Análise Comparativa: Threads Virtuais vs. Threads de Plataforma em APIs Spring Boot

Mikaell Miguel da Silva*, Pablo Henrique Ferreira da Silva*, Sandrírames Albino Fausto*
*Centro de Informática (CIn), Universidade Federal de Pernambuco (UFPE), Recife, Brasil
Email: {*mms14, *phfs2, *saf}@cin.ufpe.br

***Resumo*—A escalabilidade de aplicações web Java é tradicionalmente limitada pelo alto custo de recursos das threads do sistema operacional no modelo *thread-per-request*. Este trabalho apresenta uma análise comparativa quantitativa entre o novo modelo de Threads Virtuais (Java 21) e as Threads de Plataforma em uma API Spring Boot. Por meio de experimentos controlados em contêineres Docker sob alta concorrência e operações de I/O bloqueantes, os resultados demonstraram que as Threads Virtuais aumentaram a vazão (*throughput*) em cerca de 77% e mantiveram a latência estável, contrastando com a saturação observada no modelo tradicional. Observou-se também uma maior eficiência no uso de CPU e uma redução drástica na dependência de threads nativas, confirmando que as Threads Virtuais representam uma solução eficaz para ampliar a escalabilidade vertical de microsserviços limitados por I/O sem complexidade adicional.**

***Palavras-chave*—Java, Spring Boot, Threads Virtuais, Project Loom, Análise de Desempenho.**

***Abstract*—Java web application scalability is traditionally constrained by the high resource cost of operating system threads within the thread-per-request model. This paper presents a quantitative comparative analysis between the new Virtual Threads model (introduced in Java 21) and traditional Platform Threads in a Spring Boot API. Through controlled experiments using Docker containers under high concurrency and blocking I/O scenarios, results demonstrated that Virtual Threads increased throughput by approximately 77% and maintained stable latency, in contrast to the saturation and degradation observed in the traditional model. Furthermore, superior CPU efficiency and a drastic reduction in native thread dependency were observed, confirming that Virtual Threads represent an effective solution for enhancing the vertical scalability of I/O-bound microservices without introducing asynchronous complexity.**

***Index Terms*—Java, Spring Boot, Virtual Threads, Project Loom, Performance Analysis.**

I. Introdução

O desenvolvimento de aplicações web de alto desempenho e escalabilidade tem sido, historicamente, um dos principais desafios na engenharia de software, especialmente no ecossistema Java. A plataforma Java consolidou-se como o padrão da indústria para sistemas corporativos, apoiando-se predominantemente no modelo de concorrência baseado em threads, onde cada requisição HTTP é atendida por uma thread independente durante todo o seu ciclo de vida. Este modelo, conhecido como *thread-per-request*, oferece simplicidade no desenvolvimento e facilidade na depuração, alinhando-se ao estilo imperativo de programação [1].

No entanto, a implementação tradicional de concorrência em Java utiliza *Platform Threads* (threads de plataforma), que são wrappers diretos sobre as threads do sistema operacional (SO). Esta abordagem apresenta limitações arquiteturais significativas em cenários modernos de alta demanda. Como as threads do SO são recursos caros e finitos — consumindo memória considerável para suas pilhas e exigindo trocas de contexto custosas pelo agendador do kernel —, o número de requisições simultâneas que um servidor pode manipular torna-se limitado pela quantidade de threads que o hardware pode sustentar. Em aplicações intensivas em I/O (*Input/Output*), como APIs REST que frequentemente aguardam respostas de bancos de dados ou outros microsserviços, esse modelo resulta em desperdício de recursos, pois as threads permanecem bloqueadas e ociosas enquanto aguardam a conclusão de operações externas [2], [3].

Para mitigar essas limitações sem abandonar a simplicidade do modelo imperativo, o Java 21 introduziu oficialmente as *Virtual Threads* (Threads Virtuais) através da proposta JEP 444, oriunda do Projeto Loom. Diferentemente das threads tradicionais, as Threads Virtuais são entidades leves gerenciadas pela própria Máquina Virtual Java (JVM) e não possuem um mapeamento 1:1 com as threads do SO. Elas adotam um modelo M:N, permitindo que milhares de threads virtuais sejam executadas sobre um pequeno conjunto de threads de plataforma, prometendo revolucionar a escalabilidade de aplicações bloqueantes [3].

O *framework* Spring Boot, amplamente utilizado para construção de microsserviços, integrou suporte a essa tecnologia em suas versões recentes, permitindo que desenvolvedores alternem entre os modelos de concorrência via configuração. Contudo, a adoção dessa tecnologia exige uma compreensão empírica de seus benefícios reais e de seus impactos no consumo de recursos em comparação ao modelo estabelecido. [2]

Nesse contexto, este trabalho tem como objetivo realizar uma análise comparativa quantitativa entre o uso de Threads Virtuais e Threads de Plataforma em uma API Spring Boot. O estudo é guiado pelas seguintes questões de pesquisa, que definem os objetivos específicos desta investigação:

1) Avaliar em que medida a adoção de Threads Virtuais altera o desempenho da aplicação, considerando métricas de *throughput* (vazão), latência e taxa de erro em cenários de alta concorrência com operações bloqueantes.
2) Analisar o impacto no consumo de recursos computacionais, especificamente o uso de CPU e Memória RAM do contêiner da aplicação, comparando a eficiência energética e estrutural dos dois modelos durante picos de carga.
3) Investigar a correlação entre o aumento de usuários concorrentes e o comportamento do *pool* de threads, verificando a quantidade de threads de plataforma ativas necessárias para sustentar a carga em cada abordagem.

