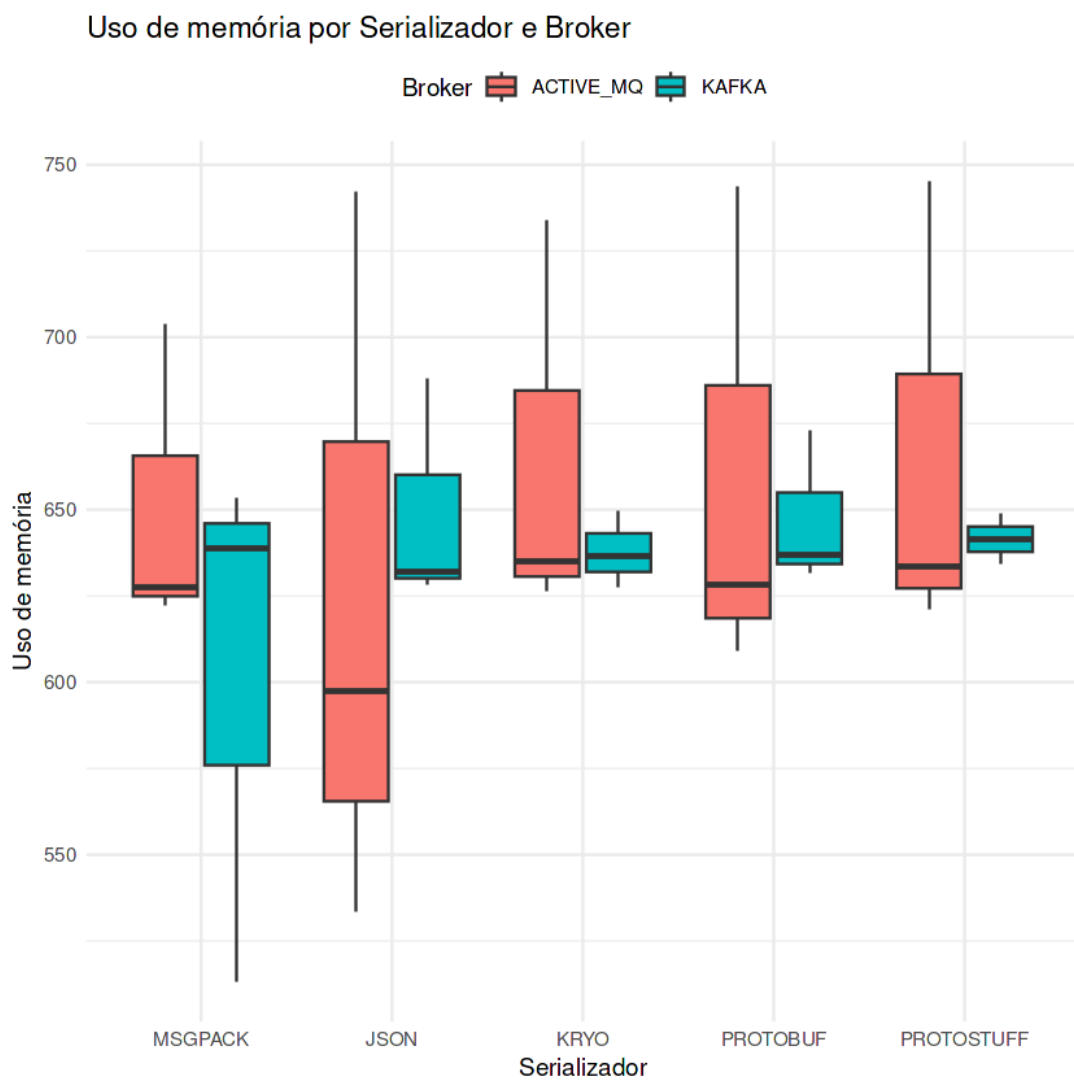


Figura I.20: Gráfico auxiliar do pipeline de análise estatística, utilizado para inspeção intermediária dos resultados.

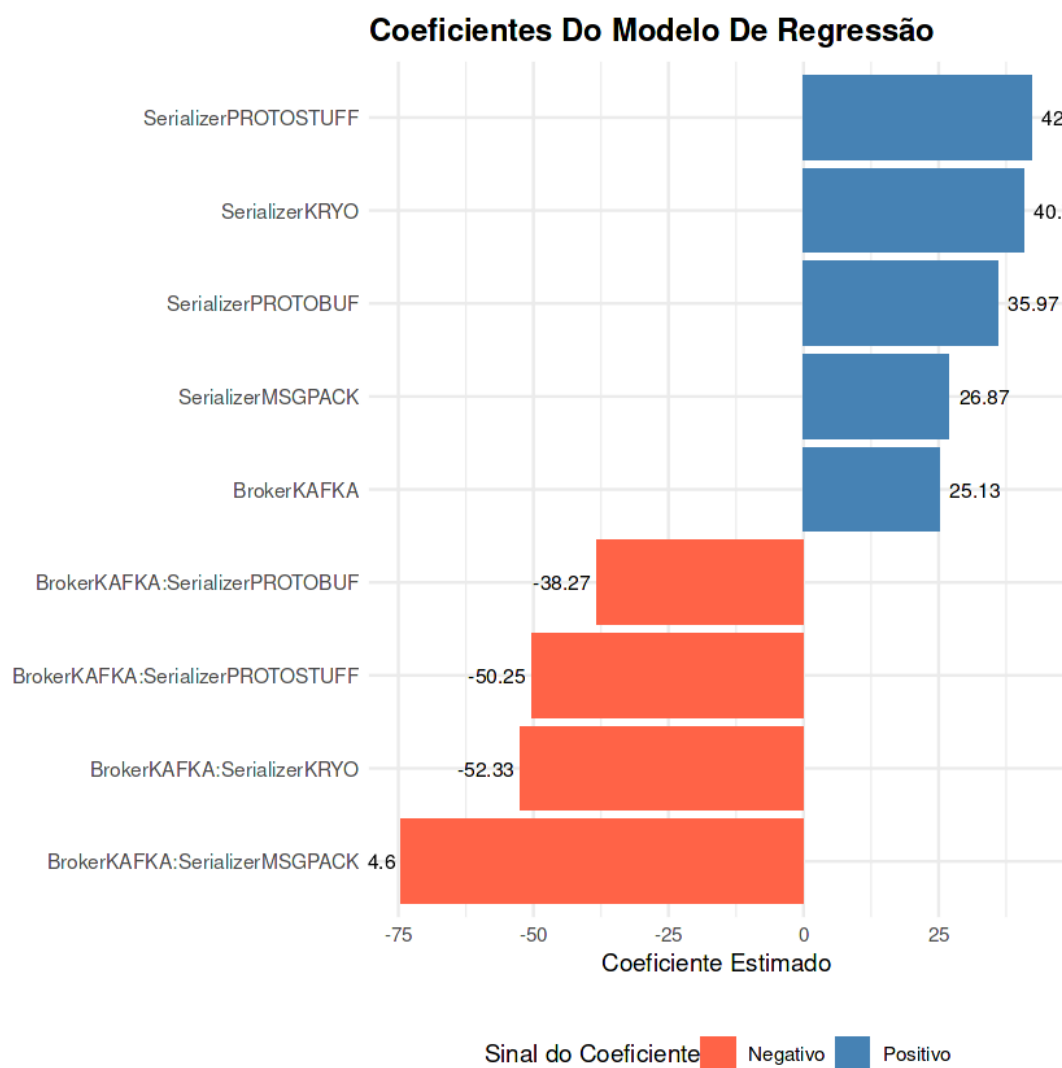


Figura I.21: Gráfico auxiliar do pipeline de análise estatística, utilizado para inspeção intermediária dos resultados.

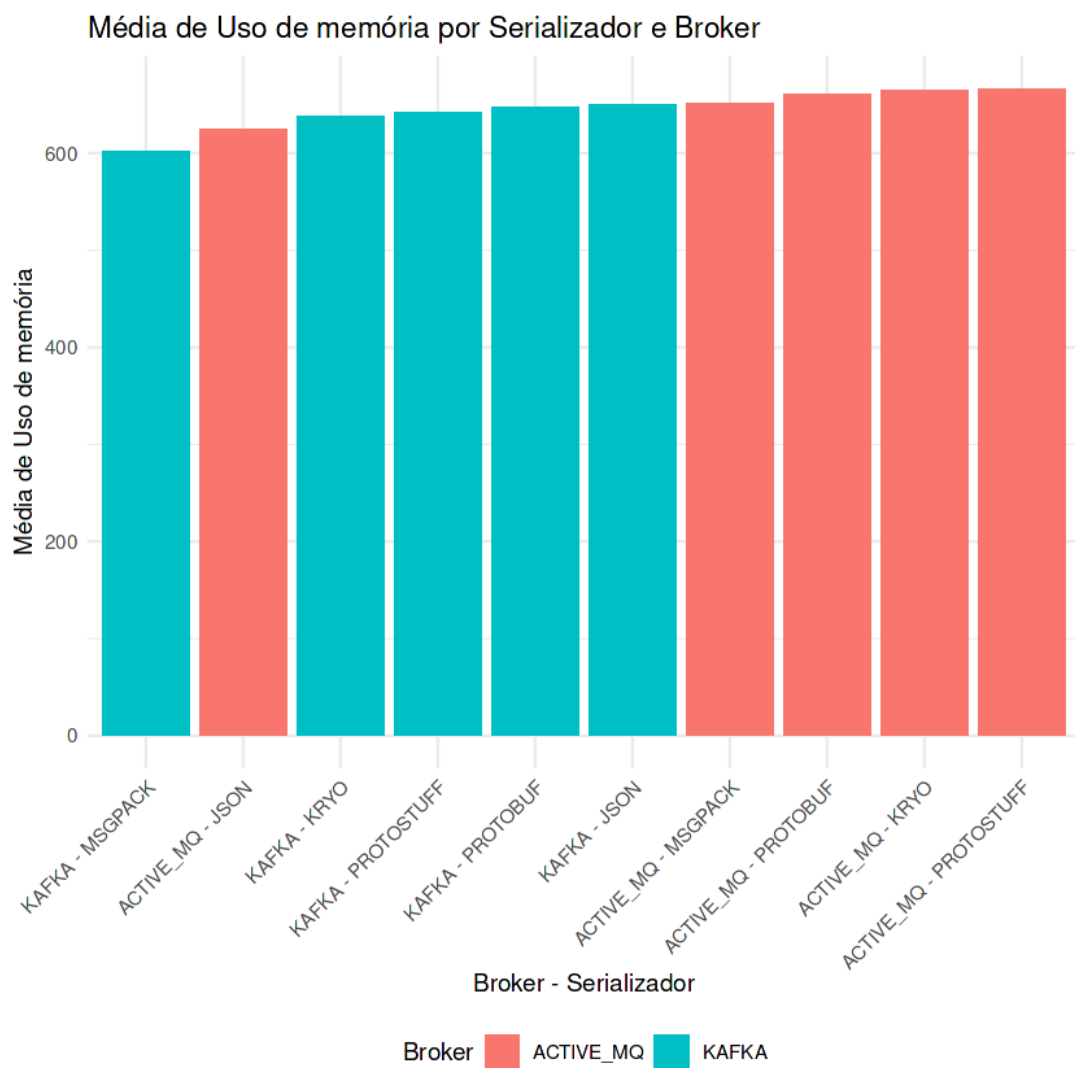


Figura I.22: Gráfico auxiliar do pipeline de análise estatística, utilizado para inspeção intermediária dos resultados.

