UNIVERSIDADE ESTADUAL PAULISTA
"JÚLIO DE MESQUITA FILHO"
Campus de São José do Rio Preto

DEPARTAMENTO DE CIÊNCIAS DE COMPUTAÇÃO E ESTATÍSTICA

ANÁLISE DE DESEMPENHO DE APIS RESTFUL PARA *E-COMMERCE*: COMPARAÇÃO ENTRE ACTIX, EXPRESS.JS E GIN

GABRIEL SCARANO DE OLIVEIRA
LUCAS FURRIEL RODRIGUES

São José do Rio Preto - SP, 2025

GABRIEL SCARANO DE OLIVEIRA
LUCAS FURRIEL RODRIGUES

ANÁLISE DE DESEMPENHO DE APIS RESTFUL PARA *E-COMMERCE*: COMPARAÇÃO ENTRE ACTIX, EXPRESS.JS E GIN

Monografia apresentada junto ao programa de **Bacharelado em Ciência da Computação** do **Instituto de Biociências, Letras e Ciências Exatas, UNESP Universidade Estadual Paulista**, como requisito à obtenção do título de **Bacharel em Ciência da Computação**.

Orientadora:

Prof.ª Dra. Adriana Barbosa Santos.

São José do Rio Preto - SP, 2025

Oliveira, Gabriel Scarano de.
Análise de desempenho de APIs RESTful para e-commerce : comparação entre Actix, Express.js e Gin / Gabriel Scarano de Oliveira, Lucas Furriel Rodrigues. -- São José do Rio Preto, 2025
91 f.: il., tabs.

Orientadora: Adriana Barbosa Santos
Trabalho de Conclusão de Curso (bacharelado – Ciência da Computação) – Universidade Estadual Paulista (Unesp), Instituto de Biociências, Letras e Ciências Exatas, São José do Rio Preto

1. Ciência da Computação – Estudo e ensino. 2. Plataformas digitais. 3. Comércio eletrônico. 4. Framework (Programa de computador) 5. Interface de programação de aplicativos (software de computador) I. Rodrigues, Lucas Furriel. II. Título.

CDU – 681.3

Ficha catalográfica elaborada pela Biblioteca do IBILCE
UNESP - Câmpus de São José do Rio Preto

GABRIEL SCARANO DE OLIVEIRA
LUCAS FURRIEL RODRIGUES

ANÁLISE DE DESEMPENHO DE APIS RESTFUL PARA *E-COMMERCE*: COMPARAÇÃO ENTRE ACTIX, EXPRESS.JS E GIN

Monografia apresentada ao programa de **Bacharelado em Ciência da Computação** do **Instituto de Biociências, Letras e Ciências Exatas, UNESP Universidade Estadual Paulista**, como requisito à obtenção do título de **Bacharel em Ciência da Computação**.

Banca Examinadora:

Prof.ª Dra. Adriana Barbosa Santos
UNESP - IBILCE
Orientadora

Prof. Dr. Adriano Mauro Cansian
UNESP - IBILCE

Prof. Dr. Aleardo Manacero Junior
UNESP - IBILCE

Dedicamos este trabalho a todos os que, direta ou indiretamente, contribuíram para nossa formação e crescimento.

AGRADECIMENTOS

Gabriel Scarano

Agradeço primeiramente aos meus pais, Vanessa Piereti Scarano Oliveira e Ricardo Rogério de Oliveira, e à minha irmã, Bianca Scarano de Oliveira, por todo o apoio incondicional, incentivo e amor dedicados durante toda a minha trajetória acadêmica. Sem vocês, esta conquista não seria possível. Obrigado por acreditarem em mim e por serem minha base e inspiração.

Ao João Gabriel Santiago de Souza, amigo que considero um irmão, agradeço pela amizade verdadeira, pelo companheirismo e por estar ao meu lado nos momentos de alegria e desafio. Sua presença foi fundamental nesta jornada.

Ao meu amigo Lucas Furriel Rodrigues, agradeço pela parceria, dedicação e colaboração durante todo o desenvolvimento deste trabalho. Compartilhar esta experiência com você tornou o processo mais enriquecedor e prazeroso.

À minha orientadora, Profa. Dra. Adriana Barbosa Santos, expresso minha profunda gratidão pela orientação dedicada e pelos ensinamentos valiosos ao longo deste trabalho.

Lucas Furriel

Agradeço aos meus pais, Cristiane Furriel Rodrigues e Renato Campiteli Rodrigues, que, mesmo com a distância, me deram total suporte durante minha graduação. Agradeço também à minha irmã, Mariana Furriel Rodrigues, e a toda a minha família.

Aos meus colegas e amigos que me fizeram companhia e me deram todo o apoio nessa jornada. Um agradecimento especial a Bianca Lançoni, Gabriel Scarano, Júlia Rodrigues, Pedro Cardana e Túlio Henry, que estiveram mais próximos de mim nesses últimos quatro anos.

À minha orientadora, Profa. Dra. Adriana Barbosa Santos, pela disponibilidade e pelos ensinamentos.

"I promise you, a thousand year journey, guided by compassion."

Miquella, *The Kind*, Elden Ring

RESUMO

O desempenho de APIs RESTful é fator crítico para o sucesso de plataformas de *e-commerce*, onde alta demanda e confiabilidade são requisitos essenciais. Este trabalho apresenta uma análise comparativa de desempenho entre três *frameworks web* para desenvolvimento de APIs: Express.js (JavaScript), Gin (Go) e Actix (Rust). Experimentos de simulação foram realizados com base em duas plataformas de *e-commerce* com características distintas (https://www.amazon.com e https://shopee.com.br), mais especificamente diferenciadas pelo volume de dados retornados. Quatro tipos de testes de performance foram realizados: carga, estresse, pico e ruptura. Os experimentos foram conduzidos em ambiente isolado com Docker, garantindo reprodutibilidade. Os resultados evidenciam que o Gin apresentou escalabilidade quase linear, mantendo confiabilidade próxima à perfeição, mesmo sob cargas extremas (7.000+ usuários virtuais simultâneos), com utilização de CPU consistentemente abaixo de 35%. O Actix revelou ter capacidade sólida até 2.400 usuários virtuais, mas com sensibilidade pronunciada a *payloads* volumosos. O Express.js apresentou limitações críticas, com taxas de falha superiores a 15% em testes de pico e subutilização crônica de CPU, evidenciando inadequação para cenários de alta demanda sem reengenharia arquitetural. Os resultados traz contribuições para auxiliar na escolha tecnológica diante da necessidade de escalar operações, manter qualidade de experiência do usuário e na avaliação da viabilidade comercial de plataformas digitais.

Palavras-chave: API; API RESTful; *e-commerce*; análise de desempenho.

