# Avaliação de Desempenho de Frameworks Web Java entre Reatividade e Threads Virtuais: Um Estudo Comparativo entre Spring e Quarkus

**Ítalo Martins de Deus[1], Rubens de Souza Matos Junior[1], George Leite Júnior[1], Gustavo da Silva Quirino[1]**

[1]Instituto Federal de Educação Ciência e Tecnologia de Sergipe – Campus Lagarto

{italo.deus089,rubens.junior,george.junior,gustavo.quirino}
@academico.ifs.edu.br

***Abstract.*** *Scalability in Java web applications faces challenges in traditional concurrency models. This study compares two solutions: Spring Boot with virtual threads and Quarkus with reactive programming, evaluating their performance in blocking operations. Controlled tests show that each framework has distinct advantages – Spring in performance and Quarkus in stability. The choice between them should consider the specific requirements of the project, balancing efficiency and robustness for modern web applications.*

***Resumo.*** *A escalabilidade em aplicações web Java enfrenta desafios nos modelos tradicionais de concorrência. Este estudo compara duas soluções: Spring Boot com threads virtuais e Quarkus com programação reativa, avaliando seu desempenho em operações bloqueantes. Testes controlados mostram que cada framework tem vantagens distintas – Spring em desempenho e Quarkus em estabilidade. A escolha entre eles deve considerar os requisitos específicos do projeto, balanceando eficiência e robustez para aplicações web modernas.*

## 1. Introdução

A linguagem Java tem sido predominantemente utilizada no desenvolvimento de aplicações web, representando 38% dos softwares construídos com a linguagem em 2023 (Mariasova, 2024). Dentro desse contexto, os principais *frameworks* utilizados são o Spring Boot (incluindo Spring MVC) e o Quarkus. O Spring Boot lidera com 72% de adoção, seguido pelo Spring MVC com 39%, enquanto o Quarkus tem 4% de participação (JetBrains, 2023).

Um dos principais desafios enfrentados por esses *frameworks* é a capacidade de lidar com múltiplas requisições simultâneas. Tradicionalmente, o desempenho dos sistemas web estava limitado pela quantidade de *threads* disponíveis no *hardware*, o que impunha restrições à escalabilidade, já que *threads* são recursos caros em termos de memória e processamento. Cada requisição consumia uma *thread* do início ao fim, gerando desperdício de recursos e limitando o número de requisições que um servidor poderia suportar simultaneamente (Quarkus, [s.d.]; Vertx, [s.d.]).

Como destaca Pressler (2018), há uma incompatibilidade significativa entre a escala de concorrência necessária para aplicações modernas (como milhões de conexões simultâneas) e a capacidade dos *threads* do sistema operacional, que não conseguem lidar eficientemente com mais de alguns milhares. Essa limitação impacta diretamente o

desempenho dos sistemas.

Para superar esse problema, duas abordagens principais têm sido adotadas:

- APIs de programação reativa: Permitem o processamento assíncrono e não bloqueante, otimizando o uso de recursos.
- *Threads* no espaço do usuário: Reduzem a dependência das *threads* do sistema operacional, permitindo maior escalabilidade.

Tanto o Spring Boot quanto o Quarkus têm incorporado essas abordagens em seus ecossistemas para melhorar a escalabilidade das aplicações. O estudo visa comparar o desempenho da programação reativa no Quarkus com o uso de threads no espaço do usuário no Spring Boot, a fim de avaliar qual abordagem e framework oferecem o melhor modelo de concorrência para aplicações web.

## 2. Objetivos

O presente estudo tem como objetivo mensurar a diferença de desempenho entre o Spring Boot, utilizando *threads* virtuais em sua abordagem tradicional (código sequencial e I/O bloqueante), e o Quarkus, com sua API reativa e *threads* nativas. A finalidade é identificar qual *framework*, com sua respectiva abordagem, suporta um maior volume de requisições, mantendo respostas satisfatórias aos usuários.

### 2.1. Objetivos específicos

Os objetivos específicos buscam responder às seguintes perguntas:

- **Pergunta A:** Qual *framework*, com sua abordagem específica, consegue processar o maior número de requisições concorrentes antes de atingir o ponto de falha?
- **Pergunta B:** Qual *framework* ou abordagem oferece melhores tempos de resposta ao usuário à medida que o volume de requisições aumenta?
- **Pergunta C:** Qual *framework* ou abordagem consegue processar o maior número de requisições em um mesmo intervalo de tempo?
- **Pergunta D:** Qual abordagem é mais vantajosa ao considerar a complexidade de desenvolvimento em relação aos resultados obtidos?

## 3. Justificativa

A escolha pelo Spring Boot com *threads* virtuais justifica-se por sua ampla adoção pelos desenvolvedores, sendo sua abordagem tradicional mais utilizada. Já o Quarkus, com sua API reativa, destaca-se como uma das implementações mais populares, embora também suporte programação imperativa e *threads* virtuais.