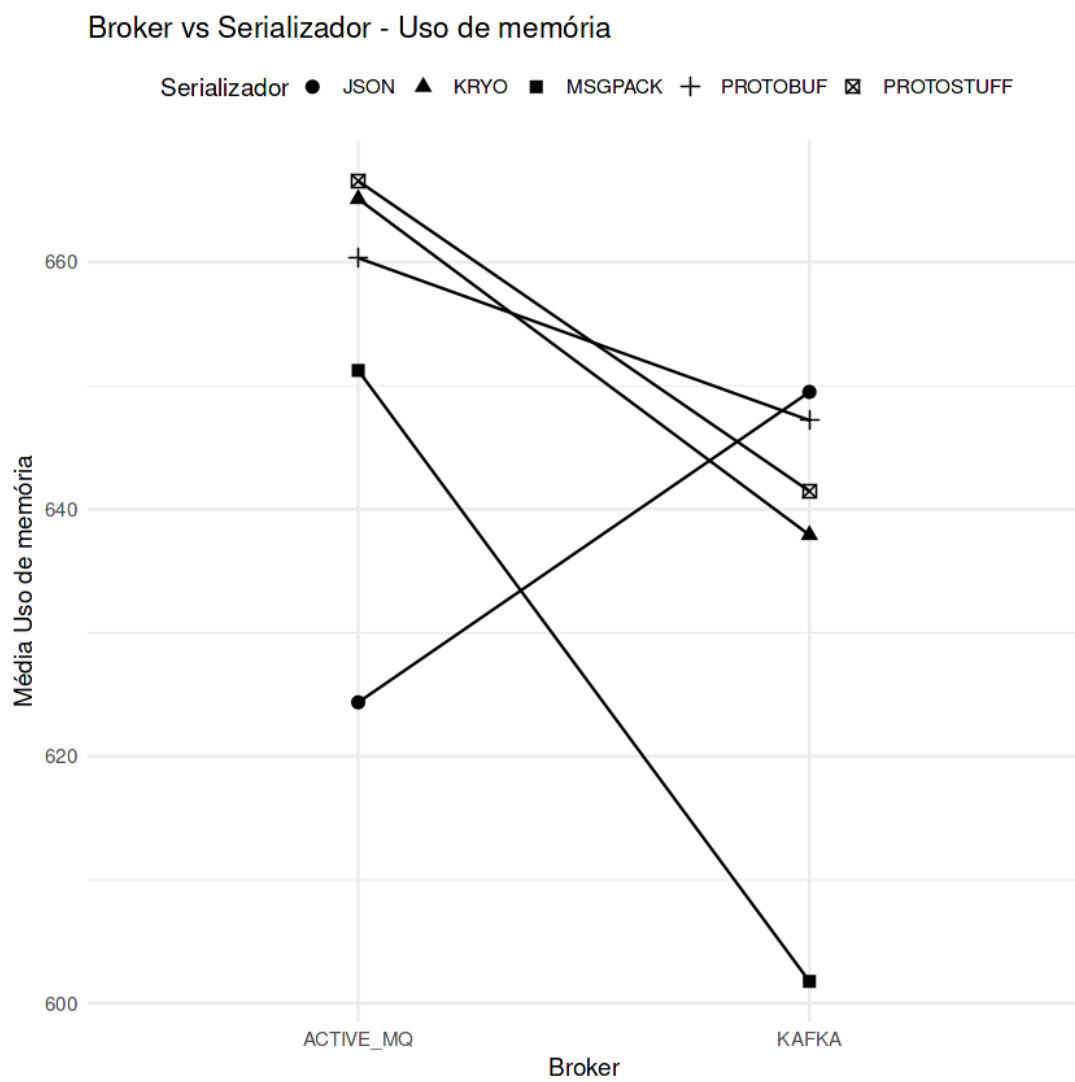


Figura I.23: Gráfico auxiliar do pipeline de análise estatística, utilizado para inspeção intermediária dos resultados.

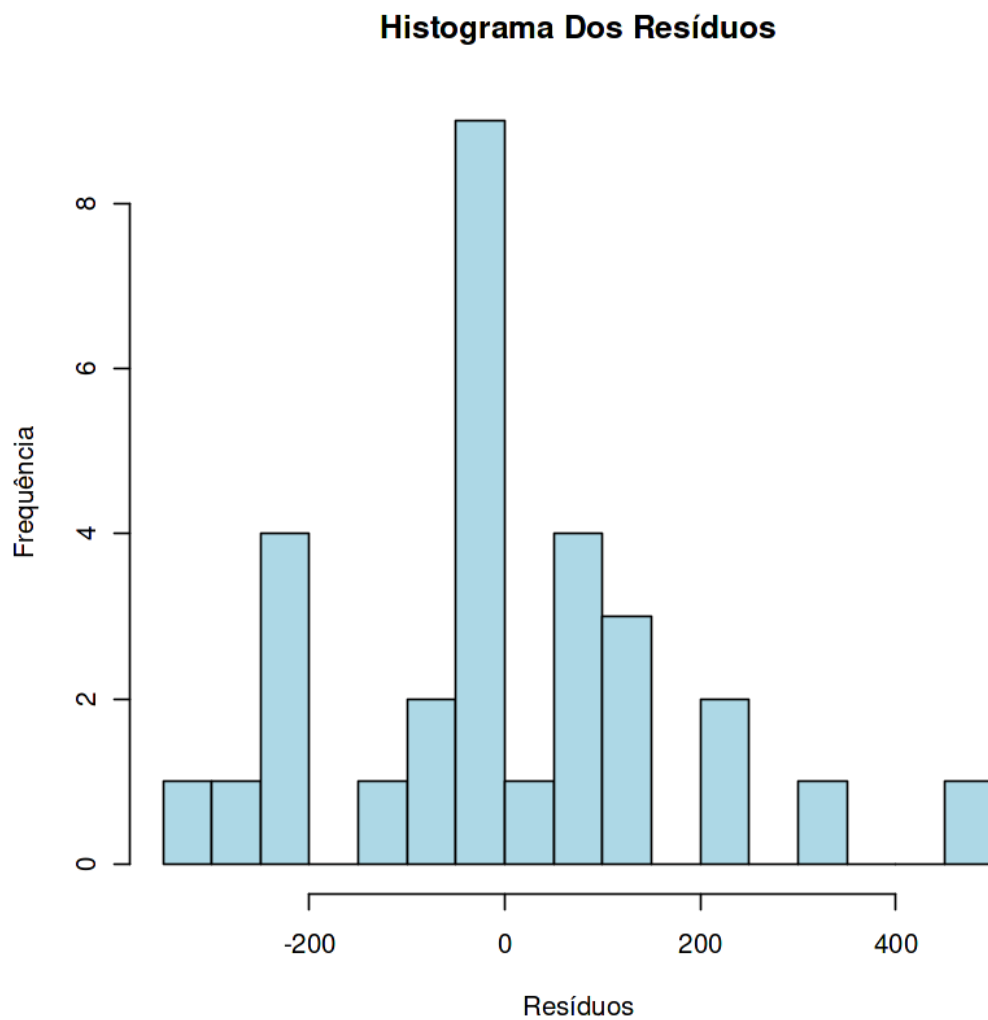


Figura I.24: Gráfico auxiliar do pipeline de análise estatística, utilizado para inspeção intermediária dos resultados.

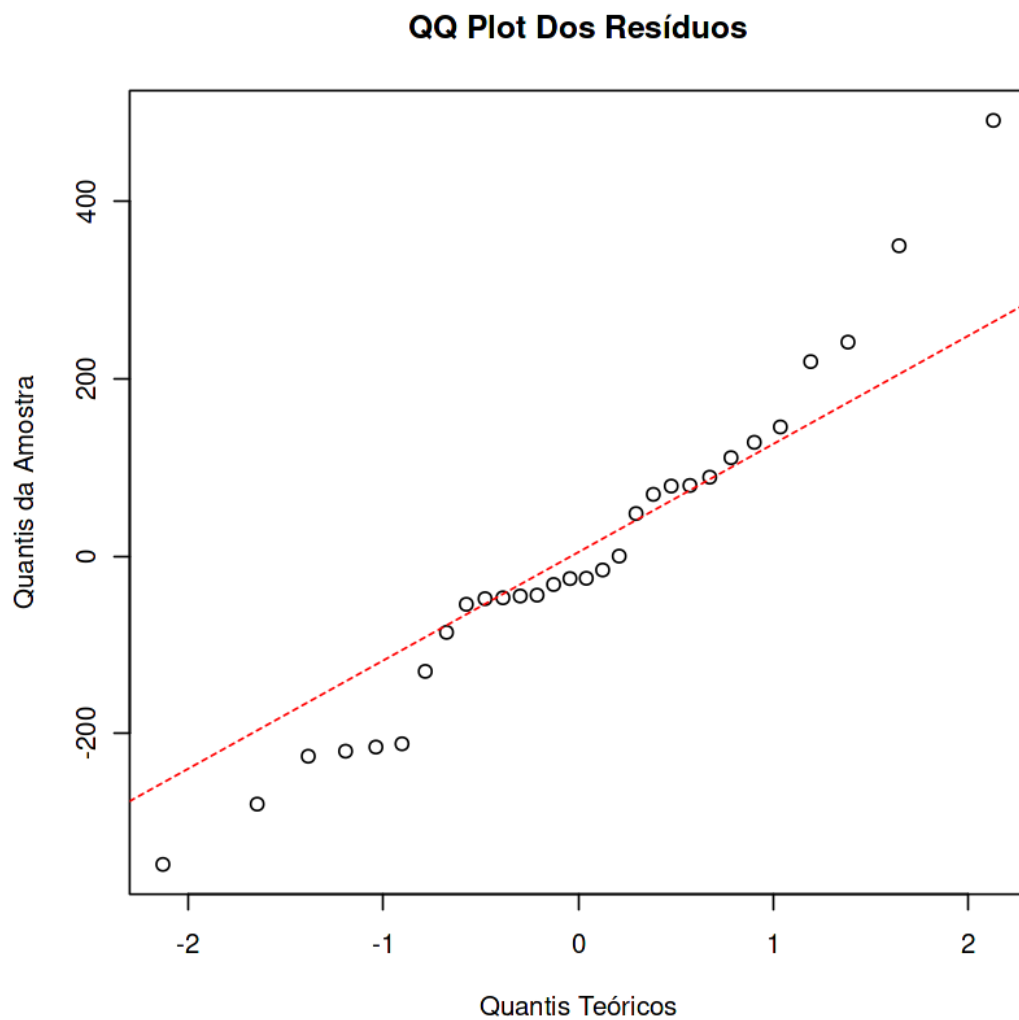


Figura I.25: Gráfico auxiliar do pipeline de análise estatística, utilizado para inspeção intermediária dos resultados.