Para responder a essas questões, foi conduzido um experimento controlado utilizando contêineres Docker para isolamento de recursos, a ferramenta k6 para geração de carga sintética e o conjunto Prometheus/Grafana para monitoramento em tempo real. Os resultados obtidos visam fornecer subsídios técnicos para arquitetos e desenvolvedores na decisão de migrar ou adotar Threads Virtuais em seus projetos.

II. Fundamentação Teórica

A. Java

Java é uma plataforma de programação criada pela Sun Microsystems em 1995 com o objetivo de fornecer portabilidade e segurança por meio do conceito "write once, run anywhere". A linguagem e sua arquitetura foram formalizadas posteriormente na *Java Language Specification*, que define sua sintaxe, modelo de memória e suporte nativo a multithreading [4]. A execução das aplicações ocorre na Java Virtual Machine (JVM), descrita na Java Virtual Machine Specification, responsável por garantir independência de plataforma e otimizações de tempo de execução [5].

Ao longo de sua evolução, especialmente após sua incorporação pela Oracle, Java consolidou-se como uma das plataformas mais utilizadas em sistemas corporativos e distribuídos. Obras como *Java Concurrency in Practice* destacam que seu modelo de concorrência baseado em threads é um dos pilares para o desenvolvimento de aplicações servidoras de alto desempenho [1]. Esses aspectos tornam Java adequado para APIs modernas e justificam a análise de diferentes modelos de threads, como os tradicionais platform threads e as virtual threads introduzidas mais recentemente.

B. Spring Boot

Spring Boot é um framework oficial do ecossistema Spring projetado para simplificar o desenvolvimento de aplicações Java modernas. A documentação oficial descreve o framework como uma abordagem opinionated, que fornece configurações automáticas, dependências pré-selecionadas e servidores embutidos, permitindo que aplicações sejam inicializadas rapidamente e com mínima configuração manual [2]. Esse modelo reduz a complexidade inerente ao desenvolvimento de aplicações Java tradicionais, tornando o processo mais ágil e padronizado.

No contexto de aplicações web e APIs, Spring Boot fornece uma infraestrutura pronta para lidar com múltiplas requisições simultâneas, abstraindo detalhes internos de gerenciamento de servidores, componentes e execução. A documentação oficial destaca que uma das metas centrais do framework é favorecer escalabilidade, observabilidade e facilidade de operação em ambientes produtivos, características essenciais para sistemas orientados a alto volume de requisições [2].

Mais recentemente, versões modernas do Spring Boot (3.2+) passaram a integrar suporte direto às threads virtuais, por meio da propriedade de configuração spring.threads.virtual.enabled, conforme registrado em sua documentação de referência. Essa adição demonstra a evolução do framework para aproveitar o novo modelo de concorrência da JVM, permitindo que aplicações Spring Boot utilizem o estilo tradicional thread-per-request de forma mais eficiente em cenários de alta concorrência [2].

C. Threads

As threads são unidades básicas de execução dentro de um processo e permitem que múltiplas tarefas sejam realizadas de forma concorrente em um programa. No contexto da plataforma Java, uma thread representa um fluxo independente de controle capaz de executar instruções de maneira paralela a outras partes da aplicação. Segundo a especificação da linguagem Java, as threads possibilitam que programas explorem paralelismo e concorrência, oferecendo mecanismos para melhorar responsividade, utilização de recursos e desempenho em sistemas que lidam com múltiplas requisições simultâneas [1].

D. Threads de Plataforma

As *threads de plataforma* representam o modelo tradicional de execução do Java, no qual cada 'java.lang.Thread' corresponde diretamente a uma thread nativa do sistema operacional. Esse acoplamento implica custos elevados de criação, gerenciamento e troca de contexto, tornando esse modelo pouco eficiente em cenários com alto volume de requisições simultâneas — como em APIs REST desenvolvidas com Spring Boot. Segundo Goetz et al. [1], o uso intensivo de threads nativas tende a comprometer o desempenho devido à sobrecarga imposta ao sistema operacional, especialmente quando o número de threads ultrapassa a capacidade de escalonamento eficiente.

Outro ponto crítico é que threads de plataforma bloqueiam completamente ao executar operações de I/O, o que reduz a capacidade do servidor de processar requisições concorrentes. A literatura destaca que, para mitigar esse problema, frameworks tradicionais recorrem a pools de threads e estratégias assíncronas, mas essas abordagens adicionam complexidade sem eliminar o custo estrutural das threads nativas [1].