ABSTRACT

The performance of RESTful APIs is a critical factor for the success of e-commerce platforms, where high demand and reliability are essential requirements. This work presents a comparative performance analysis of three web frameworks for API development: Express.js (JavaScript), Gin (Go), and Actix (Rust). Simulations of two distinct e-commerce platforms (https://www.amazon.com e https://shopee.com.br) were implemented, differing in the volume of returned data, and subjected to four performance test types: load, stress, spike, and breakpoint tests. The experiments were conducted in a isolated environment using Docker, ensuring reproducibility. The results showed that Gin achieved near-linear scalability, maintaining reliability close to perfection even under extreme loads (7,000+ concurrent virtual users), with CPU utilization consistently below 35%. Actix demonstrated solid capacity up to 2,400 virtual users but showed pronounced sensitivity to large payloads. Express.js revealed critical limitations, with failure rates above 15% in spike tests and chronic CPU underutilization, highlighting its unsuitability for high-demand scenarios without architectural reengineering.

Keywords: API; API RESTful; e-commerce; performance analysis.

LISTA DE FIGURAS

LISTA DE QUADROS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

API	*Application Programming Interface*
B2C	*Business to Consumer*
CORS	*Cross-Origin Resource Sharing*
CRUD	*Create, Read, Update, Delete*
HTTP	*Hypertext Transfer Protocol*
JSON	*JavaScript Object Notation*
npm	*Node Package Manager*
REST	*Representational State Transfer*
SLA	*Service-Level Agreement*
SKU	*Stock Keeping Unit*
URI	*Uniform Resource Identifier*
URL	*Uniform Resource Locator*
UUID	*Universally Unique Identifier*
VU	*Virtual User*
XML	*Extensible Markup Language*

SUMÁRIO

1 INTRODUÇÃO

Nos últimos anos, a crescente digitalização dos serviços e a popularização das compras online transformaram profundamente a maneira como consumidores e empresas interagem no ambiente virtual. O comércio eletrônico, ou *e-commerce*, emergiu como uma das principais formas de consumo, permitindo que empresas de diferentes portes alcancem um público global sem a necessidade de uma loja física. A conveniência e a acessibilidade proporcionadas por essa modalidade impulsionaram seu crescimento exponencial, tornando-o um setor estratégico para a economia digital. A pandemia de COVID-19 acelerou significativamente essa transformação: segundo dados do Compre&Confie, o *e-commerce* brasileiro registrou faturamento de R$ 9,4 bilhões em abril de 2020, representando crescimento de 81% em relação ao mesmo período do ano anterior (FERNANDES, 2020)

No contexto da tecnologia *web*, um dos principais facilitadores dessa evolução são as APIs (*Application Programming Interfaces*), que permitem a comunicação eficiente entre diferentes sistemas. Desenvolvidas para serem utilizadas por desenvolvedores e outras aplicações, as APIs operam nos bastidores, possibilitando integrações e trocas de informações de forma automatizada e padronizada. Em *e-commerce* B2C (*Business to Consumer*, vendas de empresa para consumidor final), sua importância se destaca ao viabilizar funcionalidades essenciais. Por exemplo, Almeida (2022) desenvolveu uma plataforma de *e-commerce* utilizando arquitetura de microserviços, com três APIs distintas para clientes, vendedores e transações, evidenciando a importância das APIs na integração e comunicação eficiente entre diferentes sistemas no ambiente de *e-commerce*.

Entre os diferentes tipos de APIs, as APIs RESTful ganharam ampla adoção devido à sua arquitetura simples, escalável e independente de plataforma. Uma API RESTful é uma interface de programação de aplicações baseada na arquitetura REST (*Representational State Transfer*), que utiliza princípios e restrições arquiteturais que promovem simplicidade, escalabilidade e independência entre cliente e servidor. Como descrito por Masse (2011), uma API REST é um tipo de servidor *web* que permite que um cliente possa acessar recursos de um determinado sistema, alterando dados ou utilizando de funções.

Essas APIs são amplamente utilizadas no contexto da *web* porque se integram naturalmente ao protocolo HTTP, a base da comunicação *online*, aproveitando métodos padrão como *GET*, *POST*, *PUT* e *DELETE* para realizar operações sobre recursos. Isso facilita a interação com navegadores e outras aplicações *web*. Além disso, sua arquitetura simples, baseada em padrões como URIs e formatos amplamente suportados, como JSON

e XML, torna o desenvolvimento e a integração mais acessíveis.

Neste contexto, o desenvolvimento de APIs eficientes e escaláveis é um desafio crucial no cenário atual do *e-commerce*. A escolha das linguagens de programação e *frameworks* adequados tem impacto direto no desempenho, escalabilidade e na experiência do usuário final. Frente à demanda crescente por sistemas rápidos e confiáveis para as transações B2C, identificar qual tecnologia é mais indicada para cada situação de aplicação é um aspecto essencial para garantir operações contínuas. Vale salientar, que em períodos de alta demanda, especialmente durante picos de tráfego na tradicional *Black Friday* e em outras datas promocionais, a eficiência das APIs se revela ainda mais significante.

Neste trabalho, a avaliação de fatores essenciais relativos a essas tecnologias é valorizada, de modo que proporcione um embasamento para facilitar a escolha mais adequada para esse tipo de desenvolvimento.

1.1 OBJETIVO

O objetivo principal deste estudo é realizar uma análise comparativa de desempenho entre *web frameworks* de diferentes linguagens de programação para o desenvolvimento de APIs REST. Mais especificamente:

- Simular três diferentes APIs, baseadas em duas plataformas de *e-commerce* existentes voltadas para o B2C;
- Analisar o desempenho dessas APIs desenvolvidas em *frameworks* distintos, considerando testes de carga, estresse, pico e ruptura.

1.2 JUSTIFICATIVA

Chen e Yang (2021), baseando-se em trabalhos relacionados, destaca que uma experiência de alta qualidade para o cliente é essencial para o sucesso do *e-commerce*. Ela não só impacta positivamente a intenção de compra inicial do cliente, mas também incentiva compras futuras. O desempenho de uma API tem papel fundamental para manter uma experiência de qualidade para o usuário final do *e-commerce*.

A escolha dos *frameworks* utilizados neste estudo considerou critérios como ampla adoção no mercado, desempenho reportado na literatura e solidez do ecossistema de

desenvolvimento. O Express.js, por exemplo, é amplamente documentado e utilizado, sendo abordado por Mardan (2014) como uma solução madura para aplicações em JavaScript. O Gin, escrito em Go, destaca-se pela eficiência e simplicidade na criação de APIs escaláveis, como apresentado por Sona et al. (2024). Já o Actix Web (frequentemente referido apenas como "Actix" em materiais técnicos), baseado em Rust, é reconhecido por seu alto desempenho e segurança, conforme demonstrado pelo TechEmpower (2025). A pesquisa anual do *Stack Overflow* de 2024 reforça a popularidade dessas tecnologias entre desenvolvedores.