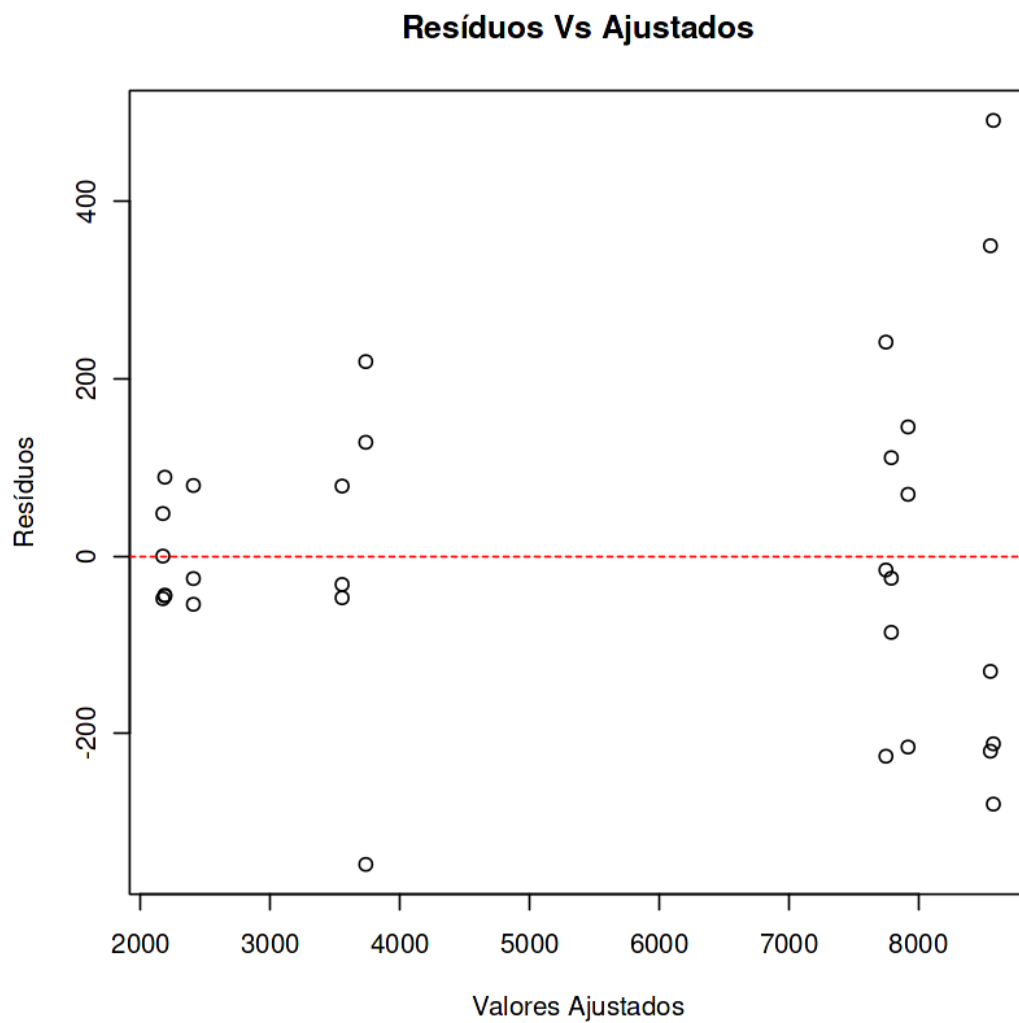


Figura I.26: Gráfico auxiliar do pipeline de análise estatística, utilizado para inspeção intermediária dos resultados.

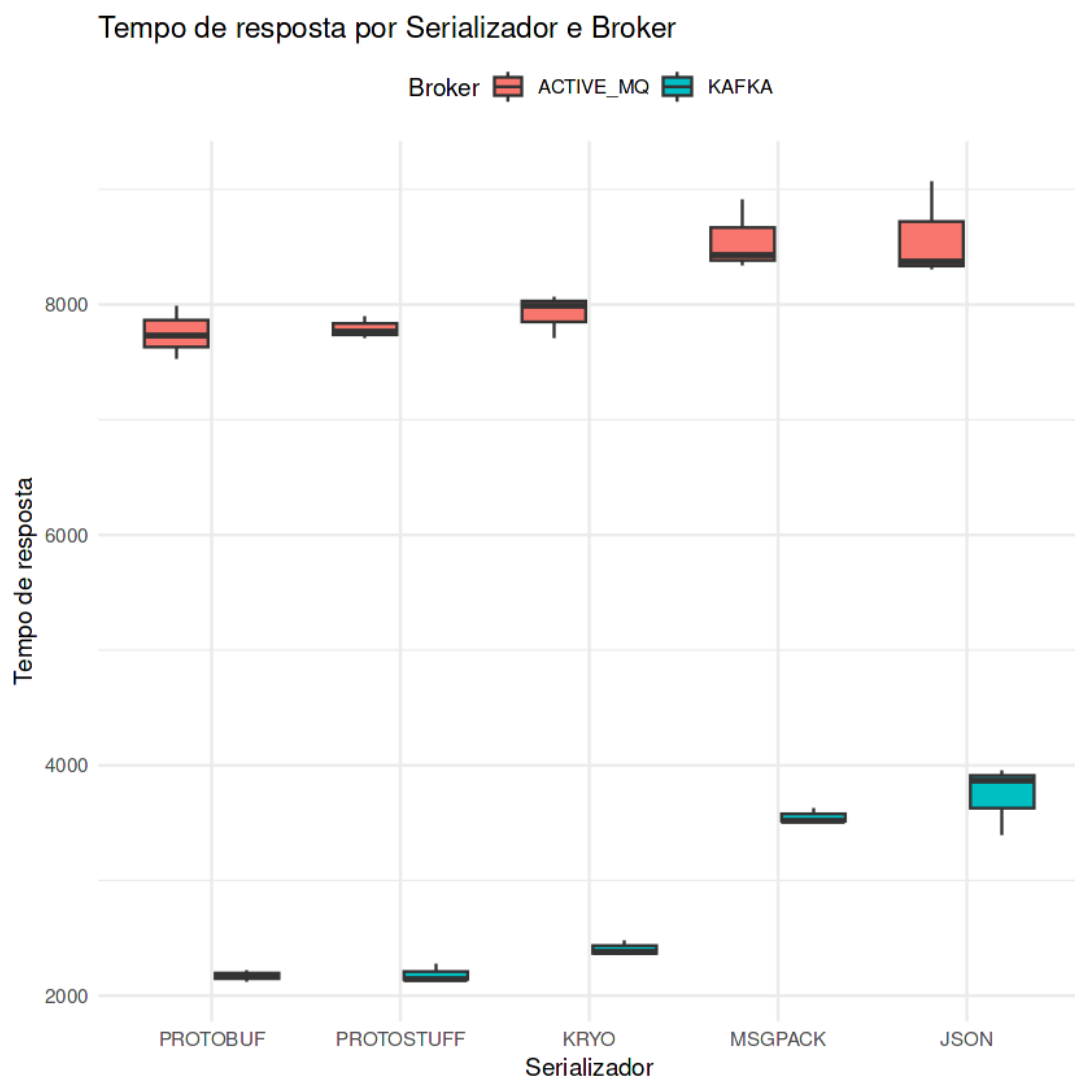


Figura I.27: Gráfico auxiliar do pipeline de análise estatística, utilizado para inspeção intermediária dos resultados.

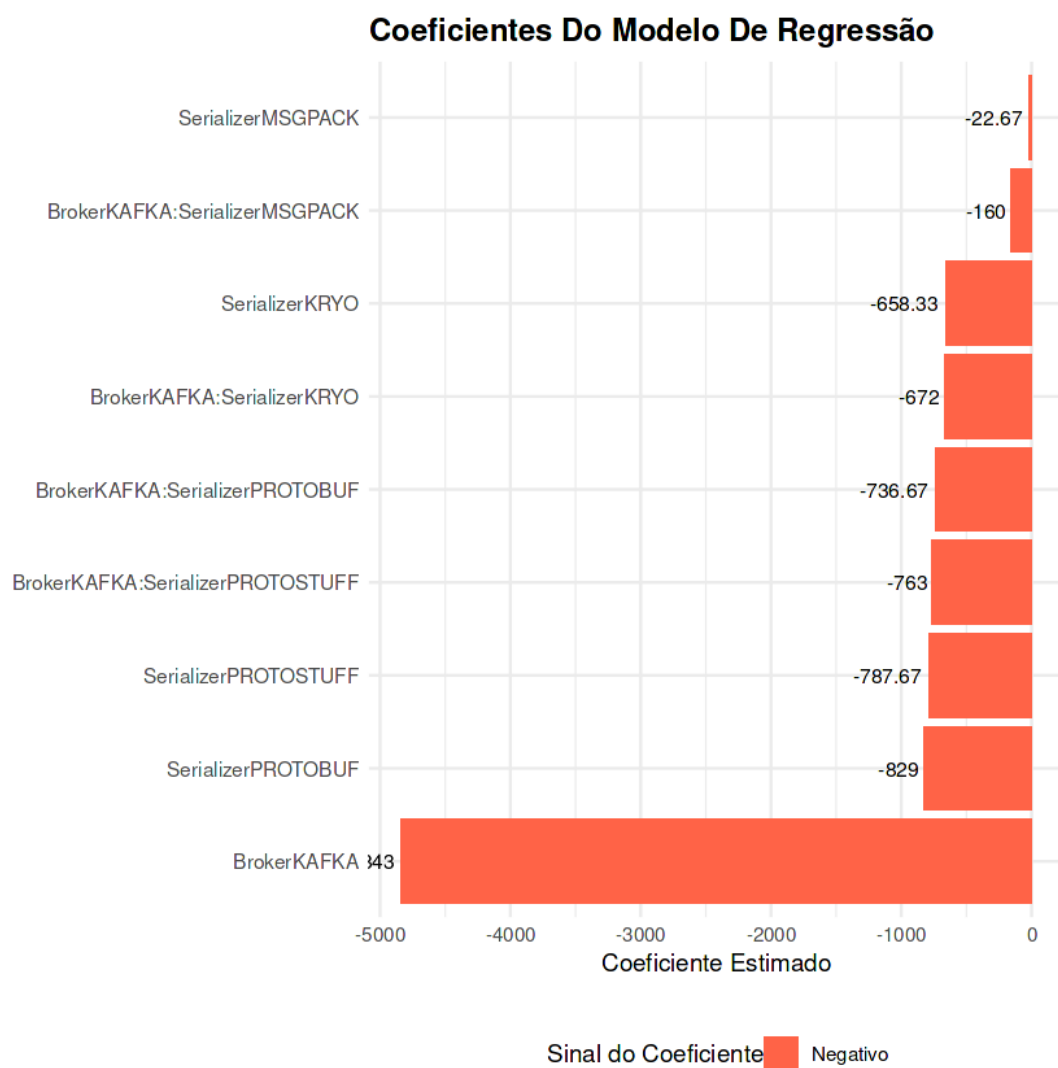


Figura I.28: Gráfico auxiliar do pipeline de análise estatística, utilizado para inspeção intermediária dos resultados.

