INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA DO PIAUÍ
CAMPUS TERESINA CENTRAL
ANÁLISE E DESENVOLVIMENTO DE SISTEMAS

LUIS HENRIQUE DE MOURA SANTOS

ESCALABILIDADE EM SISTEMAS WEB: uma análise comparativa entre Java e Rust em arquitetura monolítica

TERESINA, PIAUÍ
2026

LUIS HENRIQUE DE MOURA SANTOS

ESCALABILIDADE EM SISTEMAS WEB: uma análise comparativa entre Java e Rust em arquitetura monolítica

Trabalho de Conclusão de Curso (Artigo Científico) apresentado como exigência parcial para obtenção do diploma do Curso de Análise e Desenvolvimento de Sistemas do Instituto Federal de Educação, Ciência e Tecnologia do Piauí, Campus Teresina.

Orientador: Prof. M^{e} Aislan Rafael Rodrigues de Sousa

TERESINA, PIAUÍ
2026

ESCALABILIDADE EM SISTEMAS WEB: uma análise comparativa entre Java e Rust em arquitetura monolítica

Luis Henrique de Moura Santos[1]
Aislan Rafael Rodrigues de Sousa[2]

RESUMO

A expansão das aplicações web intensificou a busca por escalabilidade, e a arquitetura monolítica permanece como ponto de partida para muitos sistemas. Diante do aumento da carga, presume-se que a migração para arquiteturas distribuídas seja inevitável, ignorando-se uma variável anterior: o ecossistema de execução da linguagem, que compreende o modelo de gerenciamento de memória, o framework e o modelo de concorrência. Este trabalho compara duas implementações funcionalmente equivalentes de um serviço de encurtamento de endereços web, uma em Java com Spring Boot e servidor embarcado Tomcat, e outra em Rust com Actix-web sobre runtime assíncrono Tokio, submetidas a cargas progressivas de até 1.500 requisições por segundo em containers limitados a meio núcleo de processamento e 512 megabytes de memória. Três algoritmos de coleta de lixo da máquina virtual Java foram avaliados como variável independente. Os resultados mostram que a implementação em Rust manteve latência mediana estável entre 260 e 272 microssegundos com consumo fixo de 7 megabytes, enquanto a configuração Java padrão registrou picos de latência no percentil 95 de até 196 milissegundos e consumo de aproximadamente 420 megabytes. O coletor concorrente de última geração, projetado para latência mínima, falhou nos critérios de aceitação nas três execuções devido à contenção por processamento fracionado. O coletor sequencial, o mais simples disponível, superou as demais configurações Java em latência, variância e consumo de memória. No cenário investigado, a escolha do ecossistema de execução e do algoritmo de coleta mostrou-se fator determinante nos limites de desempenho do monólito antes de migração arquitetural.

Palavras-chave: escalabilidade; arquitetura monolítica; gerenciamento de memória; coleta de lixo; desempenho sob carga.

ABSTRACT

The expansion of web applications has intensified the pursuit of scalability, and monolithic architecture remains the starting point for many systems. As load increases, migration toward distributed architectures is often assumed to be inevitable, overlooking an earlier variable: the execution ecosystem of the language, encompassing its memory management model, framework, and concurrency model. This work compares two functionally equivalent implementations of a web address shortening service, one in Java with Spring Boot and embedded Tomcat server, and another in Rust with Actix-web on the asynchronous Tokio runtime, subjected to progressive

[1] Graduando(a) em Análise e Desenvolvimento de Sistemas pelo Instituto Federal do Piauí (IFPI). E-mail: luishenriquemourasantos1@gmail.com.
[2] Mestre em Engenharia de Software pelo CESAR.edu. Professor do Instituto Federal do Piauí (IFPI), Campus Teresina Central. E-mail: aislanrafael@ifpi.edu.br.

loads of up to 1,500 requests per second in containers limited to half a processing core and 512 megabytes of memory. Three garbage collection algorithms of the Java virtual machine were evaluated as an independent variable. Results show that the Rust implementation maintained a stable median latency between 260 and 272 microseconds with a fixed footprint of 7 megabytes, while the default Java configuration exhibited 95th-percentile latency peaks of up to 196 milliseconds and approximately 420 megabytes of memory consumption. The latest-generation concurrent collector, designed for minimal latency, failed the acceptance thresholds in all three runs due to contention over the fractional processing capacity. The sequential collector, the simplest available, outperformed the other Java configurations in latency, variance, and memory consumption. Within the investigated scenario, the choice of execution ecosystem and collection algorithm proved to be a determining factor in the performance limits of the monolith before architectural migration.