Estudos mostram que a API reativa do Spring WebFlux não apresenta grandes diferenças de desempenho em relação à abordagem tradicional com *threads* nativas (Nordlund e Nordström, 2022). Em contraste, o Quarkus, com sua API reativa, demonstrou resultados superiores tanto ao WebFlux quanto à sua própria API não reativa. Um artigo de Choubey (2024) reforça essa vantagem, mostrando que o Quarkus foi 68% mais rápido que o Spring WebFlux em requisições por segundo.

Ambas as abordagens — *threads* virtuais e APIs reativas — buscam resolver o problema de escalabilidade no modelo thread-per-request, que limita o número de

requisições concorrentes devido ao bloqueio de *threads* em operações de I/O (Pressler, 2018). As APIs reativas, como a do Quarkus, compartilham *threads* nativas e utilizam chamadas não-bloqueantes, mas exigem que o desenvolvedor gerencie a lógica assíncrona. Já as *threads* virtuais mantêm o modelo *thread-per-request*, mas com *threads* em espaço de usuário, permitindo maior escalabilidade e simplificando o desenvolvimento, pois a JVM gerencia a sincronização.

O estudo visa comparar o desempenho dessas abordagens, destacando as vantagens das *threads* virtuais no Spring Boot em relação à programação reativa no Quarkus.

## 4. Fundamentação Teórica

### 4.1. Programação Reativa

O Manifesto Reativo define sistemas reativos como responsivos, elásticos, resilientes e baseados em mensagens assíncronas (Bonér et al., 2014). A programação reativa foca em reagir a mudanças de eventos e dados, evitando bloqueios de *threads* em operações de I/O. APIs reativas, como Spring WebFlux e Vert.x, utilizam chamadas não bloqueantes, permitindo que threads sejam reutilizadas enquanto operações de I/O estão pendentes (Quarkus, [s.d.]; VertX, [s.d.]).

Essa abordagem permite que poucas *threads* gerenciem um grande volume de operações simultâneas. No entanto, a programação reativa é mais complexa, resultando em código difícil de entender, manter e depurar, com pilhas de execução menos úteis e desafios no uso de profilers (Pressler, 2018).

### 4.2. Vert.X: O Núcleo Reativo do Quarkus

O Vert.x, desenvolvido pela Eclipse Foundation, é um ecossistema reativo para sistemas web na JVM, oferecendo módulos reativos como *drivers* de banco de dados (VertX, [s.d.]).

Diferente do modelo *thread-per-request*, o Vert.x utiliza um *loop* de eventos que evita o bloqueio de *threads* do *kernel*, implementando operações de I/O assíncronas. Quando uma operação de I/O é iniciada, a *thread* é liberada para outras tarefas, e o processamento é retomado por outra *thread* disponível após a conclusão (VertX, [s.d.]).

Para trabalhar eficientemente com o *loop* de eventos do Vert.x, é essencial:

- Evitar operações de I/O bloqueantes.
- Minimizar processamentos demorados dentro do *loop* (VertX, [s.d.]).

### 4.3. Threads no espaço do usuário

Diferente das *threads* do sistema operacional, gerenciadas pelo núcleo, as *threads* de usuário são gerenciadas por processos. Isso exige a implementação interna de estruturas como uma tabela de *threads* e um sistema supervisor, responsável pela execução e preempção de recursos (Tanenbaum; Bos, 2016).

No modelo convencional, o núcleo do sistema operacional gerencia a tabela de *threads*, trocas de contexto, contador de programa e ponteiro de pilha (Tanenbaum; Bos, 2016). Já as *threads* de usuário têm custo significativamente menor, com chaveamento e criação/destruição mais eficientes. Além disso, podem ser criadas em grande escala (milhões), sem as limitações de hardware (Pressler, 2018).

No entanto, esse modelo tem desvantagens. O núcleo do sistema não reconhece as *threads* de usuário, o que pode causar bloqueios em chamadas de sistema, paralisando todas as *threads* de usuário associadas a uma *thread* do sistema. Para evitar isso, as implementações precisam usar chamadas de I/O não bloqueantes ou verificações no núcleo para prever bloqueios (Tanenbaum; Bos, 2016).

### 4.4. Threads Virtuais

O Project Loom introduziu as *threads* virtuais no Java, uma alternativa mais eficiente às *threads* tradicionais, mapeadas diretamente para *threads* do sistema operacional (Pressler, 2018). Lançadas no JDK 19 (prévia) e JDK 21 (definitivo), elas mantêm o modelo *thread-per-request* com menor custo de hardware, sem depender de *threads* do *kernel*, e preservam a API familiar e ferramentas de depuração (Bateman; Pressler, 2024).

Gerenciadas pela JVM, as *threads* virtuais operam sobre *threads* do kernel em um esquema N:M. Em operações de I/O bloqueante, a *thread* virtual é desmontada, liberando a *thread* do *kernel* para outras tarefas, e retoma a execução após a conclusão do I/O, armazenando as estruturas necessárias. Isso permite milhões de threads virtuais com custo reduzido e trocas de contexto eficientes (Bateman; Pressler, 2024).

