Spring MVC vs. WebFlux em Cargas I/O-*bound*: Reavaliando a Superioridade do Modelo Reativo

Beatriz Fernandes Teixeira
Universidade Federal de Lavras
Departamento de Computação
Lavras, Minas Gerais, Brasil
beatriz.teixeira@estudante.ufla.br

Ricardo Terra
Universidade Federal de Lavras
Departamento de Computação
Lavras, Minas Gerais, Brasil
terra@ufla.br

Resumo

Modelos reativos, como o Spring WebFlux, são comumente apresentados como alternativas superiores ao Spring MVC em aplicações web que atendem muitas requisições simultâneas sob cargas I/O-*bound*. Este artigo curto da trilha de graduação compara empiricamente os dois modelos sob carga concorrente com acesso a banco de dados, considerando o impacto da configuração do modelo bloqueante e da capacidade computacional disponível. Os resultados indicam que o WebFlux supera em latência e vazão o MVC quando o *pool* de threads está subdimensionado, mas que essa vantagem diminui quando o MVC é adequadamente configurado. Assim, a migração para WebFlux não deve ser imediata; antes, deve considerar ajustes de configuração e escalonamento vertical.

Palavras-chave

Programação concorrente, Modelos de concorrência, Spring MVC, Spring WebFlux

1 Introdução

Em serviços web Java, o modelo de concorrência *thread-per-request*, implementado pelo Spring MVC, tem sido a abordagem historicamente predominante para processar requisições HTTP. Esse *framework*, utilizado por 65% dos desenvolvedores Java que responderam à pesquisa da JetBrains de 2025 [5], é o módulo do Spring Framework dedicado a aplicações web síncronas e foi a primeira opção introduzida pelo Spring para esse fim. Por sua simplicidade conceitual e por contar com um grande ecossistema de bibliotecas desenvolvidas para esse modelo, como Spring Security e Spring Data JPA [12], o modelo imperativo e bloqueante do MVC consolidou-se ao longo de duas décadas como a escolha predominante do ecossistema. Com o crescimento contínuo do número de usuários simultâneos em serviços online, aplicações baseadas nesse modelo passaram a enfrentar níveis cada vez mais elevados de concorrência, tornando mais evidentes os limites da abordagem bloqueante.

Em 2014, o Manifesto Reativo [2] propôs princípios para sistemas capazes de manter responsividade, resiliência e elasticidade sob carga variável. Esse movimento influenciou o lançamento do Spring WebFlux em 2017, uma alternativa assíncrona e não-bloqueante baseada no modelo de concorrência *event loop*. Por essa característica, o WebFlux é frequentemente associado a cenários com grande volume de operações de E/S concorrentes. Entretanto, em sistemas Java já consolidados sobre Spring MVC, a adoção de um modelo reativo pode implicar mudanças arquiteturais e operacionais relevantes.

Essa questão motiva este artigo curto da trilha de graduação, que investiga se a vantagem esperada do modelo reativo se mantém quando o MVC é configurado para cenários de alta concorrência. Para isso, foram desenvolvidas duas aplicações Java/Spring Boot funcionalmente equivalentes, ambas acessando PostgreSQL em uma carga I/O-*bound* com 1500 usuários virtuais simultâneos [3]. O *pool* de *threads* do MVC foi variado em seis níveis (250, 500, 750, 1000, 1250 e 1500 *threads*) e os experimentos foram realizados em duas configurações de hardware (4 e 8 vCPUs) para observar o efeito do escalonamento vertical sobre os dois modelos. A análise estatística utiliza Shapiro-Wilk seguido de teste *t* de Welch ou Wilcoxon *rank-sum*, de acordo com a distribuição das amostras, além de intervalos de confiança (IC) de 95%.

Os resultados indicam que sistemas MVC conseguem sustentar cargas limitadas por E/S altamente concorrentes quando o *pool* de *threads* e a CPU disponível são dimensionados adequadamente. Em 4 vCPUs, o WebFlux apresenta menor latência quando o *pool* do MVC é insuficiente. Com 750 *threads*, porém, o MVC alcança P95 cerca de 12% menor que o WebFlux. Em 8 vCPUs, os cenários do MVC com *pool* a partir de 750 *threads* aproximam-se do WebFlux em P50, P95 e *throughput*, restando como principal diferença o consumo de CPU, cerca de 30 % maior no WebFlux. As observações experimentais sugerem que aplicações MVC existentes podem se beneficiar de ajustes no *pool* e do aumento de CPU disponível antes de uma migração para o modelo reativo.