Dessa forma, embora adequadas para cargas moderadas, as threads de plataforma apresentam limitações significativas de escalabilidade em aplicações orientadas a requisições, motivando a busca por mecanismos mais leves — como as threads virtuais — para aumentar o throughput e reduzir latência em cenários de alta concorrência.

E. Threads Virtuais

As Threads virtuais foram introduzidas oficialmente no Java 21 por meio do JEP *(Java Enchancement Proprosal)* 444, como parte das melhorias de concorrência do Projeto Loom. Elas são implementadas como threads de usuário, gerenciadas pela JVM, e não mais mapeadas diretamente para threads do sistema operacional, diferentemente das platform threads. Essa abordagem reduz substancialmente o custo de criação e agendamento, permitindo que aplicações criem dezenas de milhares de threads de forma eficiente [3].

O modelo de execução das threads virtuais adota um esquema M:N, no qual muitas threads virtuais são multiplexadas sobre um número reduzido de threads de plataforma. Quando uma operação bloqueante ocorre (como I/O), a thread virtual é automaticamente suspensa e o *carrier thread* é liberado, melhorando a escalabilidade sem exigir o uso de modelos assíncronos complexos [3]. Esse comportamento possibilita o retorno ao estilo tradicional *thread-per-request*, porém com alta eficiência e menor consumo de recursos.

Por fim, destaca-se que threads virtuais são compatíveis com as APIs existentes do Java, facilitando sua adoção sem mudanças significativas na base de código.

F. Testes de Carga

Testes de carga são utilizados para avaliar o comportamento de um sistema quando submetido a um volume crescente e sustentado de requisições. Esses testes permitem identificar limitações de capacidade, pontos de saturação e variações na latência e throughput à medida que a demanda aumenta. Segundo Sommerville [6], testes de desempenho — incluindo testes de carga — são essenciais para verificar se requisitos não funcionais são atendidos, especialmente em sistemas distribuídos e aplicações web. Adicionalmente, modelos de qualidade como o ISO/IEC 25010 classificam o desempenho como um atributo crítico para sistemas confiáveis [7].

Em APIs Java, particularmente em arquiteturas que lidam com alta concorrência, testes de carga são fundamentais para comparar o comportamento entre diferentes modelos de concorrência, como *threads de plataforma* e *threads virtuais*, possibilitando avaliar impacto direto na escalabilidade e no uso de recursos.

Para a geração dessas cargas, podem ser utilizadas ferramentas amplamente documentadas e consolidadas, como:

1) K6: Ferramenta de código aberto mantida pela Grafana Labs. Sua documentação oficial descreve o uso de scripts em JavaScript para modelar cenários de carga, permitindo medir latência, erros, throughput e comportamento sob carga sustentada [8]. É frequentemente utilizada em pipelines de integração contínua e ambientes de teste distribuído.

2) Apache JMeter: Ferramenta mantida pela Apache Software Foundation, é uma ferramenta consolidada para testes de carga, estresse e endurance. A documentação oficial apresenta suporte para simulação de múltiplos usuários concorrentes e coleta de métricas detalhadas para aplicações web e protocolos variados [9]. Devido à sua maturidade e flexibilidade, JMeter é amplamente empregado em ambientes corporativos.

G. Docker

O Docker é uma plataforma aberta para desenvolvimento, envio e execução de aplicações que utiliza a virtualização em nível de sistema operacional para entregar software em pacotes isolados chamados contêineres. Segundo a documentação oficial, os contêineres são leves e contêm tudo o que é necessário para executar a aplicação (código, bibliotecas e dependências), compartilhando o kernel do sistema operacional do *host* sem a necessidade de um hipervisor pesado [10].

H. Observabilidade

A realização de testes de carga não é suficiente para compreender integralmente o comportamento interno de uma aplicação. Ferramentas de observabilidade permitem monitorar métricas de infraestrutura e aplicação durante o teste, fornecendo dados sobre consumo de CPU, memória, número de threads, latência interna, uso de pool de conexões, entre outros.

1) Prometheus: È uma ferramenta de monitoramento baseada em séries temporais. A documentação oficial descreve sua capacidade de coletar métricas expostas por aplicações e sistemas, permitindo análise detalhada do impacto da carga sobre os componentes internos do servidor [11].

2) Grafana: Plataforma oficial de visualização que integra-se diretamente ao Prometheus. A documentação do projeto descreve a criação de dashboards interativos para acompanhar métricas em tempo real, ajudando a identificar padrões, gargalos e degradações durante a execução dos testes [12].

III. Trabalhos Relacionados

A ascensão das Threads Virtuais (VT), formalizadas no Java 21, motivou uma série de investigações dedicadas a quantificar seus ganhos de eficiência e escalabilidade no contexto de aplicações concorrentes. Esta seção analisa os principais estudos que estabeleceram o cenário comparativo entre as Threads Virtuais e suas antecessoras, as Threads de Plataforma (PT).