Trabalhos anteriores, como o de Godinho et al. (2024), compararam o desempenho de tecnologias como .NET e *Spring Boot*, utilizando testes de carga e estresse para avaliar tempo de resposta e consumo de recursos. No entanto, há uma lacuna em análises que foquem especificamente em *frameworks* modernos como Express.js (JavaScript), Gin (Go) e Actix Web (Rust) em cenários de *e-commerce*.

Portanto, este trabalho justifica-se não apenas pela carência de dados comparativos, mas também pelo potencial de influenciar decisões técnicas em um mercado altamente competitivo.

1.3 HIPÓTESES DE PESQUISA

Este estudo pretende avaliar as seguintes hipóteses:

H1: *Frameworks* modernos, baseados em linguagens compiladas com modelos de concorrência nativos, como Gin (Go) e Actix (Rust), apresentam desempenho superior ao Express.js (Node.js) em cenários de *e-commerce* com alta demanda.

H2: Gin e Actix produzem desempenho similar em termos de latência e confiabilidade, sendo ambos superiores a Express.js, apesar deste último ter performance aceitável.

1.4 ORGANIZAÇÃO DO TRABALHO

Após essa introdução, este trabalho se inicia no capítulo 2 com a fundamentação teórica, que apresenta os conceitos e autores essenciais para embasar a pesquisa. Em seguida, no capítulo 3, a metodologia detalha os procedimentos adotados para o desenvolvimento, os testes executados e a coleta de dados. Os resultados são então expostos e discutidos no capítulo 4. Por fim, o capítulo 5 sintetiza as principais descobertas, suas implicações e

possíveis desdobramentos.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo apresenta os conceitos fundamentais e o referencial teórico necessários para embasar a pesquisa desenvolvida. Inicialmente, são discutidos os princípios das APIs RESTful e sua relevância no contexto de sistemas de *e-commerce*, seguidos pela caracterização da arquitetura típica dessas plataformas. Em seguida, são apresentados os três *frameworks web* selecionados para análise (Actix, Express.js e Gin) detalhando suas arquiteturas, características técnicas e modelos de concorrência. O capítulo também aborda os métodos de avaliação de desempenho de APIs, incluindo os diferentes tipos de testes de performance e a ferramenta k6 utilizada neste estudo. Por fim, é discutido o papel da tecnologia Docker na garantia de reprodutibilidade dos experimentos e padronização dos ambientes de execução.

2.1 CONCEITOS BÁSICOS

2.1.1 API RESTful

As *Application Programming Interfaces* (APIs) RESTful representam um estilo arquitetural para sistemas distribuídos nos princípios do *Representational State Transfer* (REST), introduzido por Fielding (2000) em sua tese de doutorado. Segundo Masse (2011), as APIs REST proporcionam uma arquitetura simples, escalável e independente de plataforma, características que as tornaram amplamente adotadas no desenvolvimento de sistemas *web* modernos.

O padrão REST se baseia em seis princípios fundamentais que garantem a sua eficiência e escalabilidade. O primeiro deles é a arquitetura cliente-servidor, que estabelece uma separação clara entre as responsabilidades do cliente e do servidor, permitindo que ambos evoluam independentemente. O segundo princípio se refere ao estado *stateless*, onde cada requisição deve conter todas as informações necessárias para ser processada, sem que o servidor mantenha contexto sobre as requisições anteriores.

A uniformidade da interface estabelece o terceiro princípio, definindo que recursos devem ser identificados por URIs únicos e manipulados através de representações padronizadas. Biehl (2016) destaca que essa uniformidade simplifica a arquitetura do sistema e melhora a visibilidade das interações. O quarto princípio, denominado cacheabilidade, permite que respostas sejam armazenadas em cache para melhorar a performance e reduzir

a latência.

O sistema em camadas representa o quinto princípio, possibilitando que a arquitetura seja composta por camadas hierárquicas que restringem o comportamento dos componentes. Por fim, o código sob demanda, embora opcional, permite que o servidor envie código executável para estender a funcionalidade do cliente temporariamente.

Os métodos HTTPs utilizados em APIs REST seguem uma semântica específica: GET para recuperação de recursos, POST para criação, PUT para atualização completa, PATCH para atualização parcial e DELETE para remoção. Esta padronização dos métodos, combinada com os status HTTP apropriados, proporciona uma comunicação clara e previsível entre cliente e servidor, aspectos essenciais para a manutenibilidade e documentação de sistemas complexos.

No contexto do *e-commerce*, as APIs RESTful desempenham papel fundamental na comunicação entre diferentes serviços. Elas facilitam operações essenciais como consulta de produtos, processamento de pedidos, gerenciamento de estoque e integração com sistemas de pagamento. A adoção deste padrão por grandes *players* do mercado, demonstra sua adequação para sistemas que precisam suportar alto volume de transações e garantir escalabilidade horizontal.

2.1.2 Arquitetura de *e-commerce*

A arquitetura de sistemas de *e-commerce* evoluiu significativamente nas últimas décadas, passando de aplicações monolíticas para arquiteturas distribuídas e baseadas em serviços. Segundo Almeida (2022), plataformas modernas de comércio eletrônico frequentemente adotam arquitetura de microserviços, onde diferentes funcionalidades são separadas em serviços independentes que se comunicam através de APIs.

Em uma arquitetura típica de *e-commerce* B2C (*Business to Consumer*), as APIs desempenham papel fundamental ao viabilizar a comunicação entre os diversos componentes do sistema. Almeida implementou uma plataforma utilizando três APIs distintas: uma para clientes, uma para vendedores e outra para transações, evidenciando a modularização característica dessas arquiteturas.

Os componentes essenciais de um sistema de *e-commerce* incluem: gerenciamento de catálogo de produtos, controle de estoque, processamento de pedidos, autenticação e autorização de usuários, carrinho de compras, processamento de pagamentos e logística de entrega. Cada um desses componentes pode ser implementado como um serviço

independente, comunicando-se através de APIs RESTful.

2.1.3 O que é *web framework*

Um *web framework* constitui uma coleção de componentes de *software* e bibliotecas que fornecem ferramentas necessárias para o desenvolvimento de aplicações *web*. Segundo Meglio et al. (2023), *frameworks web* facilitam tarefas comuns como roteamento, manipulação de requisições HTTP, gerenciamento de sessões, *middleware* para processamento de requisições, validação de dados, serialização JSON, mecanismos de tratamento de erros, integração com bancos de dados, etc. Permitindo que desenvolvedores concentrem-se na lógica de negócio da aplicação.