2 Paradigmas de Programação e Modelos de Execução

Um paradigma de programação define como o desenvolvedor estrutura a lógica da aplicação. O Spring MVC segue o paradigma imperativo, no qual o código descreve, passo a passo, o que fazer e em que ordem. Cada linha é executada na sequência e seu resultado fica imediatamente disponível para a próxima. A Listagem 1 mostra esse comportamento: a busca pelo usuário ocorre na primeira linha e a resposta HTTP é construída em seguida, sobre o valor já obtido.

Listagem 1: Fluxo imperativo no Spring MVC.

```
@GetMapping("/users/{id}")
public ResponseEntity<User> getUser(
        @PathVariable Long id) {
    User user = userRepository.findById(id);
    return ResponseEntity.ok(user);
}
```

O Spring WebFlux segue o paradigma reativo, no qual o código descreve transformações sobre valores que ainda não existem. Em vez de pedir o usuário e esperar a resposta, o método declara uma cadeia de operações que será aplicada quando o resultado estiver disponível. A Listagem 2 mostra essa abordagem: o método retorna

imediatamente um `Mono<UserDTO>`, uma referência ao `UserDTO` que será produzido a partir do usuário entregue pelo banco.

Listagem 2: Fluxo reativo no Spring WebFlux.

```
@GetMapping("/users/{id}")
public Mono<UserDTO> getUser(
        @PathVariable Long id) {
    return userRepository.findById(id)
        .map(user -> new UserDTO(
            user.getId(), user.getName()));
}
```

Embora Spring MVC e Spring WebFlux também difiram no estilo de programação (imperativo *vs.* reativo/funcional), este artigo investiga especificamente o **modelo de execução** e, em particular, o modelo de E/S subjacente: síncrono e bloqueante (*thread-per-request*) *vs.* assíncrono e não bloqueante (*event loop*). Esses aspectos constituem eixos independentes; portanto, as diferenças de desempenho aqui reportadas são atribuídas ao modelo de E/S, e não ao estilo de programação em si. No Spring MVC sobre Tomcat, esse modelo é o *thread-per-request*, em que cada requisição HTTP é atribuída a uma *thread* obtida de um *pool* limitado, configurado, por exemplo, pelo parâmetro `server.tomcat.threads.max`. Durante uma chamada JDBC síncrona ao banco, essa *thread* permanece bloqueada até a resposta retornar e, nesse intervalo, não pode atender novas requisições. Cada *thread* JVM reserva uma pilha de tamanho configurável, na ordem de centenas de KB a 1 MB, de modo que o consumo de memória cresce com o número de *threads*. Quando a concorrência excede a capacidade do *pool*, novas requisições precisam aguardar até que uma *thread* seja liberada, aumentando o tempo de resposta percebido. Esse comportamento é formalizado pela Lei de Little [6], expressa por $L = \lambda \cdot W$, em que L é o número médio de requisições em processamento, λ a vazão (requisições por segundo) e W o tempo médio de serviço por requisição. No modelo *thread-per-request*, L é limitado pelo tamanho do *pool*, o que dá um teto teórico de vazão explorado na Seção 4. A Figura 1 esquematiza esse modelo.

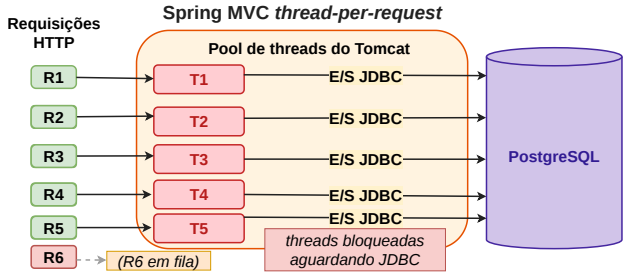

Figura 1: Modelo *thread-per-request* adotado pelo Spring MVC sobre Tomcat.

O *event loop* é um **modelo de execução** no qual poucas *threads* monitoram eventos de E/S e retomam o processamento por *callbacks*, sem bloquear durante a espera. No Spring WebFlux, esse modelo é adotado sobre o Reactor Netty [9]. As *threads* de *event loop*, dimensionadas por padrão a partir do número de processadores disponíveis, iniciam operações de E/S, registram um *callback*, isto é, a próxima etapa do processamento a ser executada quando o resultado estiver disponível, e retornam ao *loop* para atender outras requisições. Assim, o aumento no número de requisições simultâneas não exige, em princípio, um aumento proporcional no número de *threads*. Essa característica, contudo, depende de uma pilha de E/S não-bloqueante, como R2DBC em vez de JDBC no acesso a banco. A Figura 2 ilustra esse modelo.