Keywords: scalability; monolithic architecture; memory management; garbage collection; performance under load.

Data de Aprovação: 23/06/2026

1 INTRODUÇÃO

No período pós-pandemia, a expansão acelerada dos sistemas *web* elevou substancialmente a demanda por escalabilidade e disponibilidade das aplicações. O fenômeno é evidenciado pelo setor de *e-commerce* brasileiro, cujo faturamento atingiu R$ 204,3 bilhões e mais de 414 milhões de pedidos (ABCOMM, 2024), avanço catalisado pela COVID-19 e que exigiu das empresas o aprimoramento de suas infraestruturas digitais (CARREIRO; NOSE, 2023).

A escalabilidade, contudo, não possui solução universal e demanda análise rigorosa dos recursos necessários ao atendimento de múltiplos usuários simultâneos (GREGOL; SCHUTZ, 2013). A arquitetura monolítica costuma ser a opção inicial por seu menor custo e complexidade, embora tenda a perder manutenibilidade conforme o sistema cresce (MENDES, 2021). Modelos distribuídos oferecem escalabilidade independente por componente, mas impõem uma complexidade de infraestrutura nem sempre justificável (CARVALHO, 2022), o que torna pertinente investigar o quanto otimizações internas ao monólito podem ampliar seu desempenho antes de qualquer migração arquitetural.

Uma variável frequentemente subestimada nesse contexto é a linguagem de programação. Embora o *Java* seja consolidado em sistemas corporativos por sua robustez, sua *performance* sob alta carga pode ser afetada pela dependência do *garbage collector* (REZZAGHI; SILVA, 2023). Linguagens como o *Rust* adotam o modelo de *ownership*, eliminando a coleta de lixo em tempo de execução e demonstrando, nos testes dos mesmos autores, desempenho superior sob cargas elevadas. Tais evidências indicam

que a eficiência de uma linguagem compilada nativamente pode mitigar limitações inerentes ao modelo monolítico, postergando ou evitando a adoção de arquiteturas mais complexas.

A maioria dos estudos disponíveis concentra-se em comparações de desempenho em nível unitário de código, sem observar o comportamento de um sistema *web* completo sob condições reais de estresse. Para preencher essa lacuna, este trabalho realiza um estudo comparativo utilizando um serviço de encurtamento de URLs como objeto experimental, escolhido por sua natureza intensiva em operações de entrada e saída (*I/O*) e pelo requisito de baixa latência. Ao submeter duas implementações monolíticas, uma em *Java* com *Spring Boot* e outra em *Rust* com *Actix-web*, a cargas progressivas de até 1.500 requisições por segundo, o estudo fornece dados quantitativos que auxiliam a decisão sobre o momento adequado para a adoção de novas tecnologias como estratégia de escalabilidade.

Nesse contexto, este trabalho investiga como a escolha do ecossistema de execução, compreendendo linguagem, framework web, modelo de concorrência e estratégia de gerenciamento de memória, influencia o desempenho sob carga de sistemas web monolíticos, comparando implementações equivalentes em Java e Rust de um serviço de encurtamento de URLs submetidas a condições controladas de alta carga.

2 DESENVOLVIMENTO

2.1. ESCALABILIDADE E A ARQUITETURA MONOLÍTICA

A escalabilidade pode ser definida como a capacidade de um sistema de continuar operando de maneira eficiente à medida que a carga de trabalho aumenta, devendo ser compreendida não como propriedade binária, mas como um conjunto de estratégias para lidar com esse crescimento (KLEPPMANN, 2017).

A arquitetura monolítica é o padrão tradicional no qual todos os componentes funcionais, lógica de negócio e acesso a dados, são integrados e implantados como uma *single deployment unit*, empacotada em um único executável, como um *.jar* no ecossistema *Java* ou um binário compilado em *Rust* (RICHARDSON, 2018). Como todas as chamadas ocorrem via *stack* da linguagem e memória compartilhada, elimina-se o *overhead* de rede e a serialização de dados, resultando em latência interna mínima e maior simplicidade de teste e monitoramento (NEWMAN, 2015).