Inicialmente, a pesquisa de Pandita [13] forneceu uma base empírica robusta, comparando diretamente o desempenho da VT e PT em cenários de alto *throughput*. Este trabalho foi crucial para distinguir os benefícios da VT: enquanto em cargas de trabalho intensivas em E/S (*I/O-intensive*) o uso de Virtual Threads demonstrou um aumento notável de *throughput* (superior a 60%) e redução de latência (cerca de 28%), para tarefas intensivas em CPU (*CPU-intensive*) o desempenho foi similar. Tal distinção reforça que a principal vantagem das Threads Virtuais reside na otimização da escalabilidade e no tratamento eficiente de operações de bloqueio, e não no aumento da velocidade bruta de processamento.

Ainda no âmbito da comparação de desempenho, Gustafsson e Persson [14] expandiram o escopo da análise para incluir o modelo de programação reativa (como o WebFlux), que historicamente tem sido a principal alternativa para alcançar alta concorrência em Java. O estudo demonstrou que tanto as Threads Virtuais quanto os sistemas reativos superaram as Threads de Plataforma em cenários de alta carga *I/O-heavy*. Este achado é fundamental, pois posiciona a VT como um modelo que permite obter níveis de escalabilidade comparáveis aos modelos assíncronos e não bloqueantes, mas preservando o paradigma de programação imperativo, que é mais simples e tradicional (*thread-per-request*).

Não obstante os ganhos evidentes, a literatura também aponta para os *trade-offs* e a necessidade de critérios de seleção bem definidos. Rosà et al. [15] abordaram essa limitação, destacando que a VT, embora otimize operações de bloqueio, pode introduzir uma sobrecarga de desempenho em tarefas puramente ligadas à CPU devido aos custos de gerenciamento de suas estruturas de dados. Em resposta a esse dilema, os autores propuseram o design de um *framework* de seleção adaptativa capaz de determinar dinamicamente o tipo de thread ideal em tempo de execução. Este trabalho salienta a importância de a escolha entre VT e PT depender criticamente da natureza da carga de trabalho e justifica a necessidade de avaliar como essas características se manifestam em um ambiente real de API Spring Boot.

Em síntese, a literatura confirma a superioridade das Threads Virtuais em escalabilidade *I/O-bound* e a equipara ao desempenho de sistemas reativos. Contudo, ela também impõe a necessidade de investigar a aplicação prática desses *trade-offs* em *frameworks* específicos de microsserviços, como é o objetivo central do presente trabalho.

IV. Metodologia

A metodologia de avaliação foi estruturada para quantificar de forma imparcial as diferenças entre a utilização de Threads Virtuais (VT) e Threads de Plataforma (PT) em um cenário de alta concorrência. O estudo foca em dois aspectos cruciais: o uso de recursos (CPU e Memória) e o desempenho da aplicação (Latência e Vazão).

A. Ambiente de Teste e Infraestrutura

A infraestrutura física utilizada para a hospedagem dos contêineres e execução do gerador de carga consistiu em uma estação de trabalho equipada com processador **AMD Ryzen 5 4600G** (6 núcleos e 12 *threads*, clock base 3.7GHz), **16GB de memória RAM DDR4** (3200MHz) e armazenamento via **SSD NVMe** de 512GB. O sistema operacional anfitrião foi o Microsoft Windows 11 (versão 25H2), com o Docker operando sobre o subsistema WSL2 (*Windows Subsystem for Linux*).

Todos os testes de carga foram executados sobre essa infraestrutura em um ambiente **isolado** e **consistente**, utilizando Docker para orquestração. Para garantir a previsibilidade no comportamento sob carga e isolar a variável de interesse (o modelo de *threading*), o contêiner da aplicação foi configurado com limites rígidos de recursos:

- **CPU:** Limite de 2 vCPUs
- **Memória:** Limite de 2 GB de RAM

Essa restrição de recursos foi adotada para evitar variações decorrentes de alocação dinâmica do *host* e assegurar que ambos os modelos (tradicional e *Virtual Threads*) fossem submetidos exatamente à mesma capacidade computacional, garantindo a paridade do experimento.

B. Monitoramento de Métricas

A coleta de métricas de desempenho foi realizada utilizando dois conjuntos de ferramentas:

1) **Métricas de Infraestrutura (Prometheus/Grafana):** Utilizadas para monitorar o consumo de recursos da máquina hospedeira da aplicação, incluindo **Uso de CPU** e **Uso de Memória Heap**.
2) **Métricas de Desempenho do Serviço (k6):** O gerador de carga **k6** foi configurado para coletar métricas diretamente relacionadas à qualidade de serviço sob concorrência. As métricas-chave coletadas incluem:
   - **Duração da Requisição HTTP:** Medida crítica de latência, registrando o tempo total de resposta. Foi o analisado o percentil *p(95)* e a média (*avg*) para avaliar a estabilidade da latência.
   - **Vazão (*Throughput*):** O número total de requisições processadas com sucesso pela aplicação por unidade de tempo (geralmente segundos), utilizado para medir a capacidade máxima de serviço sob alta concorrência.