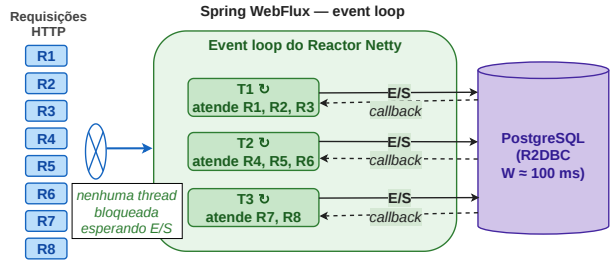

Figura 2: Modelo *event loop* adotado pelo Spring WebFlux sobre Reactor Netty. Um número reduzido de *threads* atende múltiplas requisições simultaneamente sem bloqueio em E/S.

Comparações entre modelos bloqueante e não-bloqueante em servidores web já haviam sido exploradas em trabalhos como o SEDA [13]. No ecossistema Java, trabalhos recentes têm investigado os custos e benefícios dos modelos bloqueante e não bloqueante. Sukhambekova [11] contrasta Spring MVC e Spring WebFlux quanto à arquitetura, escalabilidade e aos cenários de aplicação. Já de Deus et al. [3] comparam *frameworks* Java, incluindo Spring e Quarkus, sob modelos reativos e baseados em *threads* virtuais. Zbarcea et al. [14] avaliam os modelos *thread-per-request* e *event loop* em cargas de *cache*, CPU-*bound* e E/S bloqueante, observando que as diferenças entre eles dependem do recurso que limita a execução, com distintos custos de CPU e memória. Ponge et al. [8], por sua vez, analisam a sobrecarga de bibliotecas reativas em Java e mostram que seus benefícios de desempenho dependem das características da carga avaliada. Diferentemente desses trabalhos, este artigo manipula explicitamente o dimensionamento do *pool* de *threads* do MVC e a quantidade de CPU disponível, buscando identificar sob quais condições a vantagem do modelo não bloqueante se mantém.

3 Experimento

Esta seção descreve o objeto de estudo, a infraestrutura, o protocolo de carga, os cenários avaliados e os procedimentos estatísticos empregados na comparação empírica entre Spring MVC (modelo *thread-per-request*) e Spring WebFlux (modelo *event loop*) sob carga I/O-*bound* de banco de dados.

3.1 Aplicações e *Workload*

Foram construídas duas aplicações Java/Spring Boot funcionalmente equivalentes, ambas expondo o *endpoint* `GET /api/io`. A aplicação `app-mvc-io-db` utiliza Spring MVC com acesso ao PostgreSQL a partir dos componentes de banco de dados do Spring Data JPA/JDBC. A aplicação `app-webflux-io-db` utiliza Spring WebFlux sobre Reactor Netty, com acesso não-bloqueante ao PostgreSQL via Spring Data R2DBC.

Em ambas as aplicações, o *pool* de conexões com o banco foi configurado com 1500 conexões, o mesmo valor da concorrência

de usuários virtuais. Essa escolha não representa uma configuração comum em produção, mas o objetivo foi apenas evitar que a falta de conexões se tornasse o principal gargalo do teste. Desse modo, foi possível observar de forma mais isolada o efeito do modelo de execução do servidor de aplicação sobre o tempo de resposta quando muitas requisições chegam ao mesmo tempo.

O *workload* E/S-*bound* foi implementado com uma consulta direta ao PostgreSQL que chama a função `pg_sleep(0.1)`, fazendo o banco esperar cerca de 100 ms antes de responder cada requisição simulando uma operação de E/S com tempo previsível, sujeita à variação natural do ambiente. Como a conexão fica ocupada durante toda a consulta, o experimento também preserva a disputa por conexões sob alta concorrência, que é uma característica comum em *workloads* bloqueantes que dependem de banco de dados.

O *endpoint* `GET /api/io` executa uma única consulta nativa que seleciona uma linha por chave primária (`WHERE id = 1`), condicionada à função `pg_sleep`, e devolve uma resposta JSON pequena e de tamanho fixo. Não há paginação, junções, processamento adicional na aplicação nem computação CPU-*bound*: o custo de cada requisição concentra-se na espera pela E/S do banco. A consulta é idêntica nas duas aplicações, variando apenas a pilha de acesso a dados (Spring Data JPA/JDBC no MVC e Spring Data R2DBC no WebFlux). Essa escolha isola o modelo de execução como principal fator de variação, mas também restringe a generalização dos resultados, conforme discutido na Seção 5.