No entanto, as *threads* virtuais não têm preempção, podendo monopolizar a *thread* do *kernel* até serem interrompidas por I/O, *Garbage Collector* ou o sistema operacional (Escoffier, 2023). Ainda em amadurecimento, práticas como *pools* de *threads* e uso excessivo de *synchronized* precisam de ajustes (Bateman; Pressler, 2024).

## 5. Trabalhos relacionados

Estudos comparativos entre Quarkus e Spring Boot mostram que o Quarkus reativo é mais eficiente em transações menores e no uso de recursos (CPU/memória), enquanto o Spring Boot performa melhor com grandes volumes de dados (Nordlund e Nordström, 2022). Montenegro e Nascimento Júnior (2023) destacam que o Spring Boot supera o Quarkus em tempo de resposta e taxa de transferência. Almeida (2020) observa que o Quarkus tem menor tempo de resposta em *builds* nativos, enquanto Giangreco (2024) aponta que o Spring Boot tem desempenho 290% superior, mas o Quarkus economiza 36% mais memória. Joo e Haneklint (2023) mostram que as *threads* virtuais no Spring Boot têm desempenho superior e maior resiliência, enquanto o Spring WebFlux teve resultados inferiores em alguns cenários. No entanto, nenhum estudo correlacionou diretamente o Quarkus com reatividade e o Spring Boot com *threads* virtuais em uma comparação direta entre essas abordagens específicas.

## 6. Metodologia

O estudo adotou uma abordagem quantitativa, de caráter experimental e comparativo. Foram desenvolvidos dois protótipos de API Rest idênticos que simulam uma biblioteca virtual, retornando ao usuário, conjuntos de livros com seus dados relacionados, tais quais: autores, editora, categoria e livros relacionados, um utilizando Spring Boot com *threads* virtuais e outro utilizando Quarkus com programação reativa. Os protótipos foram submetidos a testes de carga em diferentes cenários, com taxas de concorrência variando de baixa a alta.

Para a análise comparativa dos frameworks, foram realizados cálculos estatísticos abrangentes utilizando um conjunto de métricas de desempenho. A média

geométrica e a mediana foram calculadas para avaliar a tendência central dos tempos de resposta, enquanto o desvio padrão e o coeficiente de variação permitiram analisar a dispersão dos dados em relação à média.

Os intervalos de confiança foram estabelecidos a 95% utilizando a distribuição t de Student, com grau de liberdade 9 e valor crítico correspondente de 2,26. Para verificar a significância estatística das diferenças observadas entre os frameworks, foi aplicado o teste t para amostras independentes com nível de significância de 5% ($\alpha$ = 0,05), considerando um valor crítico de referência de 2,1009.

Esta abordagem metodológica permitiu não apenas quantificar as diferenças de desempenho entre as abordagens com threads virtuais do Spring Boot e a programação reativa do Quarkus, mas também determinar a relevância estatística dessas diferenças. A análise considerou especialmente a consistência dos resultados em diferentes cenários de carga e operações bloqueantes, garantindo uma avaliação robusta e confiável do desempenho comparativo.

### 6.1. Cenários de Testes

O projeto de teste foca em avaliar o desempenho de *threads* virtuais e da API reativa do Quarkus em cenários com operações de I/O bloqueantes, visando escalabilidade sem dependência excessiva de hardware. Foram definidas três rotas de teste com diferentes complexidades de I/O:

- Rota de Complexidade simples: 10 operações de I/O.
- Rota de Complexidade intermediária: 25 operações de I/O dependentes.
- Rota de Alta complexidade: 50 operações de I/O dependentes.

As taxas de concorrência, baseadas em Almeida (2024), foram divididas em:

- Baixa concorrência: 100 usuários virtuais, 10 requisições cada, em 10 segundos.
- Média concorrência: 500 usuários virtuais, 10 requisições cada, em 10 segundos.
- Alta concorrência: 1000 usuários virtuais, 10 requisições cada, em 10 segundos.

Os cenários de teste combinam complexidade de rotas e níveis de concorrência:

- Cenário A: Simples em baixa (A1), média (A2) e alta concorrência (A3).
- Cenário B: Intermediária em baixa (B1), média (B2) e alta concorrência (B3).
- Cenário C: Alta complexidade em baixa (C1), média (C2) e alta concorrência (C3).

### 6.2. Ferramentas Utilizadas

Os testes foram realizados utilizando o Apache JMeter para simular a carga de requisições, PostgreSQL como banco de dados, e Docker para virtualização do ambiente de testes. As aplicações foram desenvolvidas utilizando o IntelliJ IDEA.

## 7. Resultados e Discussões

O código-fonte dos protótipos desenvolvidos, bem como os resultados brutos obtidos durante a execução dos testes de desempenho, estão disponíveis publicamente para consulta no repositório: https://github.com/italomded/wperformance-2025.git. Essa disponibilização visa garantir a transparência do experimento e possibilitar a

reprodutibilidade dos dados apresentados nesta seção.