O crescimento desordenado de um monólito, contudo, pode levar ao padrão *Big Ball of Mud*, no qual o aumento da base de código dificulta o entendimento sistêmico, eleva o tempo de compilação e impede a escalabilidade granular, já que escalar uma única funcionalidade exige replicar toda a aplicação (FOWLER, 2015). Antes da decomposição em serviços que traz a escalabilidade granular ao custo de maior complexidade opera-

cional (NEWMAN, 2015), uma alternativa menos explorada é investigar até que ponto a escolha do ecossistema de execução, linguagem, runtime e framework, pode ampliar os limites do monólito em cenários específicos de restrição de recursos, eixo sobre o qual este trabalho se debruça.

2.2. MODELOS DE EXECUÇÃO: JVM E *OWNERSHIP*

2.2.1. *Java* e a *JVM*

A *Java Virtual Machine* (*JVM*) utiliza compilação *Just-In-Time* (*JIT*): interpreta o código e compila progressivamente os caminhos frequentes (*hot paths*) para código nativo otimizado, de modo que o desempenho máximo só é atingido após um período de aquecimento (*warm-up*). O gerenciamento de memória é feito pelo *Garbage Collector* (*GC*), que remove objetos não referenciados do *heap*; embora elimine uma classe de erros, introduz não-determinismo temporal, pois pausas *Stop-The-World* suspendem todas as *threads* e elevam a latência de cauda sob alta concorrência (GOETZ *et al.*, 2006).

A escolha do coletor implica um *trade-off* entre o custo das pausas *Stop-The-World* e o consumo de CPU pelas *threads* de coleta concorrente (Oracle, 2021). Este trabalho avalia três coletores disponíveis no *Java* 21: o *G1GC* (padrão), incremental e regionalizado; o *ZGC* Generacional, concorrente e voltado a pausas inferiores a um milissegundo; e o *SerialGC*, *single-threaded* e historicamente associado a ambientes de recursos restritos.

2.2.2. *Rust* e o modelo de *Ownership*

O *Rust* dispensa o *Garbage Collector* por meio de um sistema de *ownership* (posse) e *borrowing* (empréstimo) verificado pelo compilador: cada valor possui um único proprietário, e a memória é liberada quando este sai de escopo, sem coleta em tempo de execução (KLABNIK; NICHOLS, 2018). O resultado é comportamento determinístico, pois a desalocação ocorre de forma previsível no próprio fluxo de execução, sem pausas. A concorrência assíncrona é viabilizada por *runtimes* como o *Tokio*, base do *Actix-web*, que implementa um *event loop* de escalonamento cooperativo: ao aguardar uma operação de I/O, a *thread* é liberada para outras tarefas sem bloquear nem manter estado de pilha por conexão contrastando com o modelo de *pool* de *threads* do *Tomcat*, no qual cada requisição ocupa uma *thread* durante a espera. Por compilar para código nativo, o *Rust* não requer *warm-up*, o que é particularmente relevante em cenários de *cold start*. A contrapartida é a curva de aprendizado, já que o *Java* mantém vantagem de produtividade na maioria dos contextos corporativos pela maturidade do ecossistema (REZZAGHI; SILVA, 2023).

2.3. IMPLEMENTAÇÃO DO ENCURTADOR DE URLS

Foram desenvolvidas duas versões funcionalmente equivalentes do encurtador, com o mesmo comportamento: um *endpoint* `POST /shorten`, que retorna HTTP 201 com o *link* encurtado, e um *endpoint* `GET /{code}`, que retorna 302 com o cabeçalho `Location` ou 404 caso o código não exista. A geração do `short_code` utiliza alfabeto BASE62 com 7 caracteres fixos e, em caso de colisão, repete a operação por até 3 tentativas. Nenhuma versão utiliza *cache*: todas as operações trafegam diretamente pelo *pool* de conexões ao PostgreSQL.

2.3.1. Monólito *Java*

Construído com *Spring Boot* e *Spring MVC* sobre servidor embarcado *Tomcat*, com acesso a dados via `JdbcTemplate` e SQL puro, sem ORM, executando sobre o Eclipse Temurin 21. O *pool* de conexões *HikariCP* foi equalizado à implementação *Rust* em 10 conexões para garantir isonomia, e o *heap* foi limitado a aproximadamente 256 MB dentro do *container* de 512 MB. O algoritmo de coleta foi variado via `JAVA_TOOL_OPTIONS`, sem reconstrução da imagem, entre *G1GC* (padrão), *ZGC* Generacional e *SerialGC*, isolando o coletor como única variável independente entre as execuções. A geração de números aleatórios adotou `ThreadLocalRandom` sem contenção de *lock*.