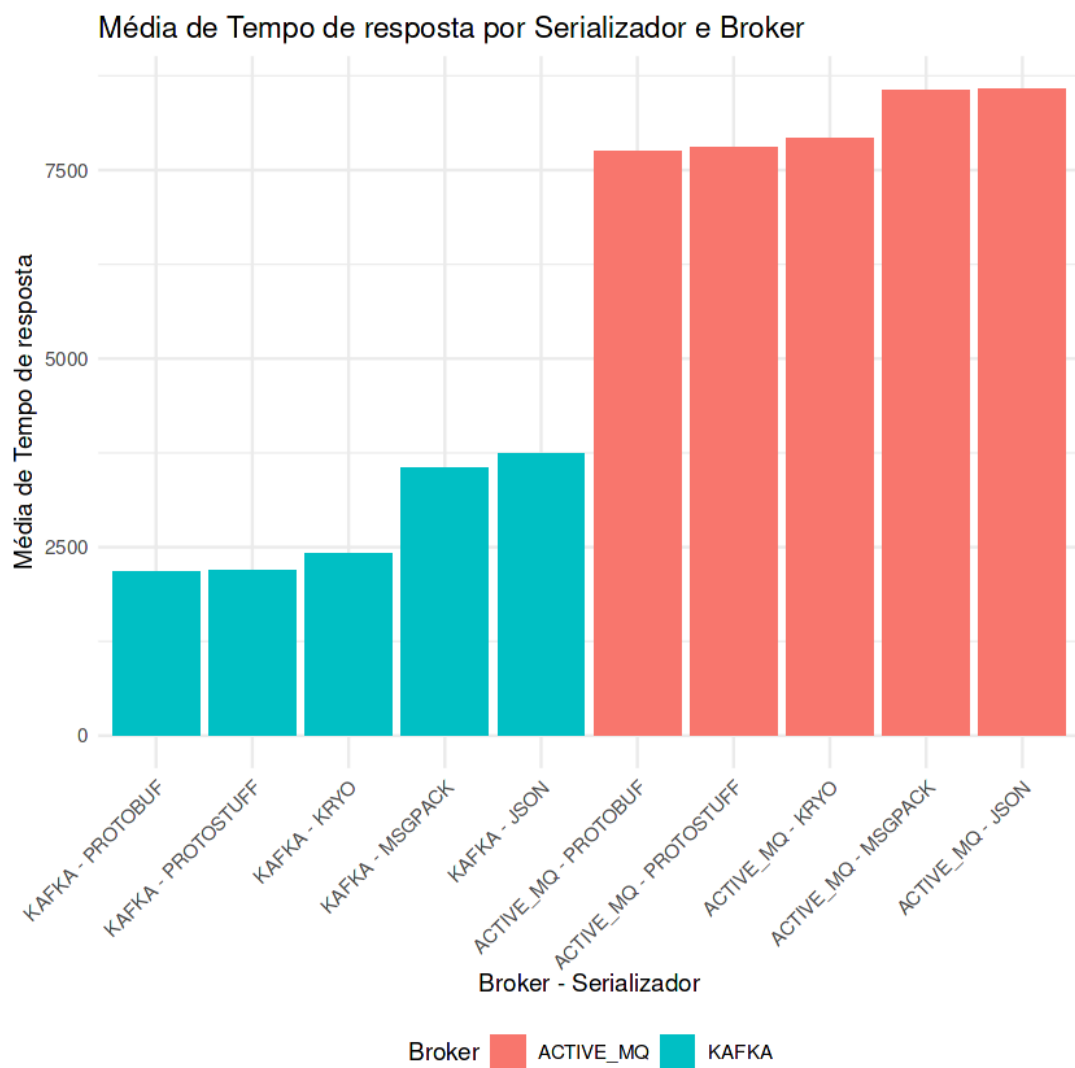


Figura I.29: Gráfico auxiliar do pipeline de análise estatística, utilizado para inspeção intermediária dos resultados.



Figura I.30: Gráfico auxiliar do pipeline de análise estatística, utilizado para inspeção intermediária dos resultados.

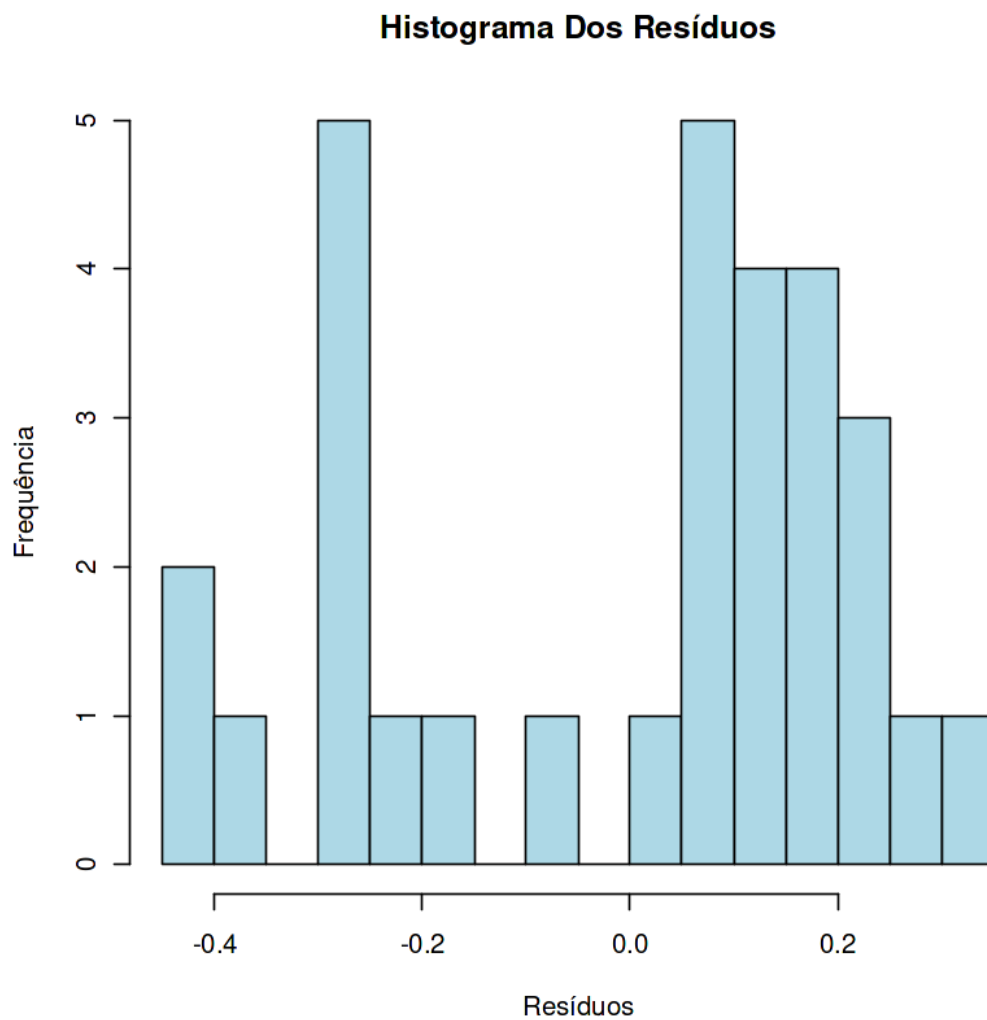


Figura I.31: Gráfico auxiliar do pipeline de análise estatística, utilizado para inspeção intermediária dos resultados.

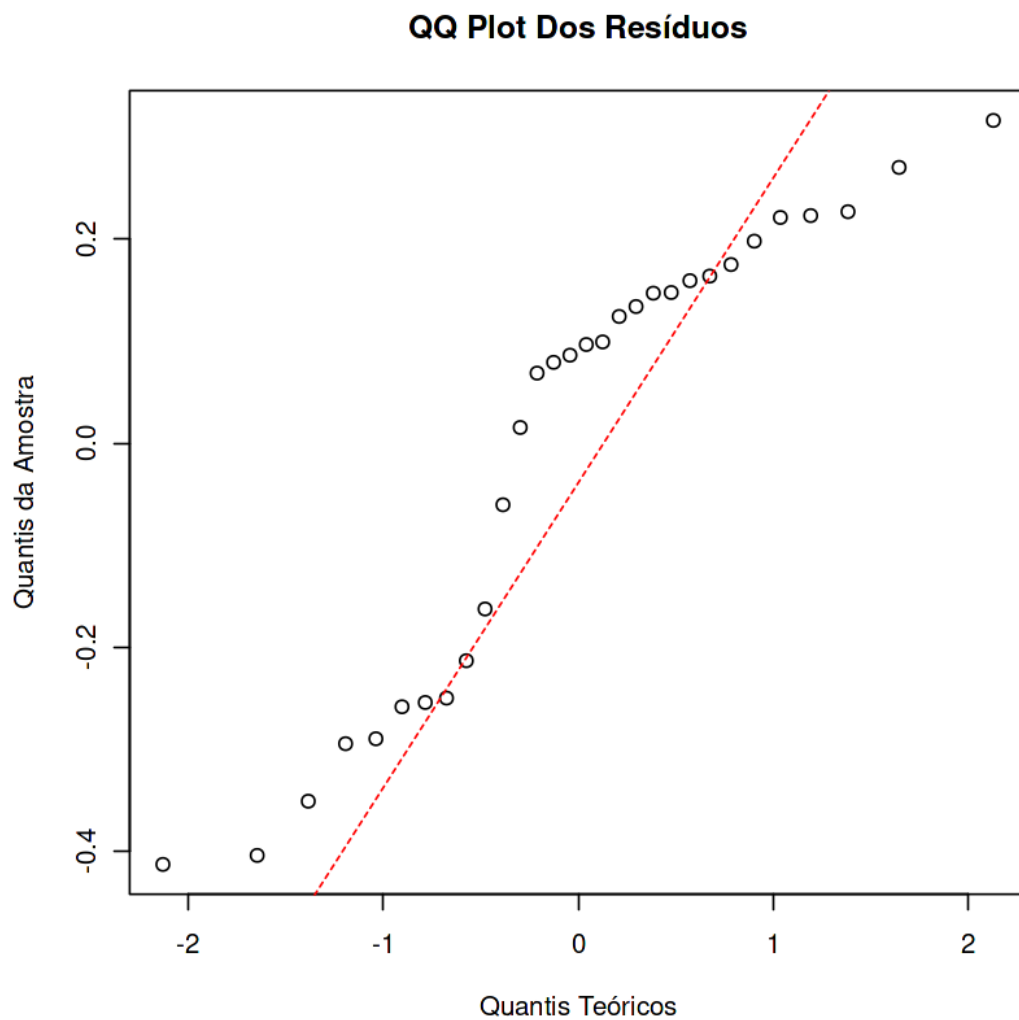


Figura I.32: Gráfico auxiliar do pipeline de análise estatística, utilizado para inspeção intermediária dos resultados.