Os *frameworks web* modernos oferecem abstrações que simplificam o desenvolvimento, reduzindo a quantidade de códigos estruturais redundantes necessários e aumentando a produtividade. Esta padronização não apenas acelera o desenvolvimento, mas também promove boas práticas de programação e facilita a manutenção do código.

2.1.3.1 Actix

Actix representa um *framework web* web de alta performance desenvolvido para a linguagem Rust, construído sobre o sistema de atores Actix e reconhecido por sua performance excepcional em *benchmarks* de desempenho (MAIOR, 2023). O *framework* aproveita as características únicas do Rust, incluindo segurança de memória sem *garbage collection* e sistema de tipos expressivo, para fornecer tanto performance quanto segurança.

A arquitetura fundamental do Actix baseia-se no padrão *Actor Model*, implementado através de um sistema de *workers* baseado em *thread pools*. Por padrão, Actix cria um *pool* de *threads workers* igual ao número de núcleos lógicos de CPU disponíveis, onde cada *worker* possui sua própria fila de tarefas (*work-stealing queue*). Requisições HTTP são distribuídas entre estes *workers* através de um mecanismo de *load balancing* interno. Este modelo proporciona alta concorrência através de processamento paralelo real, explorando plenamente arquiteturas *multi-core*, diferenciando-se de abordagens baseadas em concorrência cooperativa ou *event loops single-threaded*.

O padrão *Actor Model* proporciona isolamento efetivo de falhas, onde problemas em um ator não afetam o funcionamento de outros atores do sistema, contribuindo para maior estabilidade da aplicação. Cada ator possui *mailbox* (fila de mensagens) própria, processando mensagens de forma assíncrona e sequencial. Sob alta carga, estas *mailboxes*

podem acumular mensagens pendentes, resultando em latência adicional até que a fila seja drenada completamente. Esta característica implica que, após períodos de sobrecarga, o sistema pode apresentar inércia na recuperação enquanto processa o *backlog* acumulado.

O *framework* beneficia-se das garantias de segurança de memória fornecidas pelo sistema de *ownership* de Rust, eliminando categorias inteiras de *bugs* comuns em desenvolvimento *web*, incluindo *null pointer dereferences*, *buffer overflows* e *data races*. Esta segurança é obtida sem *overhead* de *runtime*, diferentemente de linguagens que dependem de *garbage collection*.

Segundo comparações de performance realizadas por *benchmarks* independentes, Actix consistentemente ocupa posições de destaque em métricas de *throughput* e latência, frequentemente superando *frameworks* implementados em outras linguagens (SILVA et al., 2023). Esta combinação de performance excepcional e segurança torna o Actix ideal para aplicações críticas onde tanto eficiência quanto confiabilidade são requisitos essenciais.

2.1.3.2 Express.js

Express.js é um *framework web* minimalista e flexível para Node.js, amplamente reconhecido por sua simplicidade e filosofia de "menos é mais" (MEGLIO et al., 2023). Lançado em 2010 por TJ Holowaychuk, tornou-se rapidamente o *framework* padrão de facto para desenvolvimento de APIs e aplicações *web* em JavaScript no lado servidor, sendo atualmente mantido pela OpenJS Foundation.

O *framework* adota uma arquitetura baseada em *middleware*, onde cada *middleware* é uma função que tem acesso ao objeto de requisição, ao objeto de resposta e à próxima função *middleware* no ciclo de requisição-resposta da aplicação (SILVA et al., 2023). Esta abordagem permite o processamento sequencial e modular de requisições, facilitando a implementação de funcionalidades como autenticação, *logging*, compressão e tratamento de erros.

Express.js oferece um sistema de roteamento robusto que suporta métodos HTTP padrão (GET, POST, PUT, DELETE), parâmetros de rota dinâmicos, *wildcards* e expressões regulares (MEGLIO et al., 2023) A flexibilidade do *framework* permite tanto o desenvolvimento de APIs RESTful quanto aplicações *web* tradicionais, sendo particularmente adequado para microserviços devido à sua natureza leve e ao vasto ecossistema de módulos disponíveis através do npm (gerenciador de pacotes).

Segundo *benchmarks* realizados por Silva et al. (2023), Express.js demonstra

performance competitiva em cenários de alta concorrência, beneficiando-se do motor V8 do Node.js e do modelo de I/O não-bloqueante baseado em *event loop*. Esta arquitetura assíncrona permite que uma única *thread* gerencie milhares de conexões simultâneas, resultando em baixo *overhead* de memória. Entretanto, a natureza *single-threaded* do *event loop* do Node.js impõe limitações significativas em cenários de alta carga computacional: por padrão, o runtime executa código JavaScript em uma única *thread*, não aproveitando nativamente arquiteturas *multi-core*. Esta característica, embora eficiente para operações *I/O-bound*, pode resultar em subutilização de CPU em sistemas com múltiplos núcleos processadores, especialmente sob alta concorrência. Estratégias de escalabilidade horizontal, como *clustering* via módulo nativo 'cluster' ou ferramentas como PM2, tornam-se necessárias para distribuir a carga entre múltiplos processos e explorar plenamente o hardware disponível (MARDAN, 2014).

2.1.3.3 Gin

Gin é um *framework web* de alta performance desenvolvido para a linguagem Go, criado com o objetivo específico de maximizar velocidade e minimizar *overhead* de memória (SANTOS; OLIVEIRA, 2023). Desenvolvido por Manu Martinez-Almeida e lançado oficialmente em 2014, Gin rapidamente se estabeleceu como uma das opções mais populares para desenvolvimento *web* em Go devido à sua combinação única de simplicidade e performance excepcional.

O *framework* implementa um roteador customizado baseado em estrutura de dados *radix tree*, que proporciona tempo de busca logarítmico para resolução de rotas, resultando em roteamento extremamente eficiente mesmo com um grande número de *endpoints* (MEGLIO et al., 2023) Esta implementação otimizada permite que Gin processe requisições com latência mínima, tornando-o ideal para aplicações que demandam alta performance.

A eficiência superior do Gin em cenários de alta concorrência está intrinsecamente relacionada ao modelo de concorrência da linguagem Go. Diferentemente de *threads* tradicionais, Go utiliza *goroutines*, funções leves executadas de forma concorrente que são multiplexadas sobre um conjunto menor de *threads* do sistema operacional pelo *runtime scheduler* do Go. *Goroutines* possuem *footprint* de memória inicial de aproximadamente 2KB (comparado a 1-2MB de *threads* do sistema), permitindo que milhares delas operem simultaneamente sem sobrecarga significativa. O *scheduler* do Go implementa trabalho cooperativo com preempção, distribuindo automaticamente *goroutines* entre todos os núcleos de CPU disponíveis (modelo M:N *scheduling*). Esta arquitetura permite que o

Gin escale verticalmente de forma natural, aproveitando eficientemente processadores *multi-core* sem necessidade de configuração explícita de *pools* de *threads*, diferenciando-se fundamentalmente de abordagens baseadas em *event loops single-threaded*.