2.3.2. Monólito *Rust*

Construído com *Actix-web* sobre o *runtime Tokio*, com acesso a dados via *sqlx*, SQL puro e *prepared statements*, compilado estaticamente para um binário nativo. O *pool* do *sqlx* espelha o *HikariCP* em 10 conexões, e a geração de aleatórios usa `rand::thread_rng()`, também sem contenção de *lock*. O número de *workers* HTTP foi fixado em 1 via `.workers(1)`. O padrão do *framework* criava um *worker* por *thread* lógica do *host*, gerando contenção severa de *context switching* com apenas 0,5 CPU — desnecessária, pois o modelo assíncrono do *Tokio* processa toda a concorrência em um único *event loop*.

2.3.3. Camada de Dados e Persistência

Ambas as implementações compartilham a mesma imagem (`postgres:16-alpine`) e o mesmo esquema relacional, composto pela tabela de URLs e por um índice único sobre a coluna `short_code`:

```
CREATE TABLE IF NOT EXISTS urls (
    id           BIGSERIAL   PRIMARY KEY,
    short_code   VARCHAR(10) NOT NULL,
    original_url TEXT        NOT NULL,
```

```
    created_at   TIMESTAMP   NOT NULL DEFAULT NOW()
);
CREATE UNIQUE INDEX IF NOT EXISTS idx_urls_short_code
    ON urls (short_code);
```

O índice único é o mecanismo de detecção de colisão: a violação de restrição (`DataIntegrityViolationException` no *Java* e SQLSTATE `23505` no *Rust*) dispara a lógica de nova tentativa. O esquema é criado automaticamente na inicialização, via `spring.sql.init` no *Java* e `run_migrations()` no *Rust*.

3 PROCEDIMENTOS METODOLÓGICOS

Este capítulo descreve o ambiente de execução, o protocolo experimental e os critérios de coleta adotados para garantir a validade interna e a replicabilidade dos resultados. O protocolo foi refinado ao longo de quatro versões iterativas, cada uma corrigindo fontes de variância identificadas na anterior.

3.1. AMBIENTE DE TESTES

Os testes foram conduzidos em uma máquina física dedicada. Todas as aplicações foram containerizadas via *Docker Engine*, garantindo isolamento de processo, rede e sistema de arquivos entre as instâncias comparadas. A Tabela 1 especifica o ambiente utilizado.

Tabela 1 – Especificação do ambiente de testes

Componente	**Especificação**
CPU	Intel Core i5-13420H (8 núcleos, 12 *threads*, 4,60 GHz)
RAM	16 GB DDR5 4800 MHz
Armazenamento	SSD 512 GB M.2 PCIe Gen4
Sistema Operacional	Ubuntu 24.04.2 LTS
Containerização	Docker Engine 29.5.2
Ferramenta de carga	k6 via `docker run grafana/k6`

Fonte: Elaborado pelo autor (2026).

Na versão final do protocolo, limites explícitos de recursos foram aplicados via `deploy.resources` no *Docker Compose*, tanto ao *container* da aplicação quanto ao do banco de dados: 0,5 núcleos de CPU e 512 MB de RAM, com reservas de 0,25 núcleos e 256 MB. Com isso, nenhum serviço pôde consumir recursos do *host* além da fração alocada. Ao *container* do PostgreSQL foram ainda aplicados parâmetros de ajuste destinados a reduzir a variância extrínseca causada por operações internas de manutenção:

```
postgres -c autovacuum_vacuum_scale_factor=0.5
         -c autovacuum_analyze_scale_factor=0.5
         -c checkpoint_timeout=30min
         -c checkpoint_completion_target=0.9
```

Os dois primeiros parâmetros reduzem a frequência de ativação do *autovacuum* durante o teste, tornando seu comportamento mais previsível. Os dois últimos ampliam o intervalo de *checkpoint* para 30 minutos e distribuem a escrita ao longo de 90% desse intervalo, suavizando o impacto de I/O e evitando que essa operação coincida com o pico de carga do teste.

3.2. PROTOCOLO EXPERIMENTAL

O protocolo foi refinado iterativamente ao longo de quatro versões principais (V1 a V4), com cada versão corrigindo fontes de variância identificadas na anterior.

3.2.1. Evolução Iterativa do Protocolo

A Tabela 2 resume as versões do protocolo, os problemas identificados e as correções aplicadas. A versão V4a estabeleceu o protocolo base definitivo, eliminando todas as fontes de variância extrínseca identificadas; as versões V4b e V4c mantiveram exatamente as mesmas condições, variando apenas o algoritmo de *garbage collector* da JVM, *ZGC* Generacional e *SerialGC*, respectivamente.