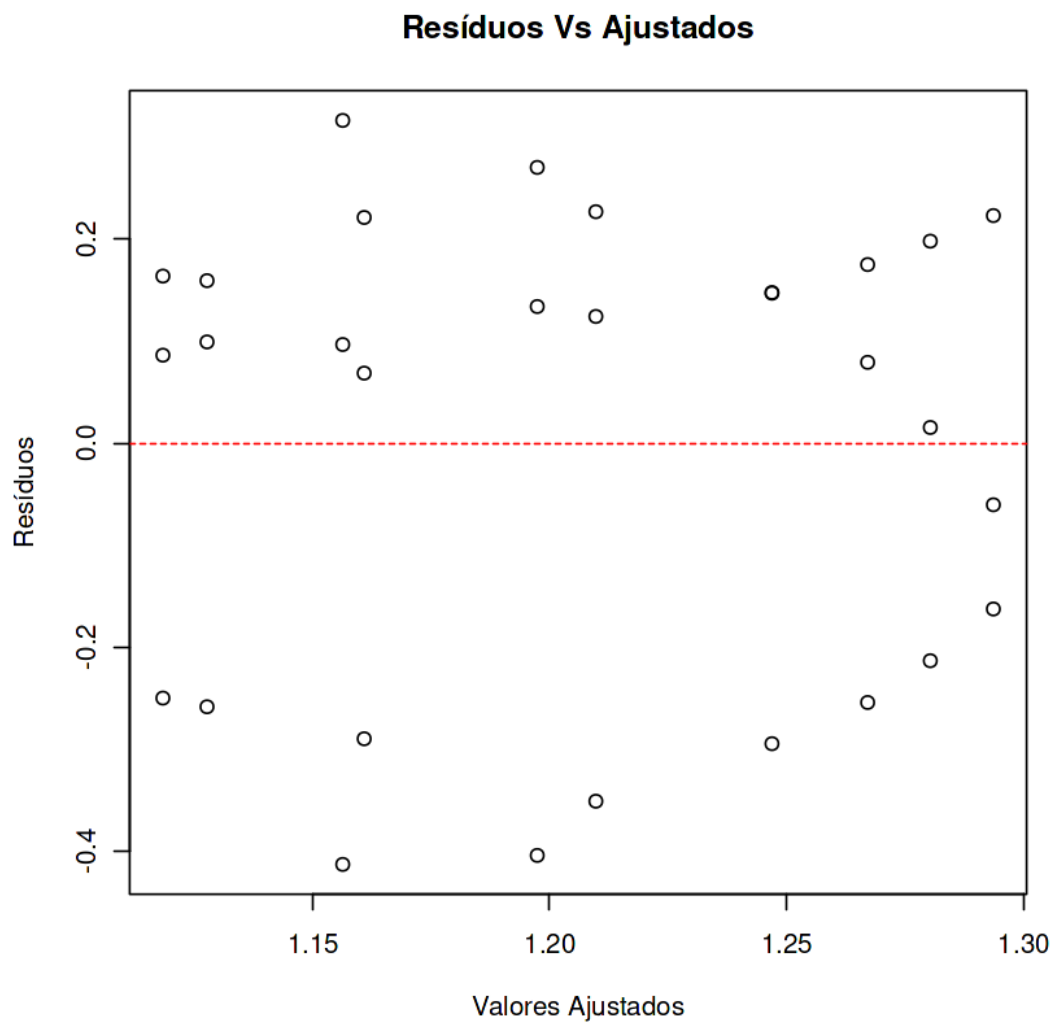


Figura I.33: Gráfico auxiliar do pipeline de análise estatística, utilizado para inspeção intermediária dos resultados.

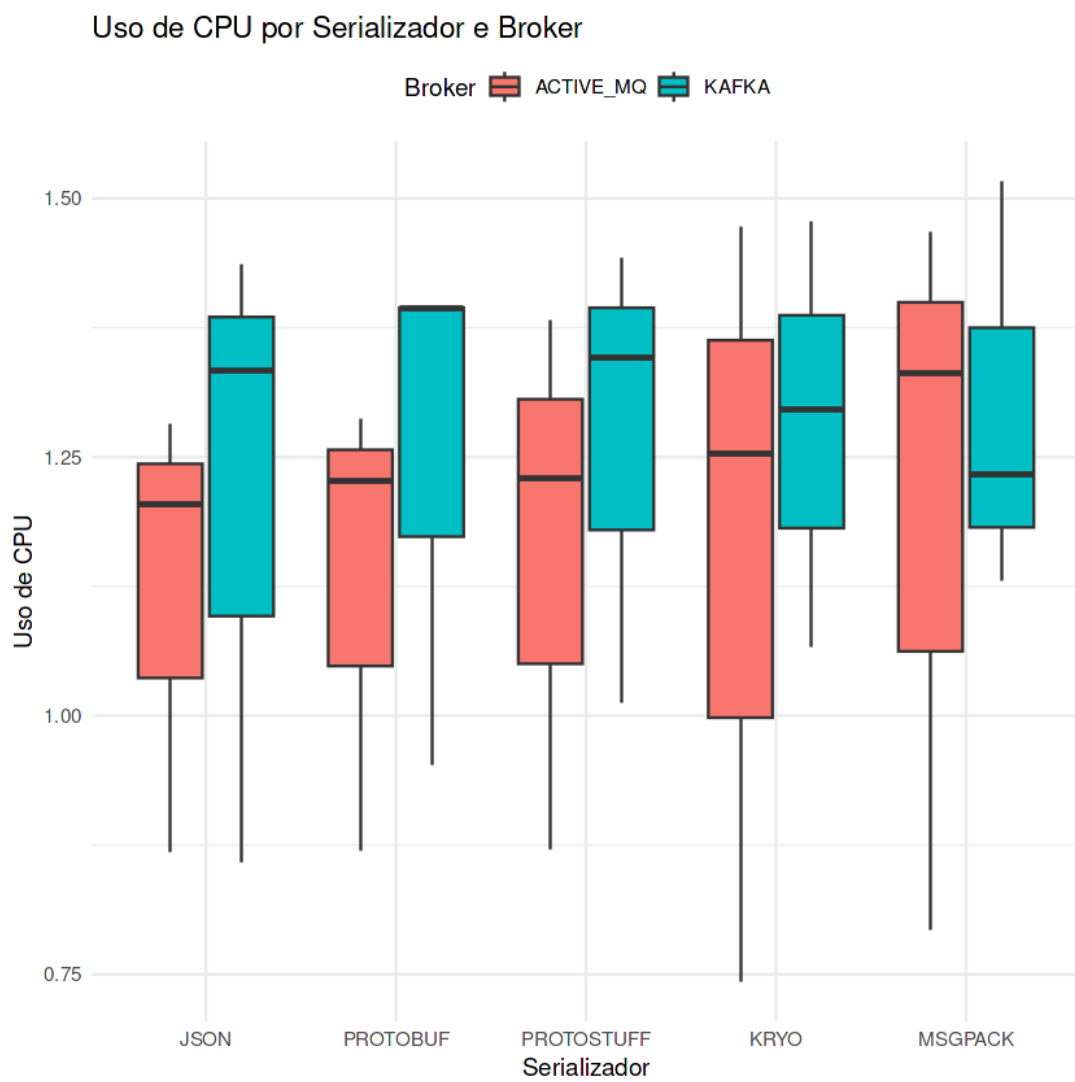


Figura I.34: Gráfico auxiliar do pipeline de análise estatística, utilizado para inspeção intermediária dos resultados.