Gin oferece funcionalidades integradas essenciais para desenvolvimento *web* moderno, incluindo *middleware* para *logging*, *recovery* automático de *panics*, suporte nativo a CORS (*Cross-Origin Resource Sharing*), compressão de resposta e *rate limiting* (SANTOS; OLIVEIRA, 2023). O *framework* também fornece *binding* automático e validação de dados para JSON, XML, *form data* e *query parameters*, utilizando *struct tags* do Go para definir regras de validação de forma declarativa.

De acordo com análises comparativas realizadas por Meglio et al. (2023) e Szewczyk e Skublewska-Paszkowska (2025), *frameworks* baseados em Go, incluindo Gin, consistentemente demonstram performance superior em testes de *throughput*, demonstrando capacidade de processar milhares de requisições por segundo com consumo mínimo de recursos computacionais. Esta performance superior é atribuída à natureza compilada da linguagem Go e ao design eficiente do *garbage collector*.

2.2 MÉTODOS DE AVALIAÇÃO DE DESEMPENHO DE APIS

A avaliação de desempenho de APIs constitui etapa fundamental no desenvolvimento de sistemas *web* modernos, permitindo identificar gargalos, validar requisitos de qualidade e garantir que a aplicação suporte adequadamente a carga esperada em ambiente de produção. Diferentes metodologias de teste foram desenvolvidas para simular cenários variados de utilização, desde condições normais de operação até situações extremas de sobrecarga.

Segundo Kemer e Samli (2019), testes de performance são essenciais para verificar como sistemas funcionam sob diferentes cargas concorrentes de usuários virtuais realizando transações durante períodos específicos. Entre as principais abordagens destacam-se os testes de carga, que avaliam o comportamento do sistema sob demanda esperada; testes de estresse, que examinam os limites operacionais sob condições progressivamente intensas; testes de pico, que simulam aumentos súbitos de tráfego; e testes de ruptura, que identificam o ponto máximo de capacidade antes do colapso sistêmico.

Conforme destacado por Godinho et al. (2024), além da análise de carga, testes de estresse também examinam vazamentos de memória, lentidão, questões de segurança e corrupção de dados. Meglio et al. (2023) reforçam que a execução sistemática desses testes, combinada com métricas objetivas como *throughput*, latência, consumo de recursos

e taxa de falhas, fornece subsídios técnicos fundamentais para decisões arquiteturais e dimensionamento de infraestrutura em ambientes de produção.

2.2.1 Ferramenta de teste

Para a realização dos testes de desempenho foi utilizada exclusivamente a ferramenta k6. Trata-se de uma ferramenta de código aberto desenvolvida para execução de testes de carga e performance em APIs e aplicações *web*. Sua principal proposta é oferecer uma interface de *script* acessível, baseada em JavaScript.

O k6 permite simular cenários com usuários virtuais (VUs), ajustar quantidades desses VUs, definir fluxos personalizados, validar respostas com verificações automatizadas, etc. A ferramenta se destaca pela facilidade de criação de fluxos que simulam o comportamento real de usuários, além de oferecer suporte nativo a métricas detalhadas, como tempo de resposta, taxa de erros, volume de dados transferidos e estatísticas baseadas em percentis.

Segundo Araujo (2024), o k6 foi adotado com sucesso em testes comparativos entre APIs desenvolvidas em Node.js, .NET e Go, demonstrando ser uma solução robusta e adequada para experimentos com alto grau de controle e repetibilidade.

2.3 DOCKER

Docker constitui uma plataforma de conteinerização que permite empacotar aplicações e suas dependências em unidades padronizadas e autônomas denominadas contêineres (NüST et al., 2020). Esta tecnologia revolucionou o desenvolvimento de software ao possibilitar a criação de ambientes computacionais consistentes, portáveis e reproduzíveis, fundamentais para garantir que aplicações executem de forma idêntica independentemente do ambiente de implantação.

A conteinerização representa uma abordagem de virtualização leve que compartilha o *kernel* do sistema operacional hospedeiro, diferindo-se da virtualização tradicional baseada em hipervisores por não emular *hardware* ou *kernels* de sistema operacional completos, resultando em menor *overhead* de recursos e maior eficiência (MERKEL, 2014). Segundo Boettiger e Eddelbuettel (2017), Docker oferece baixo *overhead*, flexibilidade, portabilidade de aplicações e reprodutibilidade, características que o tornaram líder em tecnologia de conteinerização.

2.3.1 Arquitetura e Funcionamento

Docker utiliza uma arquitetura cliente-servidor baseada no *Docker Engine*, que gerencia a criação, execução e distribuição de contêineres. A tecnologia fundamenta-se em funcionalidades nativas do *kernel* Linux, incluindo *namespaces* para isolamento de processos e *control groups* (cgroups) para limitação e monitoramento de recursos (AHMAD et al., 2017).

Um contêiner Docker é criado a partir de uma imagem, que constitui um *template read-only* contendo o sistema de arquivos, bibliotecas, dependências e configurações necessárias para execução da aplicação. As imagens são construídas através de arquivos de configuração denominados *Dockerfiles*, que especificam uma sequência de instruções para montagem do ambiente computacional (NüST et al., 2020).

2.3.2 Reprodutibilidade Científica

Na pesquisa científica computacional, Docker emergiu como ferramenta fundamental para garantir reprodutibilidade de experimentos e análises. A capacidade de encapsular completamente o ambiente computacional, incluindo sistema operacional, bibliotecas, dependências e código de aplicação, permite que pesquisadores compartilhem não apenas dados e código, mas todo o contexto computacional necessário para reprodução de resultados (BOETTIGER, 2015).

Nüst et al. (2020) argumentam que a conteinerização ajuda a fornecer instruções para empacotamento dos blocos de construção da pesquisa baseada em computador, incluindo código, dados, documentação e ambiente computacional. Esta abordagem melhora significativamente o nível de documentação, transparência e reutilização de trabalhos científicos.