Tabela 2 – Evolução do protocolo experimental V1–V4c

Versão	Problema identificado	Correção aplicada	Variância
V1	*Baseline*. Rust com 12 *workers* para 0,5 CPU; Java com `SecureRandom`; *pool* Rust ilimitado; k6 com erros de *endpoint* e campo JSON	—	—
V2	Comparação injusta: *pool* assimétrico, sem limites Docker iguais, ausência de *retry* em colisão	*Pool* Rust equalizado (max=10); limites Docker iguais; *retry* (3×); k6 corrigido	Extrínseca
V3	RNG assimétrico; sem *warm-up* JVM; pré-aquecimento de *pool* assimétrico	Java: `ThreadLocalRandom`; Rust: `min_connections(5)`; k6: 60 s @ 100 req/s	Extrínseca
V4a	PG sem limites; *warm-up* insuficiente para JIT C2; HikariCP *timeout* 30 s (cascata); Rust com 12 *workers*	PG com limites + *tuning*; 120 s @ 300 req/s; HikariCP 5 s; `.workers(1)`; G1GC explícito	Extrínseca
V4b	Isolar impacto do GC	ZGC Generacional	Intrínseca
V4c	Isolar impacto do GC	SerialGC	Intrínseca

Fonte: Elaborado pelo autor (2026).

3.2.2. Perfil de Carga e Execução (k6)

Os testes de carga foram executados com a ferramenta k6, via container `grafana/k6`, a partir de um script com dois cenários: uma fase de *warm-up*, não contabilizada nas métricas, e a fase de teste principal. A fase de *warm-up* opera por 120 segundos a 300 requisições por segundo. Esse volume garante que o compilador JIT do HotSpot atinja o nível de otimização C2, cujo limiar padrão é de 10.000 invocações, antes do

início da medição, assegurando isonomia frente à implementação *Rust*, que não requer aquecimento. A fase de teste segue o perfil da Tabela 3.

Tabela 3 – Perfil de carga da fase de teste (k6)

Etapa	**Duração**	**Taxa**	**VUs pré-alocados**
Ramp-up	3 min	10 $\rightarrow$ 1.500 req/s	500 (máx. 3.000)
Steady-state	5 min	1.500 req/s	—
Ramp-down	1 min	1.500 $\rightarrow$ 0 req/s	—
Duração total por execução: aproximadamente 11 minutos.			

Fonte: Elaborado pelo autor (2026).

Os *thresholds* de aceitação, aplicados exclusivamente sobre métricas da fase de teste, foram: $p_{50} < 200$ ms; $p_{95} < 500$ ms; $p_{99} < 1.000$ ms; e taxa de requisições com falha inferior a 5%. Cada configuração foi executada duas vezes de forma consecutiva, sendo a V4b (ZGC) executada três vezes, em razão da severidade e consistência do resultado negativo. Antes de cada execução, os *containers* foram derrubados com `docker compose down -v`, limpando o volume do banco de dados e garantindo estado inicial idêntico.

3.3. COLETA DE DADOS E MÉTRICAS

As métricas foram coletadas de duas fontes em paralelo. O k6 gerou, ao final de cada execução, os percentis de latência (p_{50}, p_{95} e p_{99}), *throughput*, taxa de erros (`http_req_failed`), número máximo de usuários virtuais ativos e iterações descartadas. O consumo de recursos dos *containers* foi monitorado via `docker stats`, com atualização a cada 5 segundos. O conjunto completo de métricas, *scripts* de teste e configurações de cada versão do protocolo está disponível no repositório público do projeto, atendendo ao objetivo de replicabilidade integral do experimento.

4 RESULTADOS E DISCUSSÃO

Este capítulo apresenta e interpreta os resultados das execuções da versão final do protocolo experimental (V4), à luz do referencial teórico do Capítulo 2. Foram avaliadas quatro configurações: o monólito *Rust* e o monólito *Java* nas variantes G1GC (V4a), ZGC Generacional (V4b) e SerialGC (V4c). As faixas reportadas na Tabela 4 refletem a variância entre execuções idênticas. Todas as métricas referem-se exclusivamente à fase de teste, excluindo o período de *warm-up*.