C. Configuração da Aplicação e Cenário de Bloqueio

O *endpoint* testado simula uma operação **bloqueante** intensiva em E/S (*I/O-intensive*) ao introduzir um atraso de 200, ms por meio da função `Time.Sleep(200)` na lógica do serviço. Essa simulação permite avaliar o comportamento do sistema diante de operações de bloqueio e observar possíveis diferenças de desempenho na utilização de Threads Virtuais.

A configuração da aplicação Spring Boot foi controlada através do arquivo `application.properties`:

```
spring.threads.virtual.enabled=[true/false]
server.tomcat.threads.max=400
```

Para alternar entre os cenários (VT e PT), apenas a propriedade `spring.threads.virtual.enabled` foi modificada.

D. Definição de Cargas de Trabalho (Workloads)

O teste de carga foi realizado utilizando o *framework* **k6**, com a metodologia *ramping-vus* (Virtual Users) para simular um aumento gradual e sustentado da concorrência. Foram definidos três níveis de concorrência baseados no número de usuários virtuais (VU):

- **Baixa Concorrência:** 100 VU
- **Média Concorrência:** 500 VU
- **Alta Concorrência:** 1000 VU

O *script* k6 utilizado para o cenário de alta concorrência (1000 VU) é detalhado abaixo, e a duração total do teste foi de 3 minutos e 30 segundods:

```
export const options = {
  executor: 'ramping-vus',
  stages: [
    { duration: '1m', target: 1000 },
    { duration: '2m', target: 1000 },
    { duration: '30s', target: 0 },
  ],
};
```

A estratégia de ramping de usuários virtuais (VUs) foi adotada em todos os cenários de concorrência (100 VU, 500 VU e 1000 VU). Para garantir a comparabilidade dos resultados, apenas a quantidade de VUs no estágio de sustentação foi ajustada, mantendo-se inalterados os tempos de subida (ramp-up), sustentação e queda (ramp-down) da carga em todos os testes.

A utilização de uma estratégia de ramping é fundamental para evitar picos abruptos que poderiam distorcer as métricas de desempenho. Este aumento gradual de carga permite que o sistema passe por um período de aquecimento progressivo, alcance um estado estável de operação (steady state), e revele de forma mais fiel e consistente seu comportamento real sob diferentes níveis de pressão. Dessa forma, garante-se uma avaliação mais robusta e fidedigna da capacidade de resposta e eficiência da aplicação conforme o nível de concorrência é controlado.

V. Resultados e Discussão

Nesta seção, são apresentados os dados coletados durante os experimentos de carga realizados com o K6, bem como a análise do consumo de recursos (CPU e Memória) monitorados via Prometheus e Grafana. O objetivo é comparar o comportamento do Spring Boot utilizando o modelo tradicional de *Platform Threads* versus o novo modelo de *Virtual Threads* (Project Loom).

A. Análise de Throughput e Latência

A Tabela I resume as métricas de latência (tempo de resposta) e vazão (*throughput*) obtidas.

1) Cenários de Baixa e Média Carga (100 e 500 VUs): No cenário inicial de 100 VUs, ambos os modelos apresentaram desempenho idêntico. Como a carga não foi suficiente para saturar o *pool* de threads tradicional, o *overhead* de gerenciamento foi mínimo em ambos os casos. A latência média ($\approx$ 203ms) ficou muito próxima do tempo de bloqueio simulado (200ms), indicando ausência de filas.

Entretanto, ao elevar a carga para 500 VUs, começam a surgir divergências. O modelo Tradicional já apresenta uma degradação no percentil 95 ($p(95)$), subindo para 301ms, enquanto as Threads Virtuais mantiveram-se estáveis em 252ms. Isso sugere que o modelo tradicional começou a enfrentar contenção no agendamento de threads do Sistema Operacional (SO).

2) O Ponto de Saturação (1000 VUs): A disparidade torna-se crítica no cenário de 1000 VUs:

- **Threads Virtuais:** O sistema escalou quase linearmente. O *throughput* saltou para **3.229 req/s**, e a latência $p(95)$ manteve-se baixa (304ms), demonstrando que as requisições não ficaram paradas em filas de espera significativas.
- **Threads Tradicionais:** O sistema atingiu um teto claro de processamento. O *throughput* estagnou em **1.817 req/s** (quase 44% menor que o modelo virtual). A latência média dobrou e o $p(95)$ disparou para **581ms**.

Este comportamento confirma que, no modelo tradicional, uma vez esgotado o *pool* de threads disponíveis, as novas requisições entram em uma fila de espera, aumentando drasticamente o tempo de resposta percebido pelo cliente.

B. Eficiência de Recursos e Comportamento da JVM

Para compreender o custo computacional dessa escalabilidade, analisamos o consumo de CPU e Memória Heap durante o teste de estresse máximo (1000 VUs). A Fig. 1 ilustra o comportamento dos recursos durante a execução.