A Tabela 1 demonstra os resultados coletados através da execução dos testes e captura das métricas utilizando o JMeter.

**Tabela 1. Resultados demonstrados pela execução dos testes**

| Cenário | Protótipo | Requisições | Média (ms) | Mediana (ms) | Máximo (ms) | Desvio Padrão (ms) | Vazão (r/s) | Falha (%) | Teste T |
|---|---|---|---|---|---|---|---|---|---|
| A1 | Spring | 10000 | 29 | 26 | 408 | 18.54 | 7.59 | 0.000 | 0.9423 |
| A1 | Quarkus | 10000 | 44 | 30 | 721 | 46.80 | 6.23 | 0.000 | 0.9423 |
| A2 | Spring | 50000 | 2317 | 2480 | 10365 | 992.08 | 26.10 | 0.000 | 0.9480 |
| A2 | Quarkus | 50000 | 1975 | 2146 | 3178 | 563.43 | 29.18 | 0.000 | 0.9480 |
| A3 | Spring | 100000 | 4101 | 4549 | 14884 | 1282.60 | 48.87 | 0.000 | 1.1778 |
| A3 | Quarkus | 100000 | 4715 | 4899 | 6145 | 1035.67 | 54.29 | 0.000 | 1.1778 |
| B1 | Spring | 10000 | 358 | 397 | 966 | 123.56 | 6.42 | 0.000 | 3.7399 |
| B1 | Quarkus | 10000 | 650 | 676 | 1224 | 213.76 | 6.63 | 0.000 | 3.7399 |
| B2 | Spring | 50000 | 4841 | 5471 | 15671 | 1466.66 | 86.09 | 0.000 | 1.5852 |
| B2 | Quarkus | 50000 | 5788 | 5953 | 7578 | 1190.59 | 21.93 | 0.000 | 1.5852 |
| B3 | Spring | 100000 | 11485 | 12142 | 30161 | 3211.49 | 12.41 | 0.003 | 0.8762 |
| B3 | Quarkus | 100000 | 12506 | 13174 | 14356 | 1806.08 | 32.12 | 0.000 | 0.8762 |
| C1 | Spring | 10000 | 1611 | 1698 | 4687 | 661.40 | 6.51 | 0.000 | 1.1881 |
| C1 | Quarkus | 10000 | 1933 | 2121 | 2862 | 545.02 | 6.06 | 0.000 | 1.1881 |
| C2 | Spring | 50000 | 11588 | 13023 | 30139 | 3238.34 | 15.16 | 0.002 | 0.8367 |
| C2 | Quarkus | 50000 | 12575 | 13278 | 17191 | 1851.56 | 18.06 | 0.000 | 0.8367 |
| C3 | Spring | 100000 | 24276 | 26268 | 30191 | 5468.87 | 19.76 | 0.388 | 0.7222 |
| C3 | Quarkus | 100000 | 25640 | 26555 | 27690 | 2398.57 | 12.87 | 0.000 | 0.7222 |

Na Tabela 1, é possível visualizar que o Spring demonstrou predominância nos tempos de resposta, enquanto o Quarkus na consistência, apresentando números mais centrados na média, com desvio padrão menor e tempos máximos menores. Analisando do ponto de vista estatístico, conclui-se que apenas no Cenário B com média concorrência, foi identificada uma diferença estatística significativa nos tempos de resposta, na qual o Spring Boot apresentou vantagem de desempenho sobre o Quarkus.