Tabela 4 – Síntese comparativa dos resultados por configuração (V4)

Configuração	p_{50}	p_{95}	p_{99}	**RAM app**	**Var.** p_{95}	**Thresholds**
Rust V4	260–272 µs	0,5–4,8 ms	28–277 ms	**7 MB**	$\sim$9,6$\times$	Sim
Java G1GC (V4a)	318–332 µs	30–196 ms	299–582 ms	$\sim$420 MB	$\sim$6,5$\times$	Sim
Java ZGC (V4b)	1,20–1,30 s	$\sim$1,59 s	1,79 s	453–462 MB	$\sim$1,0$\times$	**FALHOU**
Java SerialGC (V4c)	290–314 µs	0,5–1,55 ms	205–400 ms	$\sim$305 MB	$\sim$2,9$\times$	Sim

Fonte: Elaborado pelo autor (2026).

O *Rust* manteve p_{50} estável em 260–272 µs com consumo fixo de 7 MB, evidenciando o comportamento determinístico do modelo de *ownership* na ausência de *garbage collector*. O ZGC Generacional produziu o pior resultado, falhando nos *thresholds* nas três execuções, enquanto o SerialGC foi a melhor configuração *Java*, superando o G1GC em latência, variância e consumo de memória. Quanto aos gargalos, observaram-se dois padrões distintos: no *Rust*, o PostgreSQL saturou primeiro (50–51% do limite de CPU) enquanto a aplicação operou entre 36 e 44%; nas três variantes *Java*, a aplicação foi o componente dominante (50–56%), com o banco permanecendo entre 28 e 38%.

4.1. VARIÂNCIA EXTRÍNSECA E INTRÍNSECA

A versão V3 do protocolo registrou variância de aproximadamente 550$\times$ no p_{95} do *Java* entre execuções idênticas; após as correções descritas no Capítulo 3, a V4a registrou 6,5$\times$. A queda confirma que as correções eliminaram as principais fontes de variância *extrínseca*, de modo que os valores remanescentes caracterizam os ecossistemas, e não o ambiente de testes. A variância residual de 6,5$\times$ no G1GC é intrínseca ao coletor sob 0,5 CPU e $\sim$256 MB de *heap*, ao passo que os 9,6$\times$ do *Rust* são atribuíveis ao comportamento do PostgreSQL próximo ao limite de CPU, indicativo de *throttling* por *cgroups*, uma fonte de variância mais previsível, que substituiu as fontes extrínsecas eliminadas na V4.

4.2. MODELO DE EXECUÇÃO COMO FATOR DETERMINANTE

A comparação entre *Rust* e *Java* com G1GC, sob condições idênticas de ambiente, expõe o impacto direto do modelo de gerenciamento de memória. Nos termos de Klabnik e Nichols (2018), o modelo de *ownership* libera a memória no próprio fluxo de execução, sem suspensão de *threads*, eliminando o não-determinismo das pausas de GC. No *Java*, o G1GC opera com um *heap* aquém de seu dimensionamento ideal, o que aumenta a frequência de coletas *Stop-The-World* e eleva a latência de cauda: o pico de p_{95} de 196 ms do G1GC representa cerca de 717$\times$ a mediana do *Rust*, aproximadamente três ordens de grandeza. O deslocamento do gargalo reforça essa diferença estrutural: sem *overhead* de GC e com um único *worker* gerenciando a

concorrência via *event loop*, o *Rust* consome menos CPU por requisição, empurrando a saturação para a camada de persistência antes de esgotar a própria aplicação.

A diferença de memória (7 MB contra $\sim$420 MB) tem implicação direta em *deployments* com múltiplas instâncias. Em um *host* com 4 GB de RAM, a pegada do *Rust* permitiria cerca de 571 instâncias, frente a aproximadamente 9 do *Java* com G1GC e 13 com SerialGC ($\sim$305 MB), uma razão de até 63$\times$ na densidade de implantação, com impacto direto no custo operacional em ambientes *cloud* com cobrança por recursos provisionados.