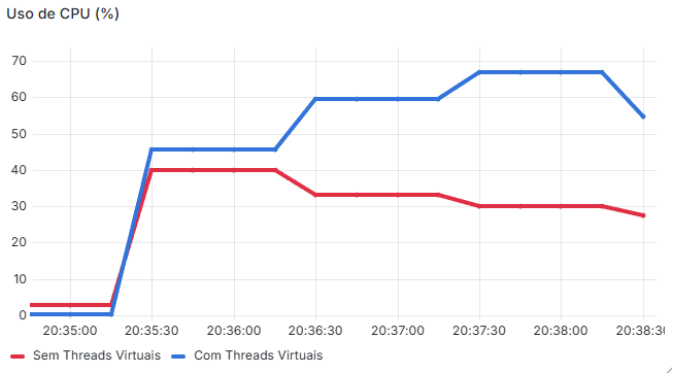

Fig. 1. Uso de CPU (%) durante o teste de alta concorrência. A linha azul representa o cenário com Threads Virtuais e a linha vermelha o cenário Tradicional

1) Análise de CPU: Observando o gráfico superior da Fig. 1:

- **Com Threads Virtuais (Linha Azul):** Atingiu cerca de 68% de uso.
- **Sem Threads Virtuais (Linha Vermelha):** Estagnou em cerca de 30%.

Neste contexto, **maior uso de CPU indica maior eficiência**. Como as Threads Virtuais não bloqueiam a thread do SO durante o I/O (`Thread.sleep`), a CPU pôde ser utilizada intensamente para processar o fluxo constante de entrada e saída de requisições. No modelo tradicional, a CPU ficou ociosa (30%) porque a maioria das threads estava bloqueada aguardando I/O, incapaz de realizar trabalho útil.

TABELA I
COMPARATIVO DE DESEMPENHO: LATÊNCIA E VAZÃO

Cenário (VUs)	Modelo	Requisições Totais	Throughput (req/s)	Latência Média	Latência $p(95)$
100 VUs	Tradicional	81.176	386,40	203,39 ms	205,98 ms
	Virtual	81.095	386,00	203,59 ms	206,38 ms
500 VUs	Tradicional	346.088	1.646,49	238,40 ms	301,46 ms
	Virtual	370.346	1.762,29	222,74 ms	252,49 ms
1000 VUs	Tradicional	382.060	1.817,75	432,10 ms	581,65 ms
	Virtual	**678.883**	**3.229,56**	**243,04 ms**	**304,89 ms**

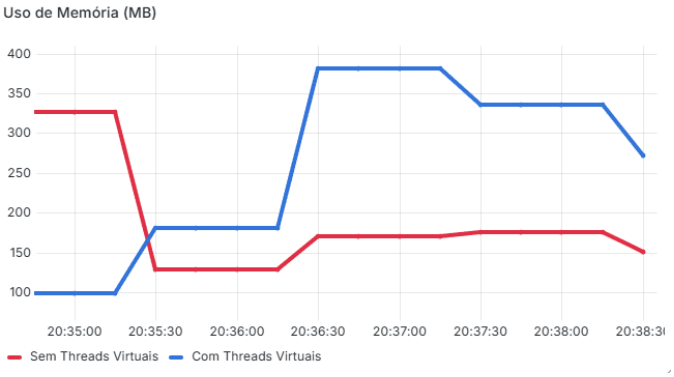

Fig. 2. Uso Memória Heap (MB) durante o teste de alta concorrência. A linha azul representa o cenário com Threads Virtuais e a linha vermelha o cenário Tradicional

2) Análise de Memória Heap: Observando gráfico de usp de memória Heap no cenário de Alta Concorrência da Fig. 2:

- **Com Threads Virtuais:** Pico de $\approx$ 380 MB.
- **Sem Threads Virtuais:** Estável em $\approx$ 175 MB.

As Threads Virtuais são objetos Java armazenados na memória *Heap*. Como o sistema processou quase o dobro de requisições por segundo (3.229 RPS vs 1.817 RPS), havia muito mais objetos "vivos" (requisições em andamento) no Heap simultaneamente, justificando o consumo maior. O modelo tradicional aparenta consumir menos Heap, mas isso não contabiliza o custo das pilhas (*stacks*) das threads nativas, que são alocadas fora do Heap.

3) Pico de Threads: Uma métrica fundamental coletada via *Prometheusr* revela a diferença arquitetural:

- **Sem Threads Virtuais (Pico):** 412 threads. O servidor precisou expandir seu *pool* até o limite para tentar atender a demanda.
- **Com Threads Virtuais (Pico):** 15 threads. A JVM utilizou apenas 15 *Carrier Threads* para carregar as milhares de requisições simultâneas.

C. Discussão

Os resultados obtidos corroboram a hipótese de que o modelo de *Virtual Threads* (Project Loom) endereça de forma eficaz o gargalo arquitetural de operações bloqueantes (*blocking I/O*) em aplicações Java baseadas em *Platform Threads*.