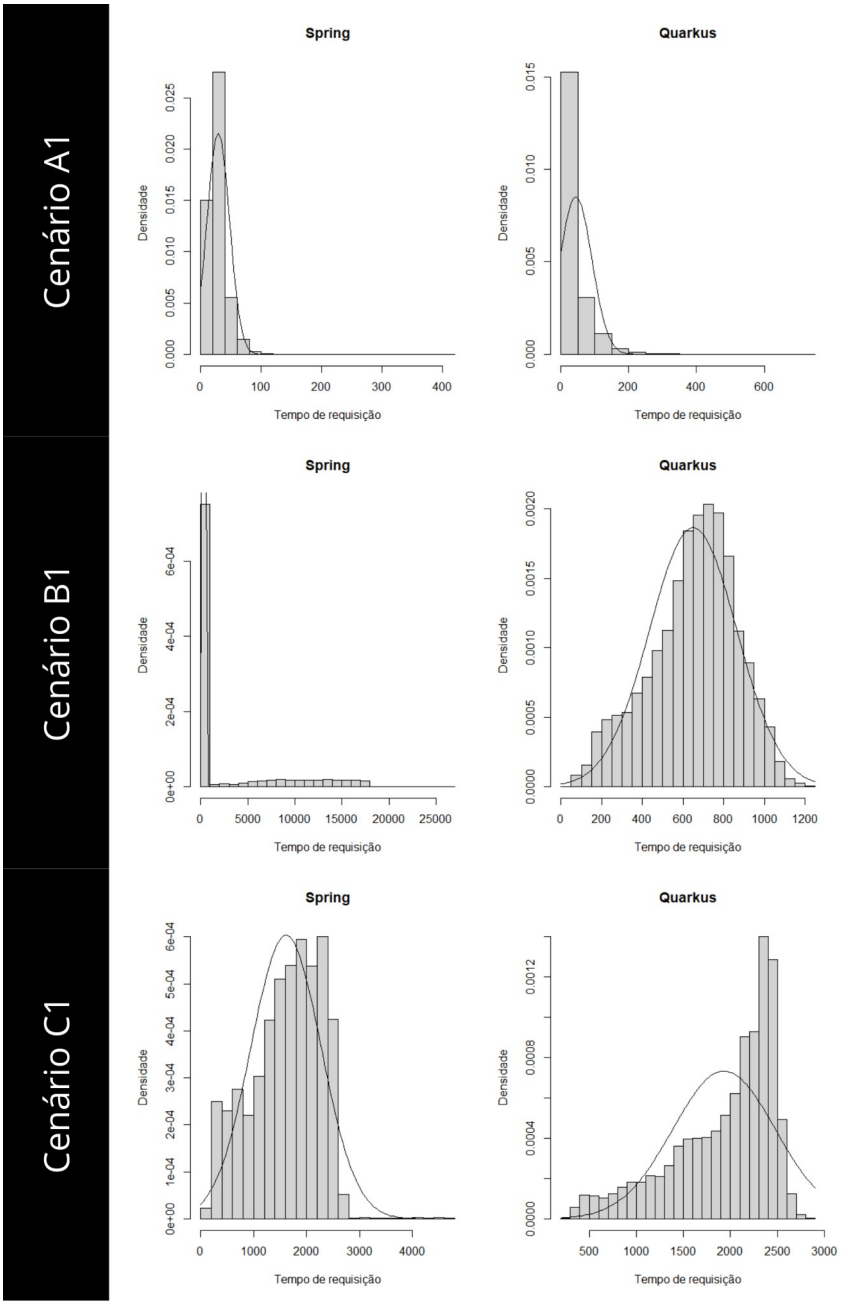


**Gráfico 1. Histograma dos cenários em baixa concorrência de cada rota.**

Os histogramas do Gráfico 1 representam os gráficos dos cenários menos desafiadores de cada rota para análise detalhada. No Cenário A1, o Spring apresentou tempos de resposta mais previsíveis e menos dispersos, enquanto o Quarkus, embora geralmente rápido, teve maior variabilidade. Ambos exibiram assimetria positiva, indicando que a maioria das requisições foi rápida, mas com alguns picos de latência.

No Cenário B1 (Gráfico 1), o Spring teve distribuição altamente assimétrica, com tempos majoritariamente baixos, mas alta variabilidade em algumas requisições. O Quarkus, por outro lado, mostrou distribuição aproximadamente normal, com respostas mais consistentes e previsíveis, vantajoso para sistemas que demandam estabilidade.

No Cenário C1 (Gráfico 1), o Spring apresentou distribuição próxima à normal,

com pico em ~2000 ms, enquanto o Quarkus teve leve assimetria, com pico em ~2500 ms, indicando diferenças sutis no comportamento sob carga.