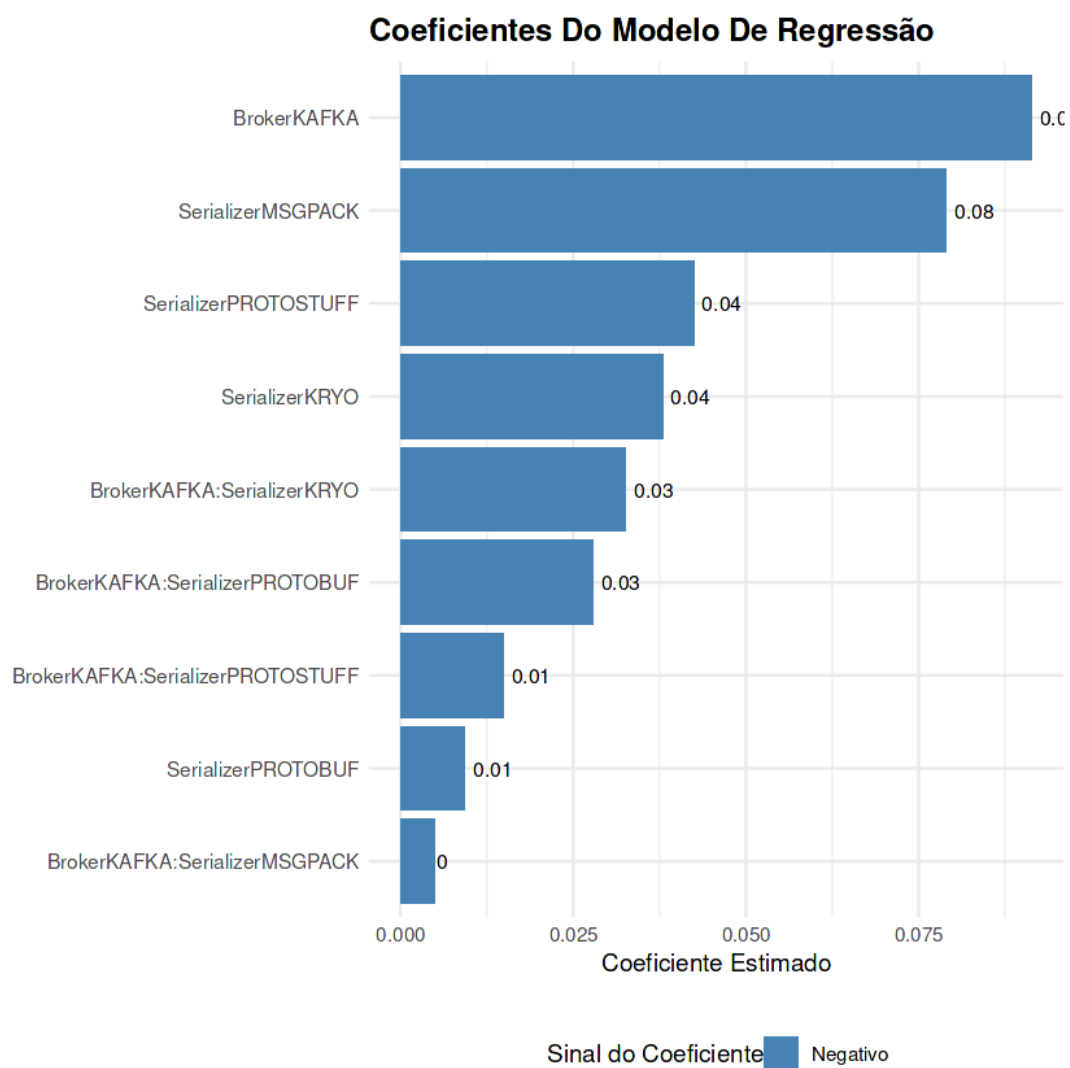


Figura I.35: Gráfico auxiliar do pipeline de análise estatística, utilizado para inspeção intermediária dos resultados.

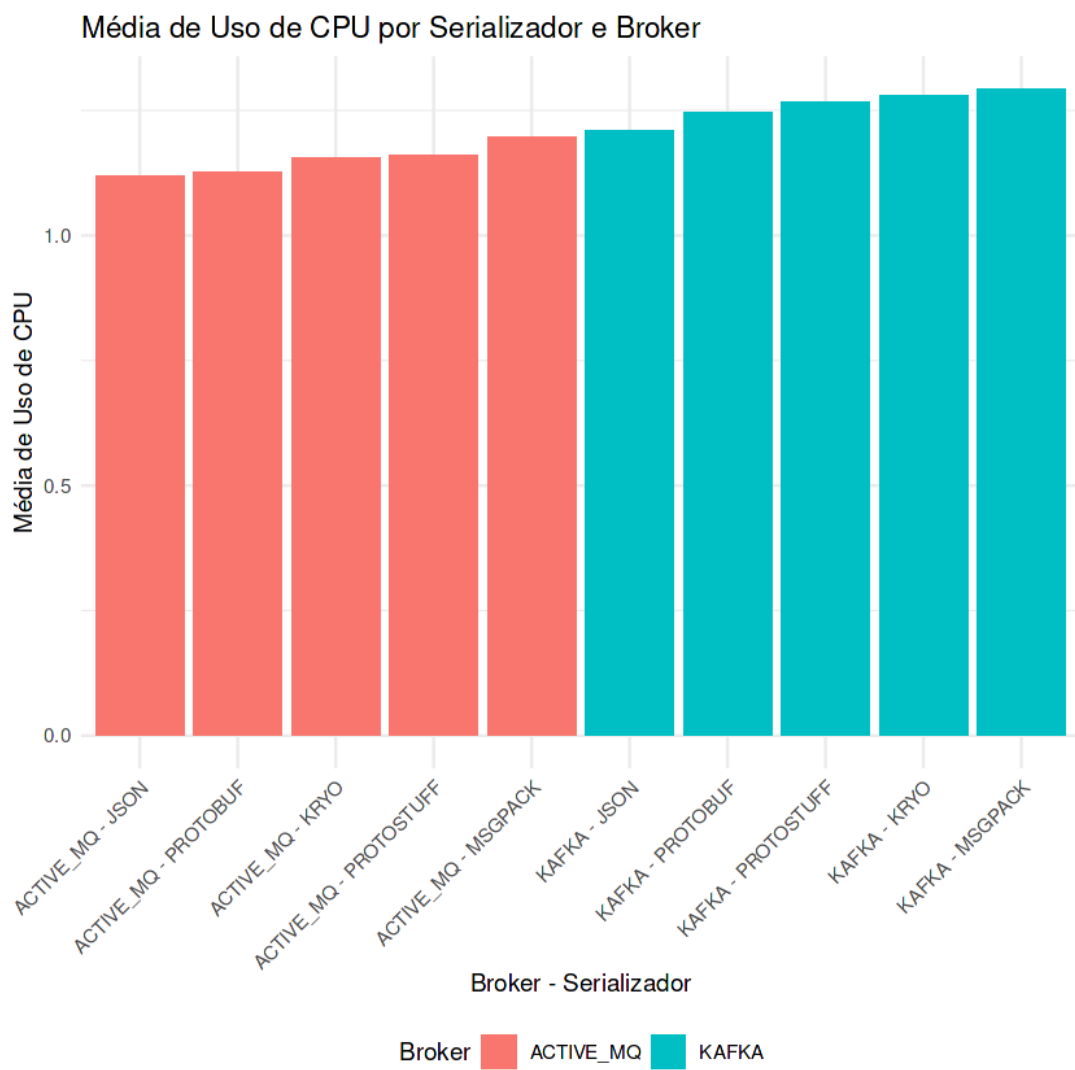


Figura I.36: Gráfico auxiliar do pipeline de análise estatística, utilizado para inspeção intermediária dos resultados.

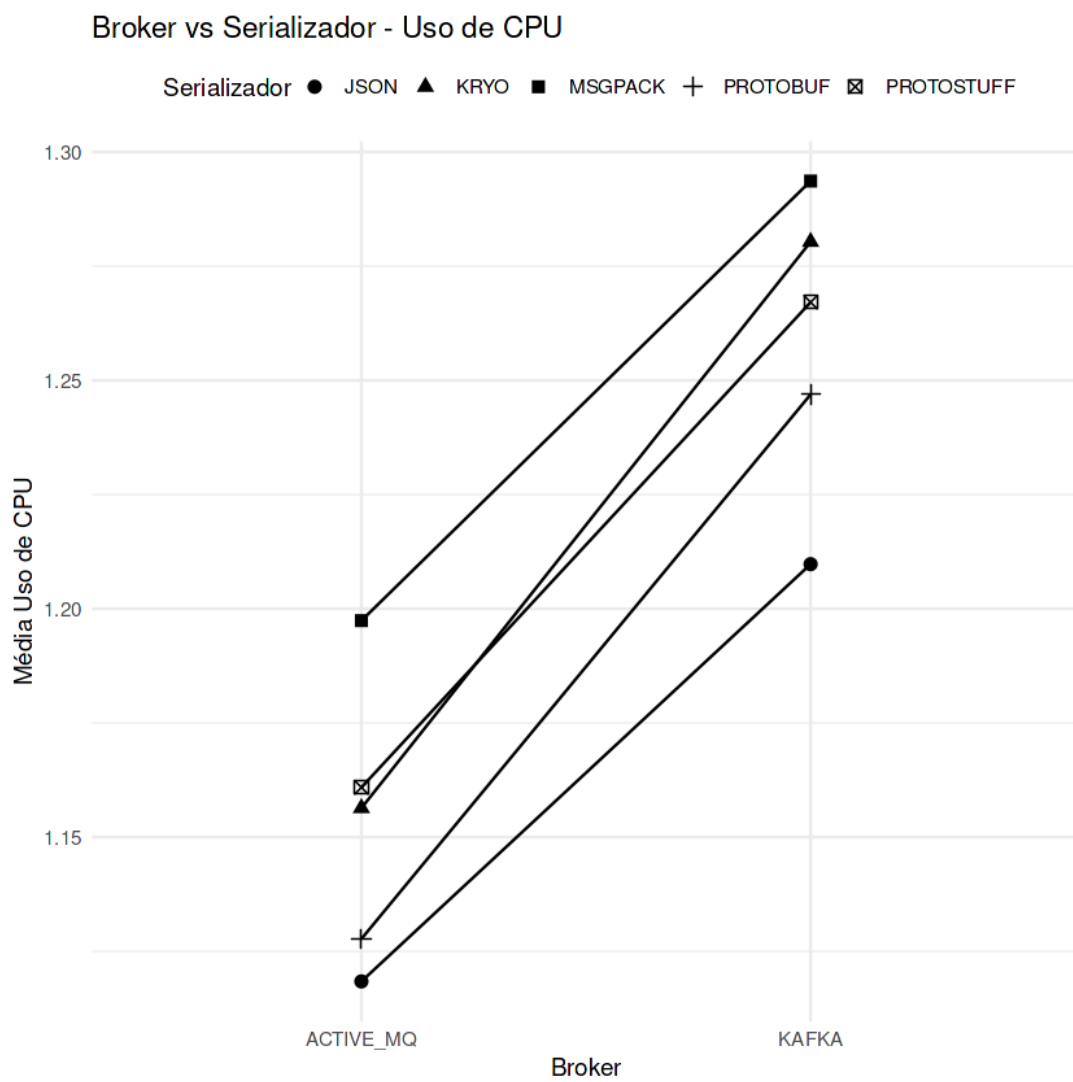


Figura I.37: Gráfico auxiliar do pipeline de análise estatística, utilizado para inspeção intermediária dos resultados.