No modelo tradicional, a relação rígida 1:1 entre cada requisição e uma thread nativa do SO impõe um limite direto ao paralelismo. Assim que o número de requisições simultâneas se aproxima do tamanho máximo do *pool* configurado, novas operações competem por threads disponíveis, gerando filas internas e aumentando significativamente a latência. O comportamento observado nos testes confirma esse mecanismo: no pico de 1000 VUs, o servidor expandiu seu *pool* até 412 threads e, ainda assim, não conseguiu atender à demanda, resultando em um *throughput* limitado (1.817 req/s) e em um aumento acentuado da latência no percentil 95 ($p(95)$ = 581 ms). Além disso, a CPU permaneceu subutilizada, estabilizando em torno de apenas 30% do uso permitido, o que indica que a maior parte das threads estava ociosa, aguardando conclusão de I/O.

O modelo de *Virtual Threads*, por sua vez, adota uma relação M:N em que milhares de threads leves são multiplexadas sobre um conjunto pequeno de *Carrier Threads*. Durante operações de bloqueio, a JVM desassocia a thread virtual do *carrier*, permitindo que a thread nativa continue atendendo outras requisições. Nos experimentos realizados, esse comportamento resultou em uso significativamente mais eficiente da CPU (cerca de 68% no pico), refletindo maior capacidade real de processamento. O servidor permaneceu responsivo mesmo sob alta concorrência, alcançando um *throughput* de 3.229 req/s — um aumento de aproximadamente 77% em relação ao modelo tradicional — e mantendo a latência em níveis previsíveis ($p(95)$ = 304 ms).

O aumento no uso de memória Heap no cenário com *Virtual Threads* também é coerente com a arquitetura: como mais requisições são processadas simultaneamente, mais objetos transitórios permanecem alocados, resultando em um pico aproximado de 380 MB. Entretanto, esse acréscimo é compensado pela eliminação do custo de criação e gerenciamento de centenas de pilhas nativas (que no modelo tradicional são alocadas fora do Heap). Em outras palavras, o modelo virtual desloca a pressão para a Heap, mas reduz a pressão sobre o subsistema de threads do SO e sobre o agendador, que são gargalos substanciais em cargas de I/O intensivas.

Outro ponto relevante é o comportamento sob estresse extremo. Enquanto a arquitetura tradicional apresentou um ponto de saturação claro — evidenciado pela queda na vazão e crescimento da fila interna — as *Virtual Threads* mostraram resiliência e linearidade na escala. O número de threads nativas permaneceu praticamente constante (15 *Carrier Threads*), evidenciando a estabilidade do modelo mesmo diante de milhares de tarefas simultâneas. Essa característica reforça o benefício estrutural das *Virtual Threads*: ao remover o vínculo rígido entre thread e requisição, o servidor pode lidar com picos abruptos de demanda com mínima sobrecarga adicional.

Em síntese, a adoção de *Virtual Threads* amplia substancialmente o potencial de escalabilidade de aplicações Java orientadas a I/O. Os resultados demonstram que, sob as mesmas restrições de hardware, o modelo virtual proporciona maior utilização dos recursos disponíveis, reduz a probabilidade de formação de filas internas e mantém latências mais consistentes. Embora a maior ocupação de Heap represente um *trade-off*, ela se mostra justificável diante do ganho expressivo em throughput e da eliminação das limitações impostas pelo agendamento de threads do sistema operacional. Esses achados indicam que o Project Loom é uma alternativa promissora para aplicações que dependem de alto volume de conexões simultâneas, como APIs REST, Gateways e serviços orientados a eventos. Além disso, essa eficiência permite alcançar níveis de desempenho equiparáveis a soluções reativas mantendo a simplicidade do paradigma imperativo síncrono, o que facilita a manutenção do código e otimiza o custo-benefício da infraestrutura em nuvem.

VI. Conclusão

Este trabalho teve como objetivo analisar quantitativamente o impacto da adoção de Threads Virtuais (Project Loom) em uma API REST desenvolvida com Spring Boot, comparando-a com o modelo tradicional de Threads de Plataforma sob cenários de alta concorrência e operações de I/O bloqueantes. Os experimentos realizados permitiram responder às questões de pesquisa propostas e evidenciaram as mudanças arquiteturais trazidas pelo Java 21.

Em resposta à primeira questão de pesquisa (QP1), os resultados demonstraram que as Threads Virtuais oferecem uma superioridade clara em desempenho para cargas de trabalho limitadas por I/O. No cenário de estresse máximo (1000 usuários simultâneos), a aplicação com Threads Virtuais alcançou um aumento de aproximadamente 77% na vazão (*throughput*) em comparação ao modelo tradicional. Além disso, a latência manteve-se estável e previsível, evitando a degradação exponencial observada no modelo de Threads de Plataforma, onde o tempo de resposta no percentil 95 quase duplicou devido ao esgotamento do *pool* de threads.