3.2 Infraestrutura

O experimento foi conduzido em três instâncias EC2 da Amazon Web Services, todas com o sistema operacional Amazon Linux 2023 e OpenJDK 21, a versão LTS com maior adoção entre desenvolvedores Java (40 %) [5]. A instância da aplicação (c7i.xlarge, 4 vCPUs/8 GB, na Fase A e c7i.2xlarge, 8 vCPUs/16 GB, na Fase B, única variação entre as duas fases) hospeda a JVM Spring Boot. Uma segunda instância (c7i.2xlarge) executa o PostgreSQL 15 em Docker (`max_connections=1500`) e uma terceira (c7i.large) executa o gerador de carga k6. A separação em três EC2s se alinha ao princípio de isolamento de componentes de Jain [4], de modo a evitar interferências entre cliente, banco e o processo Java.

As instâncias foram alocadas na mesma *Virtual Private Cloud* e zona de disponibilidade da AWS, com o objetivo de reduzir a latência e a variabilidade da rede. As aplicações executaram sobre OpenJDK 21 (Amazon Corretto), sem *flags* explícitas de *heap*; os tamanhos inicial e máximo foram definidos pela ergonomia da JVM, sendo o máximo aproximadamente 1/4 da memória da instância, com o coletor G1 padrão. No MVC, o Tomcat usou `accept-count=5000`, `max-connections=10000` e `threads.min-spare=20`, com o HikariCP fixado em 1500 conexões. No WebFlux, o Reactor Netty operou na configuração padrão, com o *pool* R2DBC também fixado em 1500 conexões. Todos os parâmetros estão versionados no repositório de artefatos.

3.3 Execução e Medição

A carga foi gerada pela ferramenta k6 em dois estágios. No primeiro, submeteu-se uma carga de *warm-up* de 60 segundos com 70 usuários virtuais (VUs). No segundo, uma carga em regime permanente (*steady*) de 1500 VUs por 40 segundos na qual as métricas de latência foram coletadas. A fase de *warm-up* foi adotada para que a JVM tenha tempo de compilar os métodos mais usados via JIT (compilação *just-in-time*) e estabilizar suas estruturas internas. Ao separarmos os cenários k6 dessa forma, evita-se coletar métricas desse período inicial de instabilidade, prática recomendada por Jain [4] para que apenas o comportamento estável (*steady*) seja considerado no experimento. O valor de 1500 VUs foi escolhido para representar um cenário de alta concorrência, acima dos 1000 VUs adotados como referência por de Deus et al. [3] em comparações de *frameworks* Java. Cada VU envia requisições sequenciais ao *endpoint* com `sleep(0.1)` entre chamadas, simulando um cliente HTTP de longa duração. Executamos cada cenário em 5 repetições (*reps*) independentes, descartando as 2 primeiras para excluir o transitório de aquecimento, resultando em 3 *reps* válidas por cenário. Métricas do processo Java (CPU, memória, *threads*, *heap* e GC) foram coletadas a cada segundo remotamente, por meio de SSH na máquina da aplicação.

3.4 Cenários e Variação do *Pool* de *Threads*

Como o objetivo do artigo é avaliar se a configuração do MVC altera sua vantagem em relação ao WebFlux sob alta concorrência, a variável manipulada principal foi o tamanho do *pool* de *threads* HTTP do Tomcat (`server.tomcat.threads.max`), parâmetro central do modelo *thread-per-request*. O WebFlux não possui um parâmetro diretamente equivalente ao tamanho desse *pool*, embora parâmetros internos do Reactor Netty possam ser configurados. Por isso, o WebFlux foi avaliado na configuração padrão, representando o comportamento *out-of-the-box* do *framework*, que se ajusta automaticamente ao hardware: o *event loop* passou de 4 para 8 *threads* entre as Fases A e B. Como a variável de interesse do estudo é o dimensionamento do *pool* do MVC, a exploração de configurações alternativas do Reactor Netty, como o número de *event loop threads* e o *backlog*, permanece fora do escopo e é reconhecida como limitação na Seção 5.

Em cada fase, foram testados sete cenários: seis configurações do MVC, com 250, 500, 750, 1000, 1250 e 1500 *threads*, e uma configuração padrão do WebFlux. Os níveis foram escolhidos para cobrir três comportamentos: *pool* subdimensionado para a carga de 1500 VUs (menor que o número de usuários virtuais), ajustado à carga e superdimensionado (maior ou igual à concorrência). A **Fase A** usou uma c7i.xlarge (4 vCPUs) como instância da aplicação, e a **Fase B** replicou os mesmos cenários em uma c7i.2xlarge (8 vCPUs) para analisar se dobrar o número de vCPUs compensa o custo do modelo *thread-per-request*.