2.3.3 Impacto no Desenvolvimento de APIs

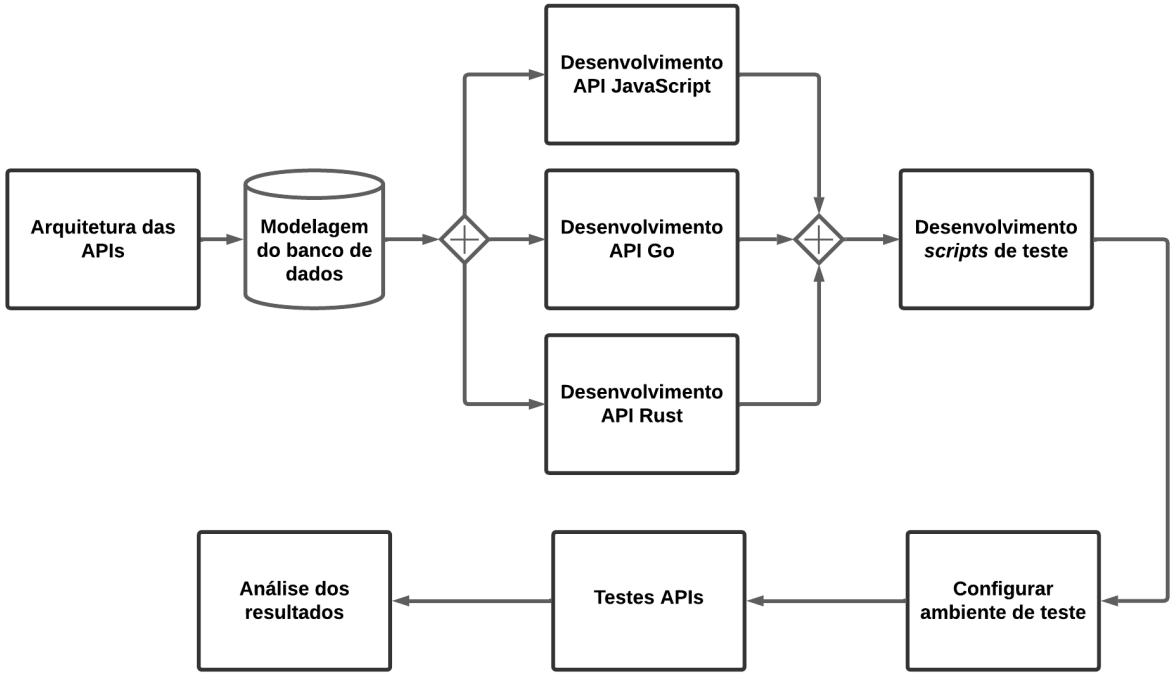
No contexto de desenvolvimento de APIs, Docker oferece benefícios substanciais para padronização de ambientes de execução, facilitação de testes em diferentes configurações e simplificação de processos de implantação. A tecnologia permite que equipes de desenvolvimento mantenham consistência entre ambientes de desenvolvimento, teste e produção, reduzindo significativamente problemas relacionados a diferenças de configuração (GHOLAMI et al., 2020).

A capacidade de criar microserviços conteinerizados facilita a implementação de arquiteturas distribuídas, onde diferentes componentes de uma aplicação podem ser desenvolvidos, testados e implantados independentemente. Esta abordagem alinha-se com práticas modernas de desenvolvimento de software e arquiteturas orientadas a serviços.

3 METODOLOGIA

O procedimento metodológico empregado neste trabalho segue a sequência representada no fluxograma da Figura 3.1, estruturando as etapas necessárias para o desenvolvimento, execução e análise dos experimentos de simulação propostos. Cada etapa foi planejada de forma a garantir consistência nas implementações, confiabilidade nos testes e reprodutibilidade dos resultados, aspectos importantes dos experimentos planejados. A abordagem contempla desde a criação das APIs *RESTful* até a aplicação de testes de desempenho em ambiente controlado.

Figura 3.1 – Sequencia metodológica de desenvolvimento do projeto.



Fonte: Elaborado pelos autores, 2025.

3.1 ARQUITETURA DAS APIS

As APIs foram arquitetadas baseando-se nos princípios da arquitetura RESTful, com a implementação de um sistema CRUD (*Create, Read, Update, Delete*) para simular operações típicas de um ambiente de *e-commerce*, como cadastro e consulta de pedidos e produtos.

A padronização foi um aspecto central na construção dos serviços, para garantir

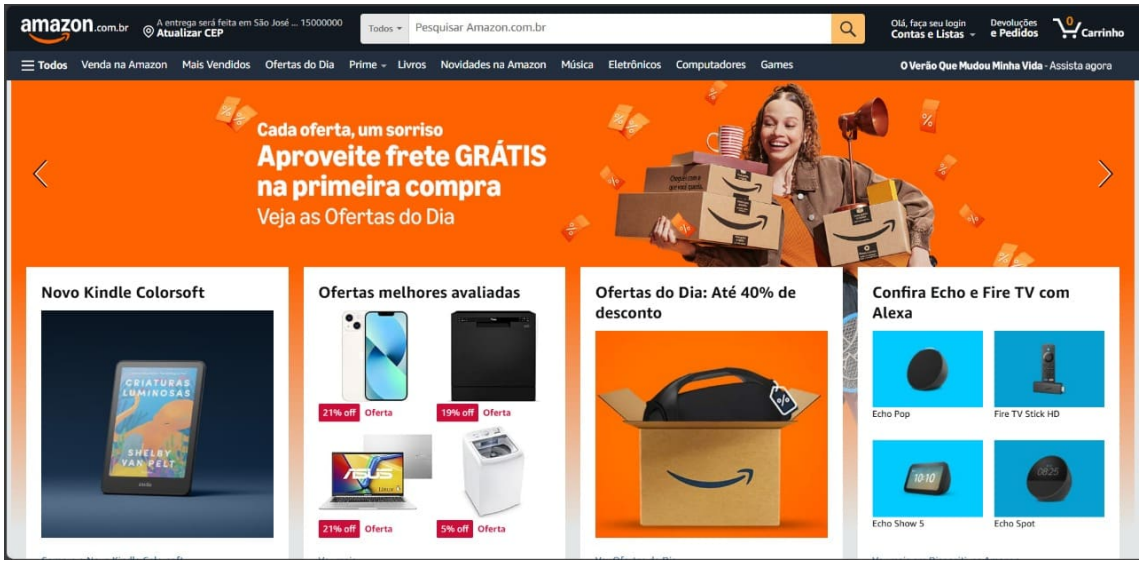
uma comparação justa e tecnicamente coerente. Foram definidos *endpoints* idênticos em todas as implementações, utilizando a mesma estrutura de dados em formato JSON. Essa equivalência estrutural tem como objetivo assegurar a neutralidade da comparação, eliminando interferências causadas por diferenças de modelagem ou comportamento funcional. Assim, os resultados obtidos nos testes refletem unicamente as características inerentes a cada linguagem e *framework* analisado.