Quanto ao consumo de recursos (QP2), observou-se uma mudança paradigmática no perfil de utilização do servidor. O modelo de Threads Virtuais permitiu um uso mais intensivo e eficiente da CPU (atingindo 68% de uso contra 30% do modelo tradicional), indicando que o processador passou menos tempo ocioso aguardando operações de I/O e mais tempo processando requisições efetivas. Embora tenha havido um aumento no consumo de memória Heap, este comportamento é um *trade-off* arquitetural esperado e justificável, uma vez que substitui a alocação custosa de memória nativa (pilhas de threads do SO) por objetos gerenciados pela JVM, resultando em uma escalabilidade vertical mais eficiente dentro dos mesmos limites de hardware.

A investigação sobre o comportamento do *pool* de threads (QP3) confirmou a eliminação do vínculo 1:1 entre requisições e threads do sistema operacional. Enquanto o modelo tradicional saturou o limite configurado de 412 threads nativas e ainda assim formou filas de espera, o modelo de Threads Virtuais sustentou uma carga de trabalho quase duas vezes maior utilizando apenas 15 threads de plataforma (*Carrier Threads*). Isso comprova que o agendamento em modo de usuário realizado pela JVM é capaz de multiplexar milhares de tarefas com overhead mínimo.

Conclui-se, portanto, que as Threads Virtuais representam uma evolução significativa para o desenvolvimento de microsserviços em Java, especialmente para aplicações como APIs REST e Gateways que lidam com alto volume de conexões simultâneas. A migração para este modelo no Spring Boot mostrou-se transparente e altamente benéfica para a escalabilidade.

Como trabalhos futuros, sugere-se investigar o comportamento das Threads Virtuais em cenários intensivos em CPU, onde o overhead de troca de contexto pode não ser vantajoso, bem como analisar o impacto desse modelo em outros recursos compartilhados, como *pools* de conexões de banco de dados, que podem tornar-se os novos gargalos em sistemas altamente concorrentes.

Referências

[1] B. Goetz, T. Peierls, J. Bloch, J. Bowbeer, D. Holmes, and D. Lea, *Java Concurrency in Practice*. Addison-Wesley, 2006.

[2] Spring Team, "Spring Boot Reference Documentatation," Disponível em: hhttps://docs.spring.io/spring-boot, acessado em: 16 Nov. 2025.

[3] R. Pressler and A. Bateman, "JEP 444: Virtual Threads," Disponível em: https://openjdk.org/jeps/444, OpenJDK, 2023, acessado em: 16 Nov. 2025.

[4] J. Gosling, B. Joy, G. Steele, G. Bracha, and A. Buckley, "The Java Language Specification, Java SE 21 Edition," Disponível emt: https://docs.oracle.com/javase/specs/jls/se21/html/index.html, acessado em: 15 Nov. 2025.

[5] T. Lindholm, F. Yellin, G. Bracha, A. Buckley, and D. Smith, "The Java® Virtual Machine Specification, Java SE 21 Edition," Disponível em: https://docs.spring.io/spring-boot/docs/current/reference/html/, acessado em: 15 Nov. 2025.

[6] I. Sommerville, *Sofware Engineering*, 10th ed. Pearson, 2016.

[7] International Organization for Standardization, "ISO/IEC 25010:2011, Systems and Software Quality Models," 2011.

[8] Grafana Labs, "k6 Documentation," Disponível em: https://k6.io/docs/, acessado em: 16 Nov. 2025.

[9] Apache Software Foundation, "Apache JMeter User Manual," Disponível em: https://jmeter.apache.org/usermanual/, acessado em: 16 Nov. 2025.

[10] Docker Inc., "Docker Overview," Disponível em: https://docs.docker.com/get-started/overview/, 2024, acessado em: 10 Dez. 2025.

[11] Prometheus, "Prometheus Documentation," Disponível em: https://prometheus.io/docs/, acessado em: 16 Nov. 2025.

[12] Grafana Labs, "Grafana Documentation," Disponível em: https://grafana.com/docs/, acessado em: 16 Nov. 2025.

[13] V. Pandita, "Benchmarking the Performance of Java Virtual Threads in High-Throughput Workloads," MSc Research Project, National College of Ireland, 2024, disponível em: https://norma.ncirl.ie/8134/1/visheshpandita.pdf.

[14] E. Gustafsson and O. N. Persson, "Comparison of Concurrency Technologies in Java (Structured Testing in a High Load Environment)," Master's Thesis, Lund University, 2024, disponível em: https://lup.lub.lu.se/student-papers/record/9166685/file/9166687.pdf.

[15] A. Rosà, M. Basso, L. Bohnhoff, and W. Binder, "Adaptive Thread Type Selection on the Java Virtual Machine," in *2023 Eleventh International Symposium on Computing and Networking Workshops (CANDARW)*. IEEE, 2023, pp. 418–423, disponível em: https://ieeexplore.ieee.org/document/10409841/.