3.5 Análise Estatística

Para cada cenário, os valores de P50, P95 e RPS foram calculados separadamente em cada repetição do experimento. A média desses valores entre as três *reps* e o respectivo intervalo de confiança de 95% foram obtidos pela distribuição t de Student com $n - 1$ graus de liberdade. Esse cálculo trata cada *rep* como uma observação independente, pois corresponde a uma execução completa do experimento. As ~190 mil requisições individuais dentro de uma mesma *rep* não foram usadas diretamente no IC, pois compartilham o mesmo estado da JVM, do banco e do ambiente de teste;

tratá-las como independentes produziria intervalos artificialmente estreitos [4].

Embora cada cenário conte com três *reps* válidas, o regime permanente apresentou baixa variabilidade na maioria dos casos, com ICs de P50 e P95 frequentemente inferiores a 2% da média. Cada *rep* compreende 100 s de execução, sendo 60 s de *warm-up* e 40 s de medição em regime permanente. Nos cenários com maior variabilidade, como MVC 1000t e 1500t na Fase A, os ICs mais largos são explicitamente reportados e os resultados interpretados com cautela. Ainda assim, reconhecemos na Seção 5 que um número maior de *reps*, ou o uso de *bootstrap*, aumentaria a robustez das estimativas.

Na comparação de cada cenário do MVC com o WebFlux, foi aplicado um procedimento em dois passos sobre os três valores de P95 obtidos nas *reps* de cada *framework*. Primeiro, o teste de Shapiro-Wilk [10], que verifica se cada conjunto é compatível com a distribuição normal ($\alpha = 0{,}05$). E a partir do resultado da normalidade, o segundo passo foi definir a escolha entre um teste paramétrico e um não-paramétrico. Quando ambos os conjuntos passam, aplica-se o teste t de Welch com a intensidade da diferença medida pelo d de Cohen; caso contrário, aplicamos o teste de Wilcoxon (Mann-Whitney) com a intensidade da diferença medida pelo r *rank-biserial*.

4 Resultados e Discussão

Os resultados são apresentados considerando o P95 como métrica principal de latência e o P50 como indicador do desempenho típico. O P50 corresponde à mediana da latência, enquanto o P95 indica o tempo em que 95% das requisições foram concluídas. O P95 é frequentemente adotado como referência para *Service Level Objectives* (SLOs) em engenharia de confiabilidade [1]. A vazão é expressa em RPS (*requests per second*), isto é, no número de requisições concluídas por segundo. A Tabela 1 reúne as métricas das duas fases, e a Figura 3 mostra como o P95 varia em função do tamanho do *pool* e da quantidade de CPU disponível.

Tabela 1: Latência (P50, P95), vazão (RPS) e uso de CPU consolidados das Fases A (4 vCPUs) e B (8 vCPUs). Médias entre três *reps* válidas com [IC 95%] pela distribuição t de Student. CPU expressa como % do total disponível na instância. Carga de 1500 VUs em todos os cenários.

Cenário	Fase	P50 (ms)	P95 (ms)	RPS	CPU
MVC 250t	A	511,2 (+0,0; -0,1)	520,3 (+1,2; -1,2)	2.447	26 %
MVC 250t	B	513,6 (+0,5; -0,4)	524,3 (+1,8; -1,7)	2.456	14 %
MVC 500t	A	205,5 (+0,2; -0,2)	271,6 (+4,5; -4,4)	4.878	51 %
MVC 500t	B	205,7 (+0,2; -0,2)	275,5 (+2,8; -2,9)	4.875	29 %
MVC 750t	A	115,5 (+4,7; -4,8)	225,6 (+8,5; -8,5)	6.210	66 %
MVC 750t	B	105,1 (+0,9; -1,0)	218,7 (+8,0; -8,1)	6.472	36 %
MVC 1000t	A	122,3 (+42,3; -42,2)	247,1 (+92,3; -92,3)	5.999	71 %
MVC 1000t	B	103,7 (+0,5; -0,5)	222,4 (+6,4; -6,5)	6.613	37 %
MVC 1250t	A	117,6 (+5,2; -5,1)	241,9 (+20,0; -20,1)	6.100	70 %
MVC 1250t	B	103,7 (+0,1; -0,0)	213,4 (+19,8; -19,9)	6.640	38 %
MVC 1500t	A	121,6 (+18,2; -18,3)	270,5 (+91,2; -91,2)	5.471	72 %
MVC 1500t	B	103,9 (+0,5; -0,5)	224,5 (+2,4; -2,4)	6.413	39 %
WebFlux	**A**	**147,5 (+1,8; -1,7)**	**257,0 (+8,1; -8,1)**	**5.674**	**86 %**
WebFlux	**B**	**105,9 (+0,4; -0,5)**	**222,2 (+4,1; -4,1)**	**6.604**	**48 %**