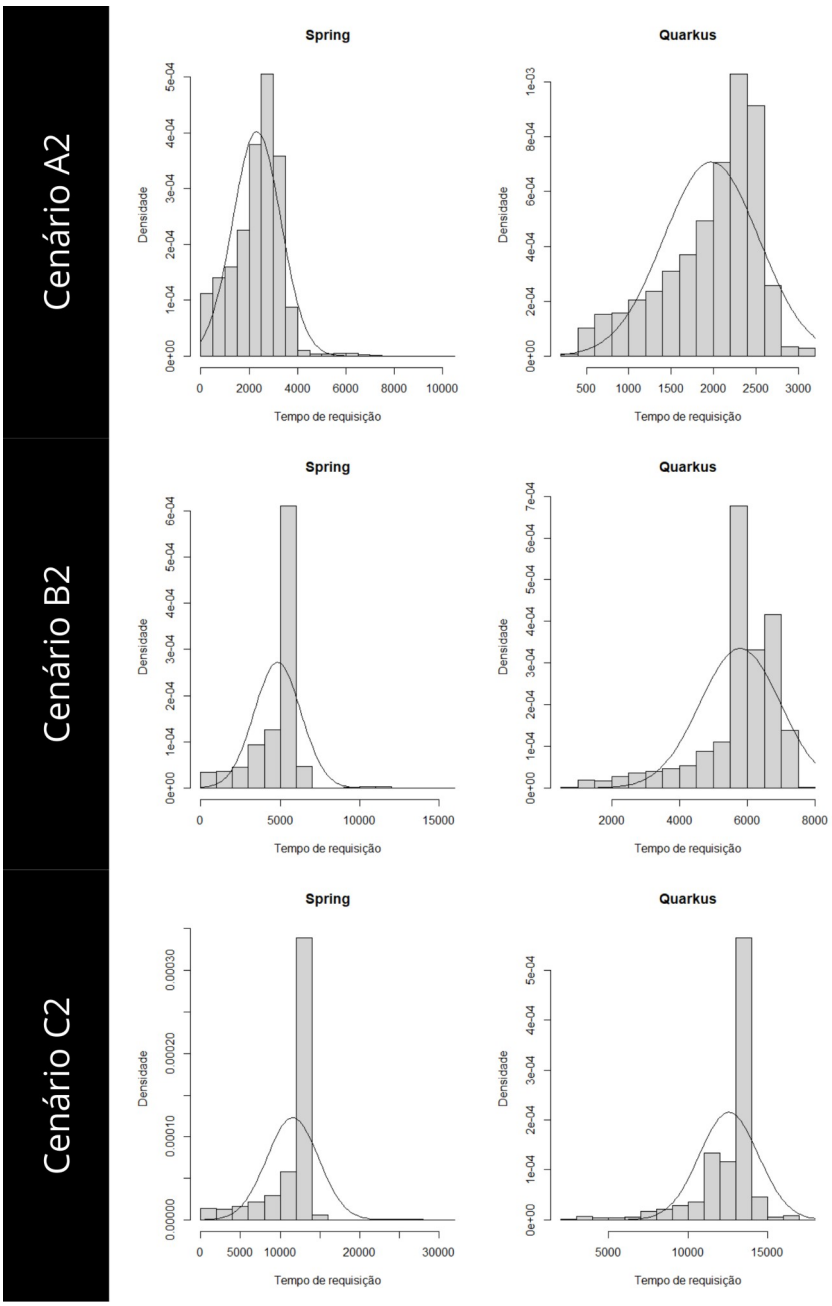


**Gráfico 2. Histograma dos cenários em média concorrência de cada rota.**

Para complementar a avaliação, foram analisados os histogramas dos cenários A2, B2 e C2 (Gráfico 2), que revelam padrões distintos de desempenho e estabilidade em carga média de concorrência.

No Cenário A2 (Gráfico 2), o Quarkus apresentou uma distribuição mais próxima da normal, indicando tempos de resposta previsíveis e consistentes. Já o Spring, ainda que rápido na maioria dos casos, exibiu uma assimetria positiva, sugerindo maior dispersão e eventuais picos de latência.

No Cenário B2 (Gráfico 2), ambos os frameworks demonstraram curvas

próximas da normal, com desempenho estável e tempos concentrados. O Spring registrou alguns valores máximos distantes da média (com pico em ~5000 ms), enquanto o Quarkus teve um pico em ~6000 ms e uma dispersão ligeiramente maior em torno da distribuição.

Por fim, no Cenário C2 (Gráfico 2), o Spring manteve uma curva relativamente normal, com pouca dispersão, mas com alguns valores extremamente altos (embora pouco frequentes). O Quarkus, por sua vez, apresentou uma distribuição normal com pico entre 10000 ms e 15000 ms, mostrando uma dispersão um pouco mais acentuada em relação ao pico, porém com maior estabilidade e ausência de *outliers* elevados.