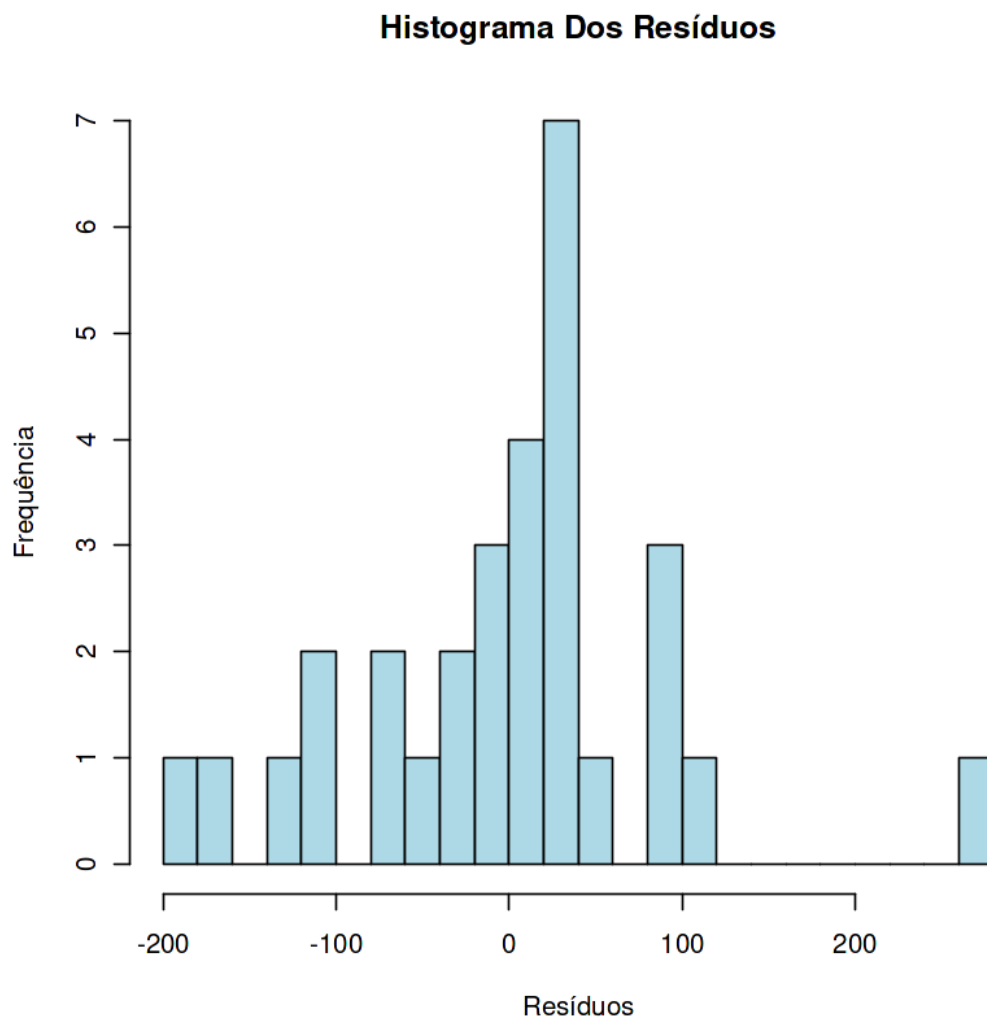


Figura I.38: Gráfico auxiliar do pipeline de análise estatística, utilizado para inspeção intermediária dos resultados.

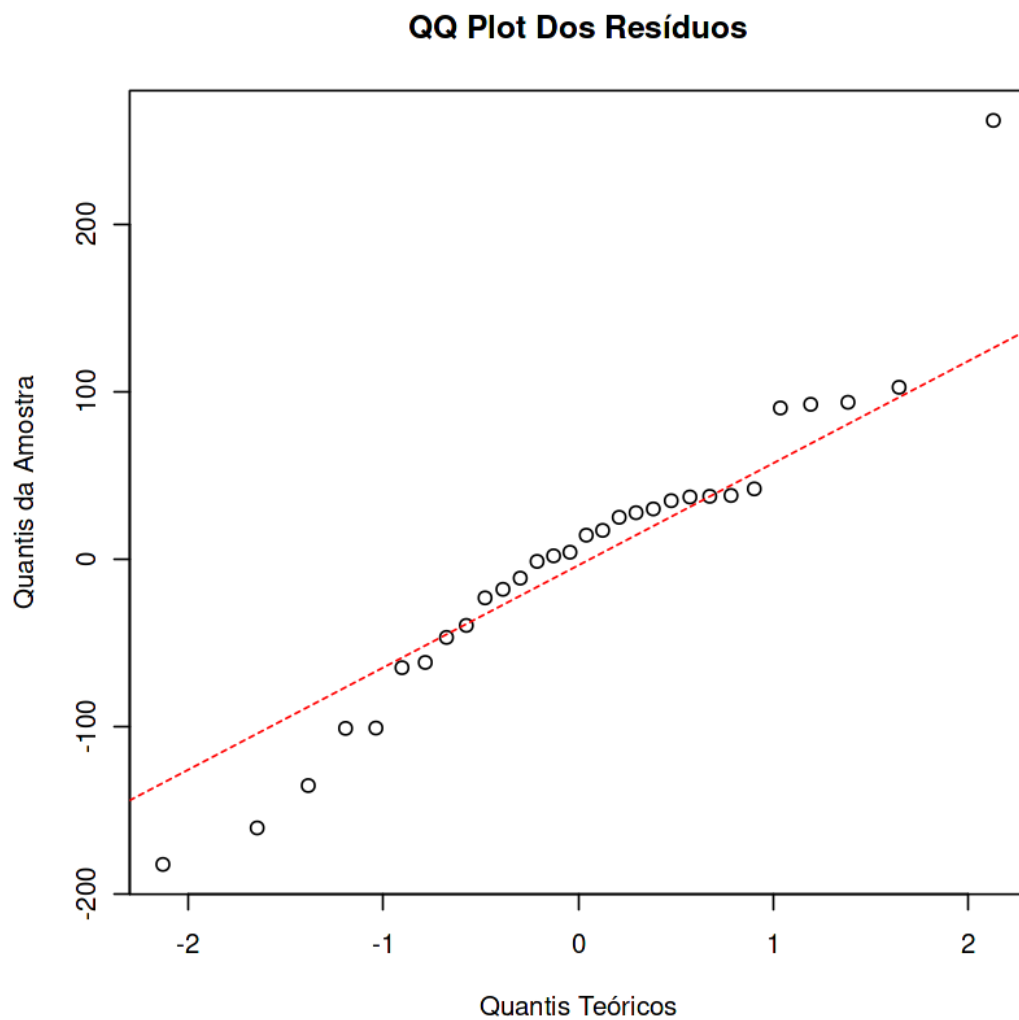


Figura I.39: Gráfico auxiliar do pipeline de análise estatística, utilizado para inspeção intermediária dos resultados.

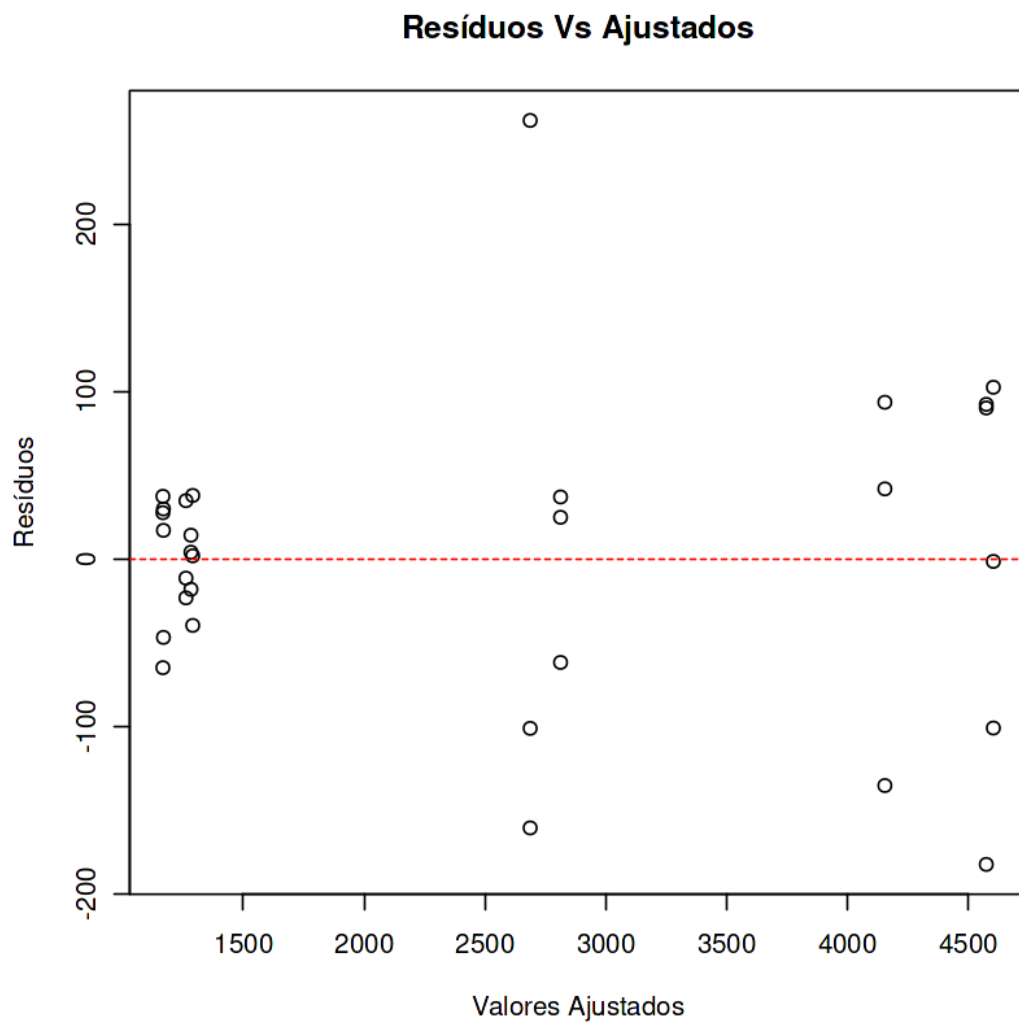


Figura I.40: Gráfico auxiliar do pipeline de análise estatística, utilizado para inspeção intermediária dos resultados.



Figura I.41: Gráfico auxiliar do pipeline de análise estatística, utilizado para inspeção intermediária dos resultados.