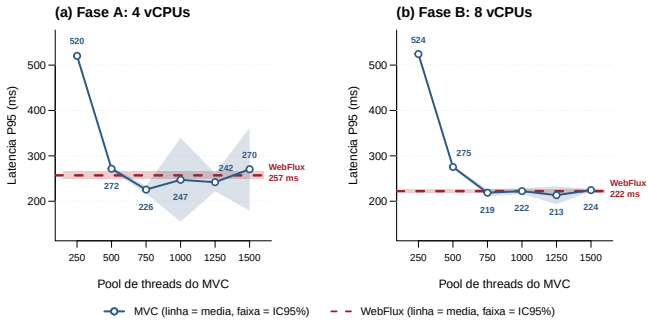

Figura 3: Latência P95 do MVC em função do tamanho do *pool* de *threads*, com IC 95% (faixa azul), comparada ao WebFlux (linha vermelha tracejada, com IC 95% em faixa). Painel (a) Fase A (4 vCPUs); painel (b) Fase B (8 vCPUs).

4.1 Fase A: Desempenho do MVC sob Diferentes Tamanhos de *Pool*

Nos cenários em que o *pool* está subdimensionado para as requisições concorrentes, a quantidade de *threads* limita as requisições que podem permanecer em processamento simultaneamente, o que é compatível com a Lei de Little [6]. Com 250 *threads* e tempo de espera do banco próximo de 100 ms, a vazão máxima esperada é cerca de $\lambda = L/W = 250/0{,}1 = 2.500$ rps, quase os 2.447 rps medidos. Já que a carga de VUs excede essa capacidade, requisições adicionais passam a aguardar, elevando a latência. Nesse cenário, o P50 do MVC com 250 threads chega a 511 ms, mais de 3× o P50 do WebFlux (147 ms), e o P95 atinge 520 ms, mais que o dobro do WebFlux (257 ms).

Com 750 *threads*, o P50 do MVC é 22% menor que o WebFlux, o P95 é 12% menor e a vazão é 9% maior, atingindo o melhor desempenho para o modelo imperativo. O MVC funciona bem porque dispõe de *threads* suficientes para acomodar as requisições enquanto elas esperam pela resposta do banco. O ponto exato em que isso acontece depende do tempo de resposta de cada requisição, valor que muda conforme a carga e o ambiente, sendo difícil de prever em produção sem medições prévias.

Uma hipótese para o menor P95 do MVC com 750 *threads* e, sem significância estatística, com 1250 *threads*, é que, havendo recursos suficientes, o modelo *thread-per-request* apresente menor sobrecarga por requisição que o *pipeline* reativo. Essa interpretação é consistente com a literatura [8, 14] e com o maior uso de CPU observado no WebFlux.

A partir de 1000 *threads*, ampliar o *pool* não produziu nova redução do P95 na Fase A em relação ao 750t. A latência ficou entre 242 ms e 271 ms, próxima ao valor observado no WebFlux (257 ms). Os ICs, porém, tornam-se mais largos do que em 750t, o que aponta maior variação entre *reps*, sem que isso represente uma queda consistente de desempenho. A vazão também não apresenta melhora clara com o aumento do *pool*, variando entre 5.471 rps e 6.100 rps. Esses resultados indicam que, após o *pool* atingir um tamanho suficiente para atender a concorrência efetiva da carga, novos aumentos deixam de trazer ganhos perceptíveis de latência ou vazão.

Na Fase A, a partir de 750 *threads*, a CPU permanece entre 66–72%, enquanto o P95 deixa de melhorar. Como a carga é dominada pela espera de E/S, ampliar o *pool* além desse ponto não reduz o tempo de serviço nem aumenta a vazão. A faixa de 242–271 ms entre 1000 e 1500 *threads*, acompanhada por ICs largos, sugere variabilidade entre *reps*, e não degradação sistemática.

4.2 Fase B: Efeito do Aumento de CPU sobre o MVC

Na Fase B, mesmo com maior disponibilidade de CPU, os casos com poucas *threads* no *pool* (MVC 250t e 500t) continuam apresentando limites semelhantes aos da Fase A. O MVC 250t, por exemplo, mantém P50 de 514 ms e RPS de 2.456. Isso indica que, nesses casos, o aumento de CPU não removeu a limitação causada pelo número de *threads* disponíveis.