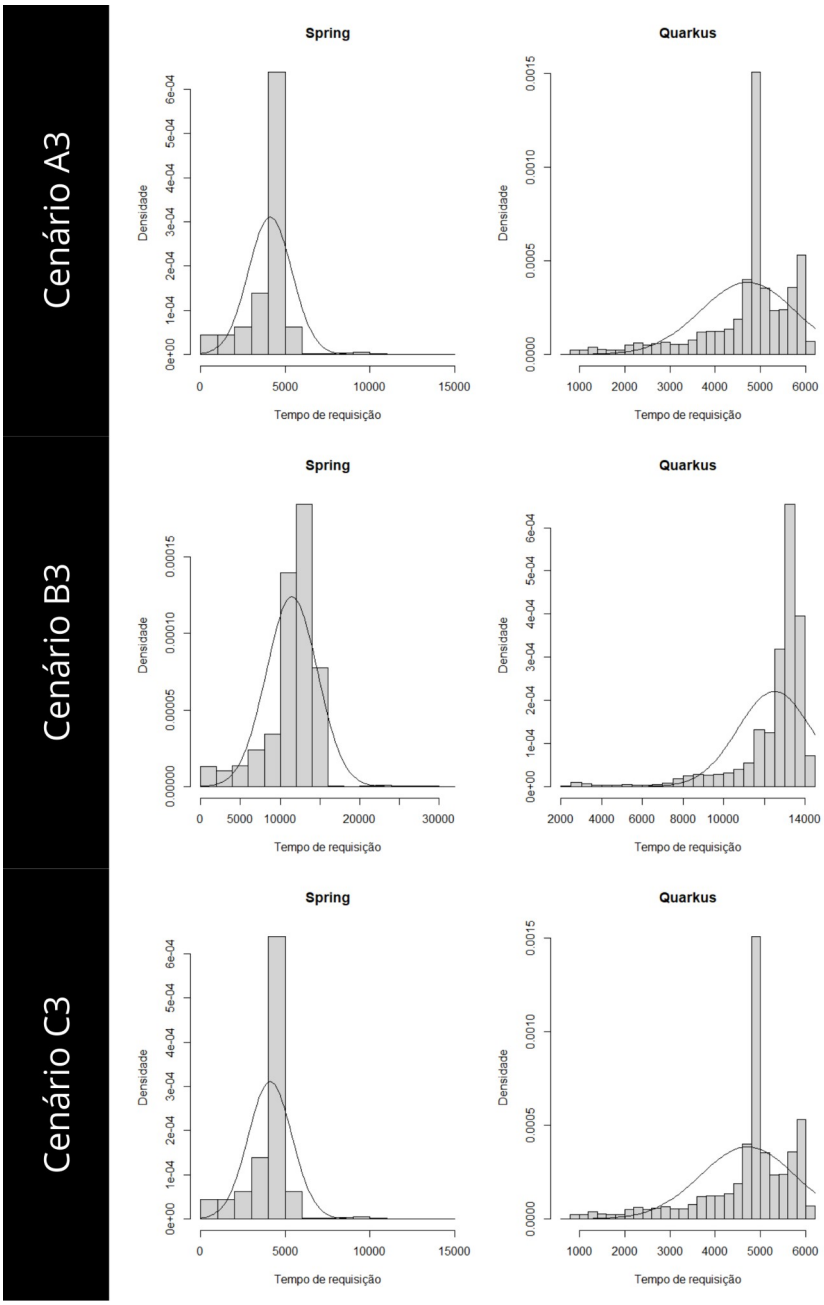


**Gráfico 3. Histograma dos cenários em alta concorrência de cada rota.**

Os histogramas do Gráfico 3 representam os gráfios dos cenários mais desafiadores de cada rota para análise detalhada. Dentro da representação do Cenário A3, o Quarkus apresentou uma curva de densidade mais suave, indicando um melhor balanceamento de carga sob alta concorrência. Já o Spring Boot mostrou uma concentração de tempos em um valor específico, sugerindo um limite estrutural nas threads virtuais.

Já no Cenário B3 (Gráfico 3), o Spring Boot teve uma distribuição assimétrica à direita, com concentração em torno da média. O Quarkus, também assimétrico, mostrou maior dispersão, com um pico acentuado em tempos mais baixos, indicando maior eficiência em processar requisições rapidamente sob carga alta.

Por fim, no Cenário C3 (Gráfico 3), ambos os *frameworks* enfrentaram desafios sob carga alta, com tempos de resposta elevados e concentrados, sugerindo limites de desempenho semelhantes em condições extremas.

Desta forma, respaldado pelos dados obtidos através da execução dos testes e análise, e respondendo as perguntas propostas pelo estudo, conseguimos concluir que:

- **Pergunta A:** O Quarkus não apresentou falhas em nenhum cenário, enquanto o Spring Boot teve taxas de falha de 0,002% no Cenário C (média concorrência) e 0,388% no Cenário C (alta concorrência). Assim, o Spring atingiu o ponto de falha antes do Quarkus.
- **Pergunta B:** O Spring Boot teve tempos médios e medianos melhores em valores absolutos, com diferença média dos tempos de resposta de 656 ms, porém, sem vantagem estatística comprovada. Mas o Quarkus mostrou maior consistência, com menores tempos máximos, desvios padrão e resultados menos dispersos.
- **Pergunta C:** O desempenho em vazão variou por cenário. O Quarkus superou o Spring nos Cenários A e B (alta concorrência), enquanto o Spring teve melhor desempenho no Cenário B (média concorrência) e no Cenário C (baixa e alta concorrência).
- **Pergunta D:** O Spring Boot com threads virtuais é mais simples de desenvolver e manter, mantendo o modelo tradicional. Já o Quarkus reativo exige maior complexidade de código, mas oferece maior confiabilidade e consistência em cenários críticos. A escolha depende das prioridades: simplicidade (Spring) ou robustez (Quarkus), visto que ambos não apresentaram vantagem estatística comprovada referente a tempos de resposta.

## 8. Conclusão

Este estudo comparou o desempenho do Spring Boot (com *threads* virtuais) e do Quarkus (com API reativa) no gerenciamento de operações de I/O bloqueantes, simuladas por interações com banco de dados. Ambos os *frameworks* mostraram-se viáveis para cenários de alta concorrência, mas com características distintas:

- O Quarkus destacou-se por sua consistência e confiabilidade, com tempos de resposta mais estáveis e menor variação sob alta concorrência.
- O Spring Boot obteve melhores médias de tempo de resposta absolutos, mas com desempenho estatisticamente equivalente ao Quarkus.