3.1.1 Simulação da Plataforma Amazon

Na simulação da Amazon foram implementadas quatro funcionalidades principais, as quais representam os fluxos mais utilizados pelos usuários, a saber: página inicial; navegação por categoria; sistema de busca; página do produto; e autenticação de usuário. As interfaces relacionadas a cada funcionalidade estão ilustradas a seguir.

A página inicial retorna um volume moderado de informações, com uma seleção de ofertas, categorias e alguns produtos agrupados por suas categorias, como mostrado na Figura 3.2.

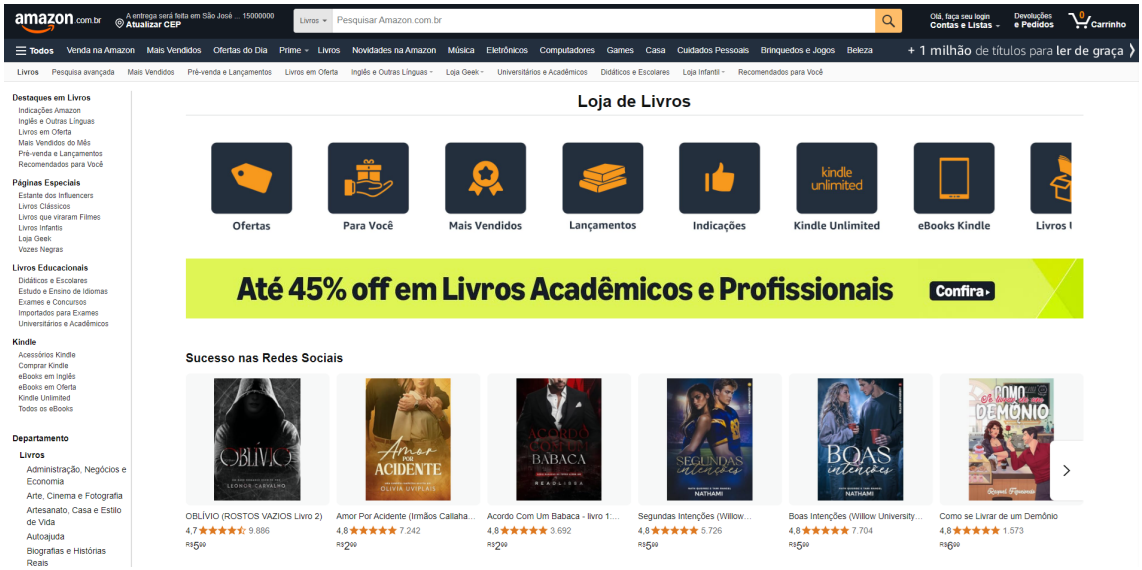
Figura 3.2 – Página inicial da Amazon



Fonte: Amazon, 2025.

A funcionalidade de navegação por categoria permite aos usuários visualizar produtos específicos de uma determinada categoria, apresentando informações detalhadas como preços, avaliações e disponibilidade, conforme ilustrado na Figura 3.3.

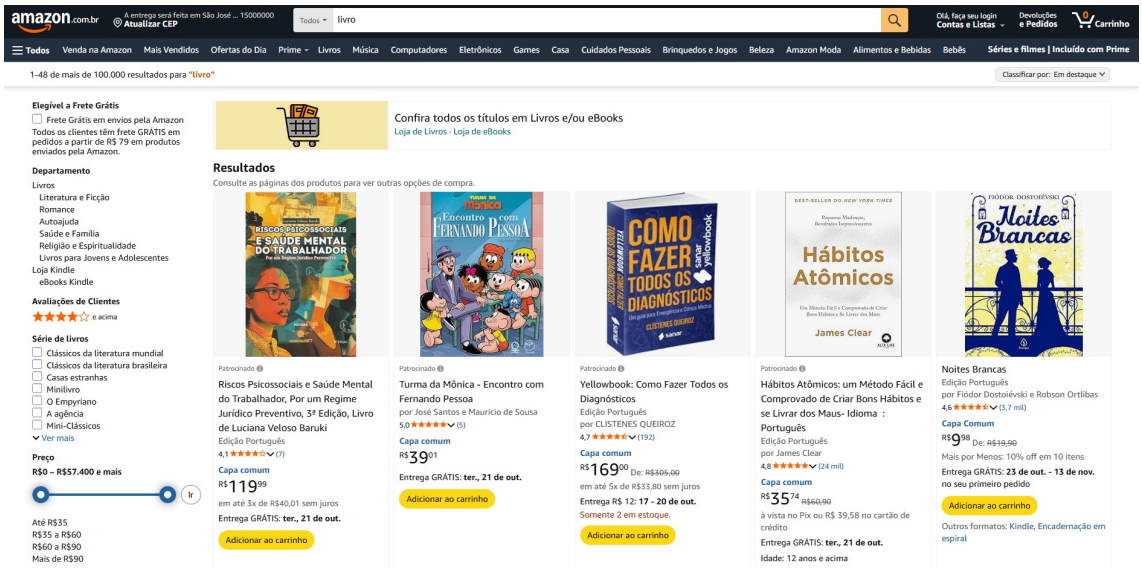
Figura 3.3 – Navegação da página da categoria "Livros" da Amazon



Fonte: Amazon, 2025.

O sistema de busca implementa a funcionalidade de pesquisa por termos específicos, retornando resultados relevantes com filtros e ordenação, similar à experiência mostrada na Figura 3.4.

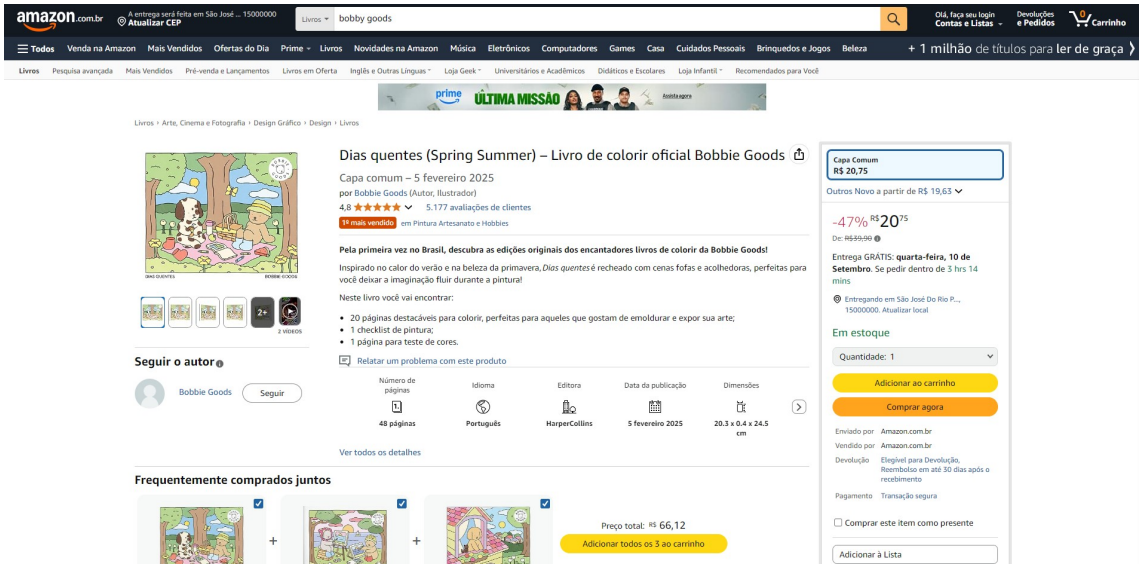
Figura 3.4 – Navegação da página de resultados da busca de um produto da Amazon



Fonte: Amazon, 2025.