Nos demais cenários do MVC, com *pool* a partir de 750 *threads*, os resultados também se aproximam do WebFlux em P50, P95 e RPS. A diferença máxima entre qualquer configuração MVC com *pool* $\geq$ 750 e o WebFlux é de 2 ms em P50 e 9 ms em P95. Os ICs de P95 também ficam mais estreitos nessa fase, o que indica menor variabilidade entre *reps* com mais CPU disponível.

A Tabela 2 descreve essas comparações por cenário. Na Fase B, somente MVC 250t e 500t apresentam diferenças significativas em relação ao WebFlux, confirmando a convergência observada acima. Na Fase A, o padrão é semelhante: MVC 250t e 500t são significativamente mais lentos ($p = 0{,}036$ e $p = 0{,}006$, respectivamente), enquanto a linha destacada de MVC 750t indica melhor desempenho ($d = +5{,}52$, $p < 0{,}001$); a partir de 1000 *threads*, as configurações já não se distinguem estatisticamente do WebFlux ($p > 0{,}05$).

Tabela 2: Comparação entre MVC e WebFlux sobre o P95 médio por *rep* nas duas fases. Efeito reportado como d para Welch e r para Wilcoxon; valores positivos indicam melhor desempenho do MVC (latência menor) em relação ao WebFlux e valores negativos indicam melhor desempenho do WebFlux.

Comparação	Fase A		Fase B	
	Efeito	p	Efeito	p
MVC 250t vs WebFlux	$r = -1{,}00$	0,036	$d = -236{,}5$	$< 0{,}001$
MVC 500t vs WebFlux	$d = -2{,}69$	0,006	$d = -37{,}3$	$< 0{,}001$
MVC 750t vs WebFlux	$d = +5{,}52$	$< 0{,}001$	$r = +0{,}56$	0,383
MVC 1000t vs WebFlux	$r = +0{,}33$	0,551	$d = -0{,}06$	0,945
MVC 1250t vs WebFlux	$d = +2{,}14$	0,057	$d = +1{,}54$	0,190
MVC 1500t vs WebFlux	$d = -0{,}62$	0,591	$d = -1{,}68$	0,125

Na Fase B, embora latência e vazão sejam semelhantes, o uso de CPU ainda diferencia os modelos. As configurações do MVC com *pool* a partir de 750 *threads* utilizam entre 36 % e 39 % da CPU disponível, enquanto o WebFlux utiliza 48 % para vazão semelhante, cerca de 30 % a mais. Esse resultado sugere maior sobrecarga do processamento reativo em cargas I/O-*bound*, em consonância com estudos que relatam custos adicionais do *pipeline* reativo e contenção no *event loop* próximo à saturação da CPU [8, 14]. Com CPU suficiente para evitar contenção relevante no MVC, essa diferença torna-se mais visível.

Como o `pg_sleep` executa no banco, a CPU da aplicação permanece majoritariamente ociosa durante a espera de E/S. Assim, o uso de CPU deve ser interpretado de forma relativa entre os modelos, sob vazão e *hardware* semelhantes, e não como medida absoluta de saturação, sujeita também ao escalonamento dinâmico da frequência do processador.

5 Considerações Finais

Este artigo investigou se a vantagem esperada do Spring WebFlux sobre o Spring MVC em cargas de E/S se mantém quando o MVC é configurado adequadamente. O desempenho do modelo *thread-per-request* mostrou-se dependente do *hardware* e da configuração do *pool*: com 4 vCPUs, o MVC perde para o WebFlux quando o *pool* é subdimensionado, mas alcança P95 cerca de 12% menor com 750 *threads*; com 8 vCPUs, os cenários do MVC com *pool* $\geq$ 750 convergem com o WebFlux em P50, P95 e *throughput*, enquanto o WebFlux utiliza cerca de 30 % mais CPU. Quando adequadamente dimensionados, ambos sustentam carga semelhante, mas com custos distintos: o MVC emprega mais *threads* e pode demandar mais memória, enquanto o WebFlux apresentou maior uso de CPU neste experimento. Como cada *thread* reserva uma pilha própria (Seção 2), esse custo tende a crescer com o tamanho do *pool*, embora não tenha sido quantificado neste estudo. A análise detalhada de memória fica como trabalho futuro.