A escolha entre os *frameworks* depende das necessidades do projeto:

- Quarkus é ideal para cenários que exigem previsibilidade e estabilidade sob alta carga.
- Spring Boot é preferível para projetos que priorizam familiaridade, facilidade de adoção e um modelo de programação tradicional.

O estudo reforça a importância de considerar não apenas métricas de desempenho, mas também fatores como curva de aprendizado, custo de implementação e manutenibilidade. Como trabalhos futuros, propõe-se a ampliação dos cenários de teste, incluindo cargas prolongadas, análise de consumo de CPU e memória, além da avaliação de tarefas CPU-bound e workloads mais realistas, como tráfego variável e picos de uso. Sugere-se também o uso de ambientes mais próximos da produção com a ausência das ferramentas de virtualização, a aplicação de metodologias qualitativas para avaliar aspectos subjetivos (como experiência de desenvolvimento e curva de aprendizado), e uma análise mais aprofundada dos trabalhos relacionados.

Em suma, o trabalho contribui para a compreensão das capacidades e limitações das *threads* virtuais e APIs reativas, auxiliando desenvolvedores e arquitetos na escolha da abordagem mais adequada para aplicações escaláveis em Java.

## Referências


ALMEIDA, Matheus Santos de. Uma análise comparativa de desempenho entre diferentes tecnologias de execução de aplicações web do lado do servidor. 2020.

BATEMAN, Alan; PRESSLER, Ron. JEP 444: Virtual threads. OpenJDK, 2024. Disponível em: https://openjdk.org/jeps/444. Acesso em: 12 nov. 2024.

BONÉR, Jonas; FARLEY, Dave; KUHN, Roland; THOMPSON, Martin. O Manifesto Reativo. The Reactive Manifesto, 2014. Disponível em: https://www.reactivemanifesto.org/pt-BR. Acesso em: 30 de out. 2024.

CHOUBEY, Mayank. Spring Boot Webflux vs Quarkus: Performance comparison for hello world case. Tech Tonic, 2024. Disponível em: https://medium.com/deno-the-complete-reference/spring-boot-webflux-vs-quarkus-performance-comparison-for-hello-world-case-a3c0d1bb0286. Acesso em: 06 nov. 2024.

ESCOFFIER, Clement. When Quarkus meets Virtual threads. Quarkus, 2023. Disponível em: https://quarkus.io/blog/virtual-thread-1/. Acesso em: 12 nov. 2024.

GIANGRECO, Samuele. Sviluppo di Microservizi Java su AWS EKS: Confronto tra Spring Boot e Quarkus. Polito.it, 11 abr. 2024.

GILLIS, Alexander S.; NOLLE, Tom. Reactive programming. TechTarget, 2024. Disponível em: https://www.techtarget.com/searchapparchitecture/definition/reactive-programming. Acesso em: 30 de out. 2024.

HANEKLINT, Carl; JOO, Yo Han. Comparing Virtual Threads and Reactive Webflux in Spring: A Comparative Performance Analysis of Concurrency Solutions in Spring. 2023.

JETBRAINS. The State of Developer Ecosystem 2023. JetBrains, 2023. Disponível em: https://www.jetbrains.com/lp/devecosystem-2023/. Acesso em: 03 out. 2024.

MARIASOVA, Irina. Is Java Still Relevant Nowadays? JetBrains, 2024. Disponível em: https://blog.jetbrains.com/idea/2024/07/is-java-still-relevant-nowadays/#what-is-

java-used-for. Acesso em: 03 out. 2024.

MONTENEGRO, Matheus Albuquerque; NASCIMENTO JUNIOR, Francisco do. Um estudo comparativo entre as tecnologias Spring Boot e Quarkus na implementação do back-end de aplicações web com MongoDB. 2023.

NORDLUND, André; NORDSTRÖM, Niklas. Reactive vs Non-Reactive Java framework: A comparison between reactive and non-reactive APIs. 2022.

PRESSLER, Ron. Project Loom: Fibers and Continuations for the Java Virtual Machine. OpenJDK, 2018. Disponível em: https://cr.openjdk.org/~rpressler/loom/Loom-Proposal.html. Acesso em: 29 out. 2024.

QUARKUS. Quarkus reactive architecture. Quarkus, [s.d.]. Disponível em: https://quarkus.io/guides/quarkus-reactive-architecture. Acesso em: 30 out. 2024.

QUARKUS. Quarkus: Supersonic Subatomic Java. Quarkus, [s.d.]. Disponível em: https://quarkus.io/. Acesso em: 05 nov. 2024.TANENBAUM, Andrew S.; BOS, Herbert. Sistemas operacionais modernos. Tradução de Jorge Ritter. 4º. ed. São Paulo: Pearson Education do Brasil, 2016.

VERTX. Eclipse Vert.X and reactive in just a few words. Vert.X, [s.d.]. Disponível em: https://vertx.io/introduction-to-vertx-and-reactive/. Acesso em: 29 out. 2024.