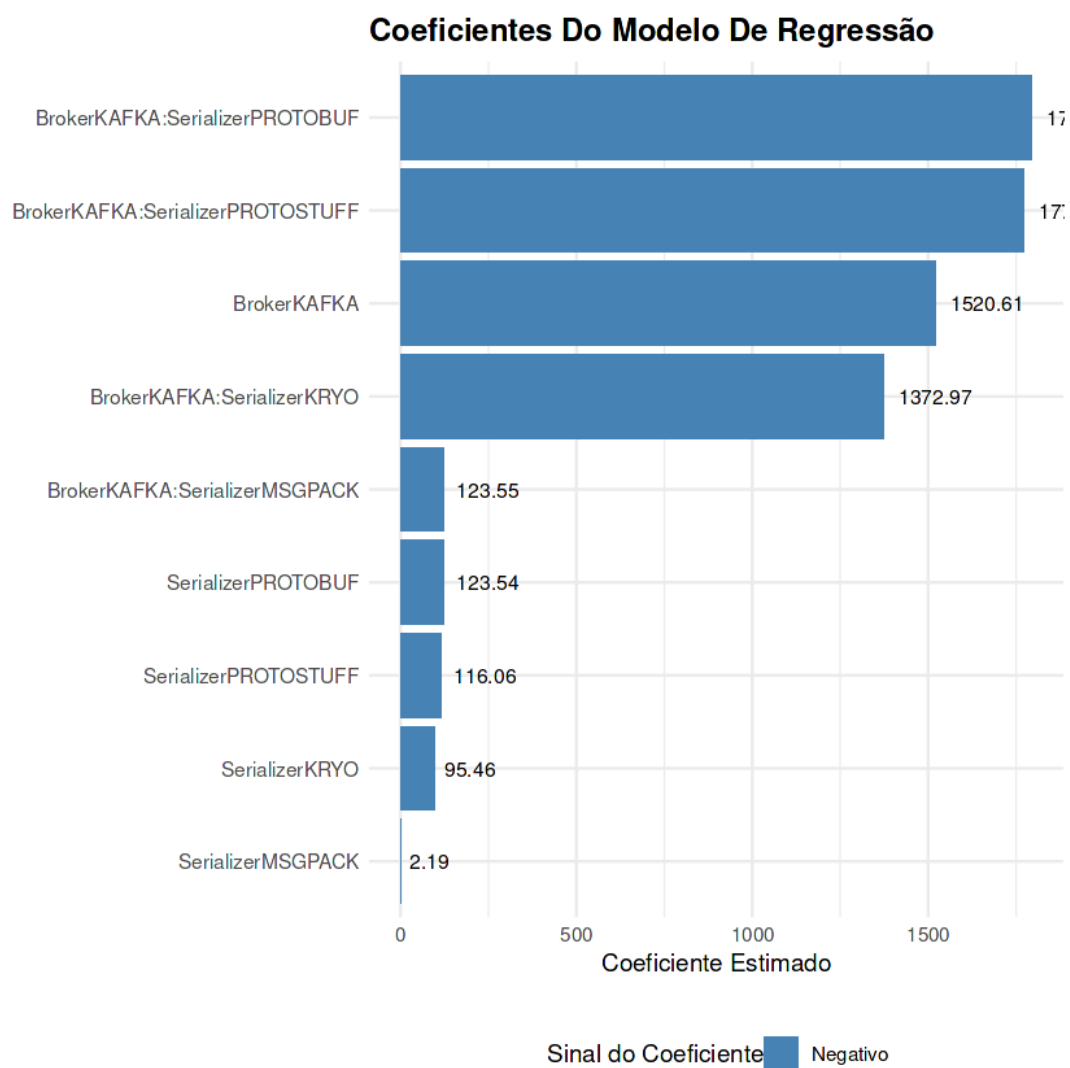


Figura I.42: Gráfico auxiliar do pipeline de análise estatística, utilizado para inspeção intermediária dos resultados.

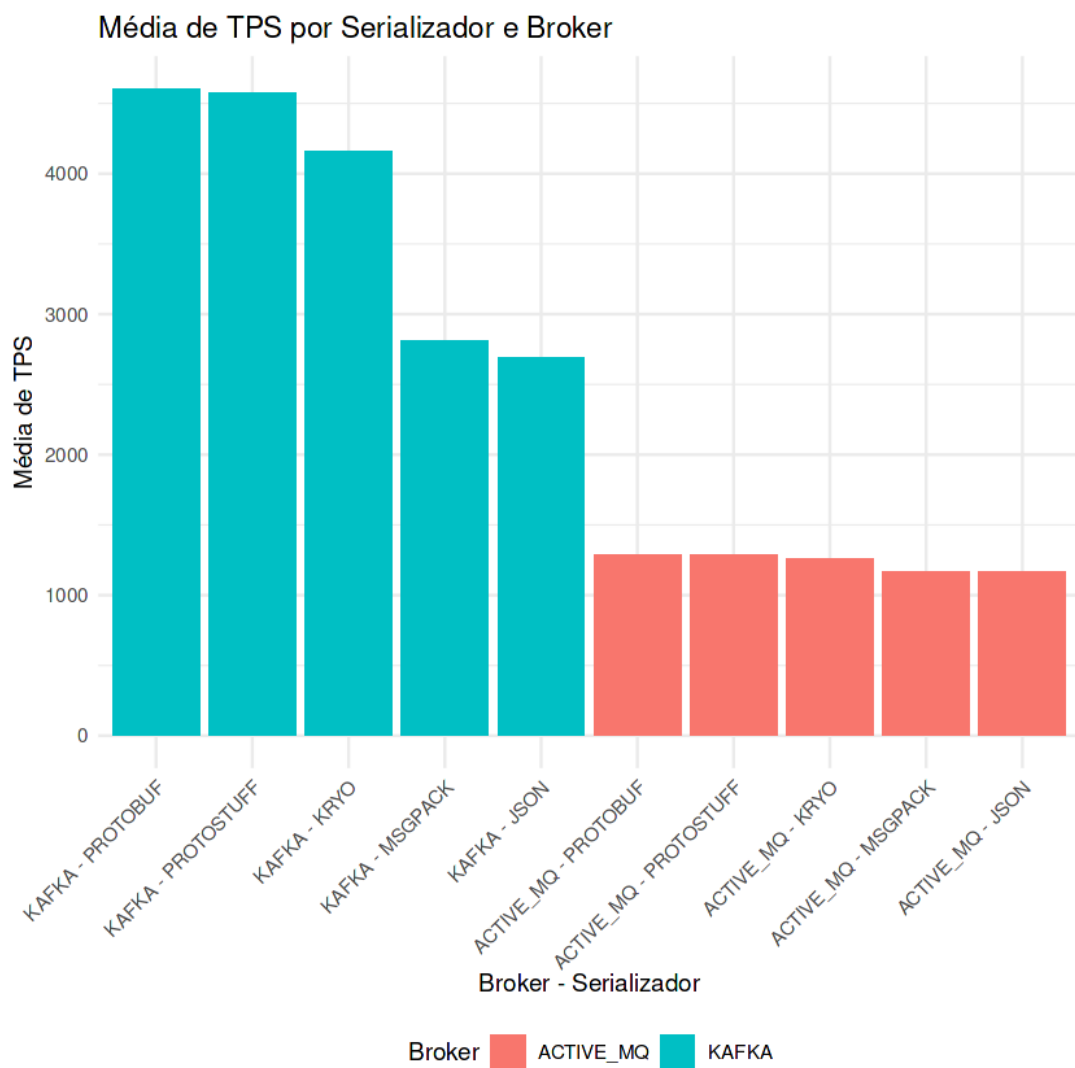


Figura I.43: Gráfico auxiliar do pipeline de análise estatística, utilizado para inspeção intermediária dos resultados.

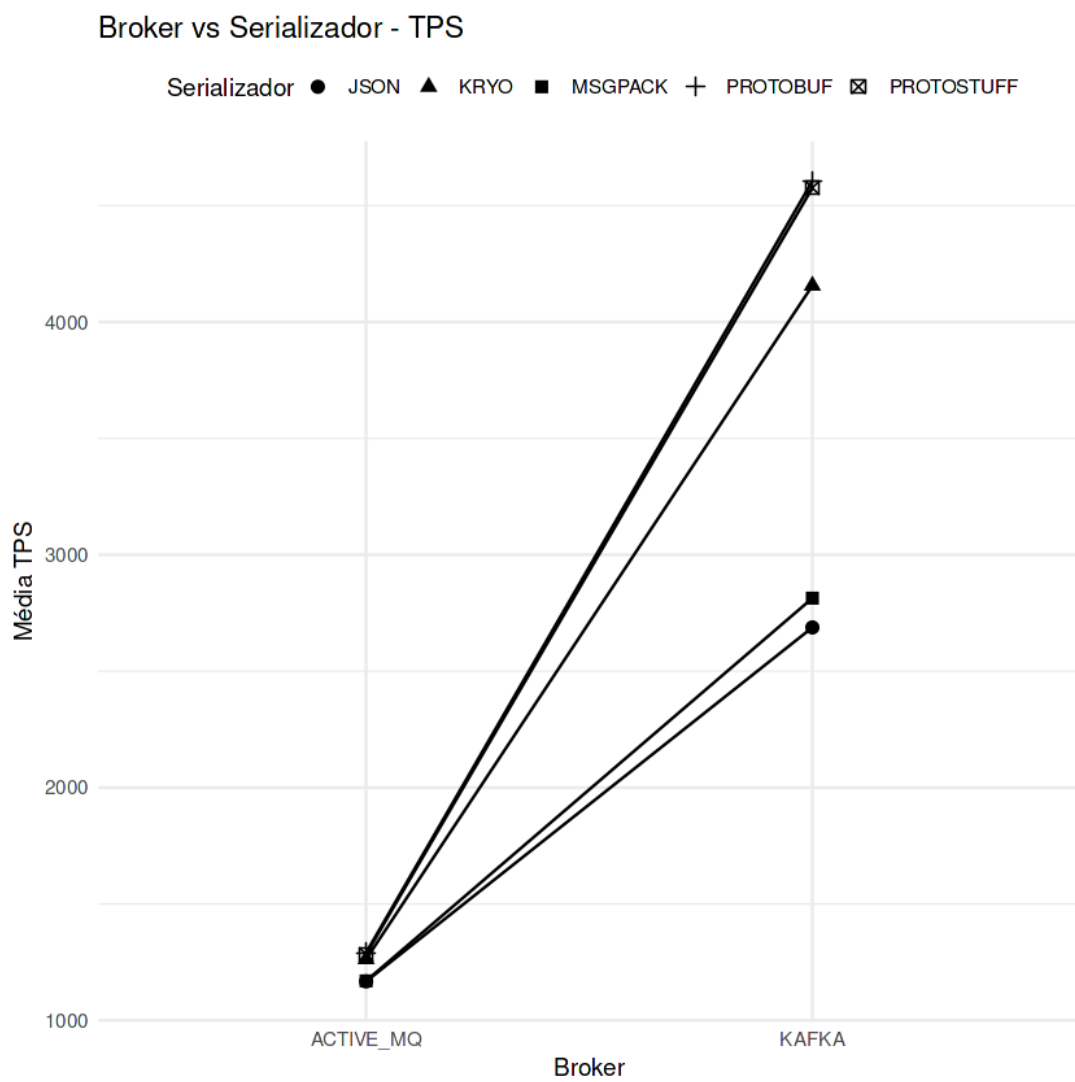


Figura I.44: Gráfico auxiliar do pipeline de análise estatística, utilizado para inspeção intermediária dos resultados.