4.3. O PARADOXO DO ZGC EM AMBIENTE CPU-RESTRITO

O resultado da V4b é o achado mais contra-intuitivo do experimento: o ZGC Generacional, projetado para minimizar latência, produziu p_{50} de 1,2–1,3 s e falhou nos *thresholds* nas três execuções. O ZGC realiza a maior parte da coleta de forma concorrente, em paralelo com as *threads* da aplicação (Oracle, 2021), arquitetura que pressupõe núcleos de CPU independentes para as *threads* de coleta. Com apenas 0,5 núcleo disponível, as *threads* do GC e as *threads* HTTP do *Tomcat* competiram diretamente pelo processamento, degradando o *throughput* e elevando a latência. A baixíssima variância do ZGC ($\sim$1,0$\times$, frente a 6,5$\times$ do G1GC) confirma a interpretação: a contenção por CPU é uma condição permanente, não intermitente. O G1GC, por contraste, varia mais justamente porque suas pausas *Stop-The-World* ocorrem em momentos imprevisíveis, mas, entre elas, a aplicação processa requisições sem competir com o GC pela CPU. O achado é consistente com a limitação sinalizada pela Oracle (Oracle, 2021): o ZGC não é universalmente superior, e seu uso em *containers* com fração de núcleo pode inverter completamente a vantagem esperada.

4.4. SERIALGC COMO CONFIGURAÇÃO ÓTIMA EM CPU-RESTRITO

A hipótese de que um coletor *single-threaded* seria mais eficiente em ambiente restrito foi confirmada: o SerialGC superou o G1GC em variância (2,9$\times$ contra 6,5$\times$), em latência absoluta (p_{95} de 541 µs–1,55 ms contra 30–196 ms) e em memória ($\sim$305 MB contra $\sim$420 MB), além de superar o ZGC em todas as métricas. O mecanismo é o modelo binário do coletor: durante as pausas, a CPU executa exclusivamente a coleta; entre elas, todo o processamento é dedicado ao *Tomcat*, sem competição de *threads* de GC concorrentes. Como o SerialGC foi concebido para ambientes de recursos limitados (Oracle, 2021), seu perfil é compatível com o do experimento. O resultado evidencia que coletores mais modernos não produzem necessariamente melhor desempenho: a sofisticação do G1GC e do ZGC é otimizada para CPU e memória abundantes e, em *containers* de 0,5 núcleo, representa custo, não benefício. A escolha do GC deve, portanto, ser tratada como decisão dependente do ambiente

de *deployment*, em instâncias pequenas, como os tipos `t3.micro` (AWS) ou `f1.micro` (GCP), o SerialGC merece avaliação como ponto de partida antes de configurações mais complexas, ressalvando-se que essa recomendação se limita a cargas de trabalho predominantemente I/O-bound em containers com fração de CPU, cenário no qual o coletor foi testado neste estudo.

4.5. LIMITAÇÕES DO ESTUDO

Os resultados devem ser interpretados dentro do escopo do experimento. A coleta de picos de CPU foi feita via `docker stats` com intervalo de 5 segundos, método sujeito a imprecisão de amostragem; os valores de CPU devem ser lidos como ordens de grandeza, não medições de precisão. O experimento foi conduzido em uma única máquina física isolada, de modo que a variabilidade de substrato típica de produção (*hypervisor*, NUMA, vizinhança de *containers*) não foi abordada. O caso de estudo, um encurtador de URLs, é intensivo em I/O e simples em lógica de negócio; aplicações com maior processamento em memória podem responder de forma distinta ao GC, e o próprio SerialGC tende a degradar quando o *heap* mantém grandes volumes de objetos vivos. Por fim, a comparação restringiu-se a 0,5 CPU por *container*: o limiar a partir do qual o ZGC supera o SerialGC, ou no qual o gargalo do *Rust* migra definitivamente para o banco, não foi investigado e constitui direção natural para trabalhos futuros.

5 CONCLUSÃO E TRABALHOS FUTUROS

Este trabalho investigou como a escolha da linguagem de programação e o modelo de gerenciamento de memória influenciam a escalabilidade de sistemas *web* monolíticos, comparando implementações em *Java* e *Rust* submetidas a cargas progressivas em ambiente containerizado e controlado.

O modelo de execução do *Rust*, sem coletor de lixo e com concorrência assíncrona de baixo *overhead*, manteve latência mediana estável e consumo de memória constante em 7 MB ao longo de todas as execuções, deslocando o gargalo de CPU para o banco de dados antes de saturar a aplicação. O *Java* com G1GC, em contraste, exibiu maior consumo de memória ($\sim$420 MB) e picos expressivos na latência de cauda, reflexo do não-determinismo das pausas *Stop-The-World*. A investigação dos algoritmos de *garbage collector* produziu o achado mais contra-intuitivo do experimento: o ZGC Generacional, projetado para latência mínima, foi a configuração de pior desempenho, falhando nos critérios de aceitação nas três execuções, pois suas *threads* de coleta concorrente competiram pelo CPU fracionado. O SerialGC, o coletor mais simples da JVM, superou ambas as configurações mais sofisticadas em variância, latência e memória, demonstrando que a sofisticação do algoritmo pode tornar-se um custo, e

não um benefício, em *containers* com CPU restrito.