Os resultados sugerem que, em determinados cenários, o ajuste do *pool* de *threads* e o escalonamento vertical podem oferecer ganhos semelhantes aos do WebFlux, sem demandar a migração para um novo paradigma. A decisão de migrar do MVC para o WebFlux em sistemas Java consolidados, portanto, não deve ser imediata.

Este estudo limita-se a uma carga I/O-*bound* sintética, com latência fixa via `pg_sleep` em PostgreSQL, sem cargas CPU-*bound*, mistas ou outras fontes de E/S. O WebFlux foi avaliado apenas na configuração padrão, e os 1500 VUs permanecem abaixo de escalas como o C10K. Assim, os resultados não demonstram escalabilidade extrema, mas indicam que a vantagem do WebFlux não é universal quando o MVC é bem configurado. O cenário foi deliberadamente simplificado para isolar o modelo de execução, e não para reproduzir integralmente uma aplicação de produção.

As *Virtual Threads* do Java 21 (JEP 444 [7]) constituem alternativa relevante ao WebFlux por preservarem o estilo imperativo com *threads* leves. Não foram incluídas por delimitação de escopo, embora já sejam avaliadas em estudos recentes [3]. Como trabalhos futuros, pretende-se avaliar *Virtual Threads*, cargas CPU-*bound* e mistas e escalonamento horizontal.

Disponibilidade de Artefatos

Os *scripts* da análise, as aplicações e os dados brutos das execuções estão publicamente disponíveis em repositório anônimo em: https://github.com/beatrizfteixeira/mvc-vs-webflux-sblp2026.

Uso de Inteligência Artificial

A IA auxiliou na elaboração de *scripts* em R com `ggplot2` para produzir os gráficos de latência P95 e intervalos de confiança. As escolhas de visualização, interpretações e conclusões foram avaliadas e revisadas pelos autores.

Referências

[1] Betsy Beyer, Chris Jones, Jennifer Petoff, and Niall Richard Murphy. 2016. *Site Reliability Engineering: How Google Runs Production Systems.* O'Reilly Media.

[2] Jonas Bonér, Dave Farley, Roland Kuhn, and Martin Thompson. 2014. The Reactive Manifesto, version 2.0. Disponível em: <https://www.reactivemanifesto.org/>. Acesso em: 24 maio 2026.

[3] Ítalo Martins de Deus, Rubens de Souza Matos Junior, George Leite Júnior, and Gustavo da Silva Quirino. 2025. Avaliação de Desempenho de Frameworks Web Java entre Reatividade e Threads Virtuais: Um Estudo Comparativo entre Spring e Quarkus. In *XXIV Workshop em Desempenho de Sistemas Computacionais e de Comunicação (WPerformance)*. 61–72.

[4] Raj Jain. 1991. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling.* John Wiley & Sons.

[5] JetBrains. 2025. The State of Java 2025: Insights from the JetBrains Developer Ecosystem Survey 2025. Disponível em: <https://lp.jetbrains.com/the-state-of-java-2025>. Acesso em: 22 maio 2026.

[6] John D. C. Little. 1961. A Proof for the Queuing Formula: $L = \lambda W$. *Operations Research* 9, 3 (1961), 383–387.

[7] Oracle Corporation. 2023. JEP 444: Virtual Threads. Disponível em: <https://openjdk.org/jeps/444>. Acesso em: 20 maio 2026.

[8] Julien Ponge, Arthur Navarro, Clément Escoffier, and Frédéric Le Mouël. 2021. Analysing the Performance and Costs of Reactive Programming Libraries in Java. In *8th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems (REBLS)*. 51–60.

[9] Project Reactor Team. [n. d.]. Reactor Netty Reference Documentation. Disponível em: <https://projectreactor.io/docs/netty/release/reference/>. Acesso em: 20 maio 2026.

[10] S. S. Shapiro and M. B. Wilk. 1965. An Analysis of Variance Test for Normality (Complete Samples). *Biometrika* 52, 3/4 (1965), 591–611.

[11] Assemgul Sukhambekova. 2025. Comparison of Spring WebFlux and Spring MVC. *Modern Scientific Method* 9 (2025).

[12] Craig Walls. 2022. *Spring in Action* (6th ed.). Manning Publications, Shelter Island, NY.

[13] Matt Welsh, David Culler, and Eric Brewer. 2001. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In *18th ACM Symposium on Operating Systems Principles (SOSP)*. 230–243.

[14] Andrei Zbarcea, Cătălin Tudose, and Alexandru Boicea. 2026. Extending the Migration from Asynchronous to Reactive Programming in Java: A Performance Analysis of Caching, CPU-Bound, and Blocking Scenarios. *Applied Sciences* 16, 1 (2026), 90.