Esses achados oferecem duas orientações concretas. Para equipes que avaliam a adoção do *Rust* como alternativa de escalabilidade sem migração arquitetural, os dados indicam, no cenário investigado, ganhos expressivos em pegada de memória e previsibilidade de latência, com a contrapartida de maior exigência no desenvolvimento. Para aplicações *Java* em instâncias *cloud* com recursos reduzidos, a escolha do GC deve ser tratada como decisão de dimensionamento: em containers com CPU fracionado e cargas predominantemente I/O-bound, o SerialGC mostrou-se a configuração mais eficiente entre as avaliadas, merecendo consideração antes de coletores mais complexos

Como contribuição metodológica, o protocolo experimental iterativo ilustra uma abordagem para a separação entre variância extrínseca e intrínseca em estudos comparativos de desempenho. O código-fonte, os *scripts* de teste e as configurações de todas as versões estão disponíveis publicamente:

<https://github.com/Luis-Moura/Trabalho-de-Conclusao-de-Curso>

5.1. TRABALHOS FUTUROS

Os resultados abrem três direções naturais para pesquisas futuras. A primeira é estender o experimento a uma arquitetura distribuída em *Java*, microsserviços, prevista no escopo original do trabalho e removida por restrições de tempo, a fim de investigar se a distribuição horizontal compensa as desvantagens do GC sob carga e em que ponto sua complexidade operacional supera os ganhos de escalabilidade. A segunda é determinar o ponto de inversão dos coletores, variando o limite de CPU de forma controlada para traçar curvas de desempenho por algoritmo e identificar a partir de quantos núcleos o ZGC supera o SerialGC e em que patamar o G1GC se torna a escolha ótima. A terceira é replicar o experimento com casos de uso de maior complexidade computacional, lógica de negócio intensiva em memória, agregações ou transformações de dados, nos quais o comportamento do GC pode diferir significativamente do padrão observado em um serviço essencialmente de I/O.

REFERÊNCIAS

ABCOMM. **Principais Indicadores do e-Commerce**. 2024. Disponível em: <https://dados.abcomm.org/numeros-do-ecommerce-brasileiro>. Acesso em: 05 jan. 2026.

CARREIRO, A. A. S.; NOSE, E. T. O avanço do e-commerce brasileiro pré e pós pandemia. **RIT – Revista Inovação Tecnológica**, v. 13, n. 1, 2023. ISSN 2179-2895.

CARVALHO, M. D. d. Migração de sistemas erp monolíticos para a arquitetura de microsserviços. **Abakos**, Belo Horizonte, v. 1, n. 1, 2022. ISSN 2316-9451.

FOWLER, M. **MonolithFirst**. 2015. Disponível em: <https://martinfowler.com/bliki/MonolithFirst.html>. Acesso em: 04 fev. 2026.

GOETZ, B. *et al.* **Java Concurrency in Practice**. Boston: Addison-Wesley Professional, 2006.

GREGOL, R. E.; SCHUTZ, F. Recursos de escalabilidade e alta disponibilidade para aplicações web. **Revista Eletrônica Científica Inovação e Tecnologia**, p. 28–30, 2013. ISSN 2175-1846.

KLABNIK, S.; NICHOLS, C. **The Rust Programming Language**. San Francisco: No Starch Press, 2018.

KLEPPMANN, M. **Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems**. Sebastopol: O'Reilly Media, 2017.

MENDES, I. S. **Arquitetura monolítica vs microsserviços: uma análise comparativa**. Monografia (Trabalho de Conclusão de Curso (Graduação em Engenharia de Software)) — Universidade de Brasília, Brasília, DF, 2021.

NEWMAN, S. **Building Microservices: Designing Fine-Grained Systems**. Sebastopol: O'Reilly Media, 2015.

Oracle. **Available Collectors**. 2021. Disponível em: <https://docs.oracle.com/en/java/javase/21/gctuning/available-collectors.html>. Acesso em: 22 mai. 2026.

REZZAGHI, L. B.; SILVA, A. M. Programação concorrente em rust e java: uma análise comparativa de segurança, produção e performance. In: **Anais do WCF**. [S.l.: s.n.], 2023. v. 10.

RICHARDSON, C. **Microservices Patterns: With examples in Java**. Shelter Island: Manning Publications, 2018.