A página do produto retorna as informações sobre o produto selecionado como nome, foto, preço, descrição, disponibilidade e outras informações da mercadoria e permite adicioná-la ao carrinho ou realizar a compra de imediato, como é mostrada na Figura 3.5.

Figura 3.5 – Navegação da página de produto da Amazon



Fonte: Amazon, 2025.

O sistema de login fornece autenticação segura para acesso às funcionalidades personalizadas da plataforma, seguindo os padrões de interface apresentados na 3.6.

Figura 3.6 – Navegação da página de autenticação da Amazon

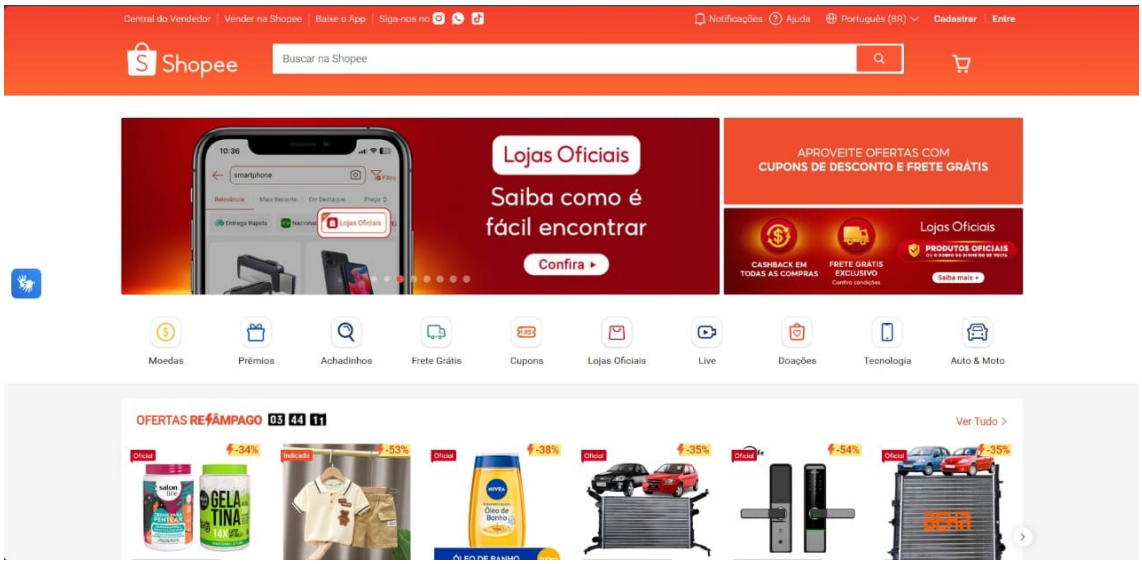
Fonte: Amazon, 2025.

3.1.2 Simulação da Plataforma Shopee

Para a Shopee, a API retorna uma quantidade maior de dados em todas as funcionalidades implementadas, seguindo uma estratégia mais densa de apresentação de informações característica desta plataforma. As funcionalidades em destaque cujas interfaces estão ilustradas a seguir foram: página inicial; navegação por categoria; sistema de busca; página do produto; cadastro de usuário; e login de usuário.

A página inicial da Shopee apresenta listas extensas de produtos, categorias, ofertas especiais e destaques promocionais, como demonstrado na Figura 3.7.

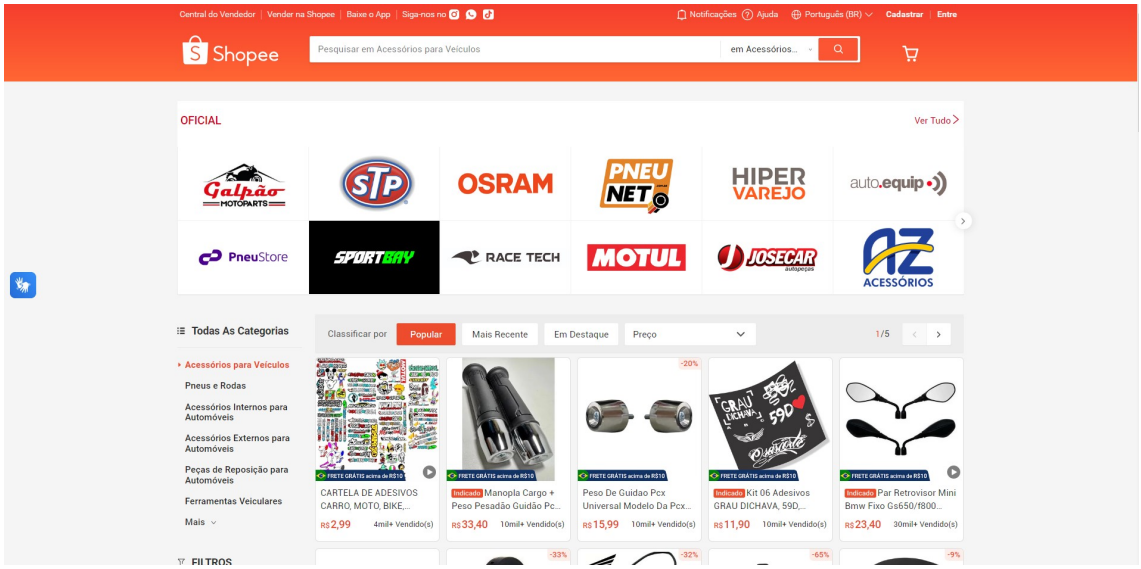
Figura 3.7 – Página inicial da Shopee



Fonte: Shopee, 2025.

A visualização por categoria na Shopee carrega um volume maior de produtos simultaneamente, com informações mais detalhadas sobre promoções, frete grátis e avaliações, conforme mostrado na Figura 3.8.

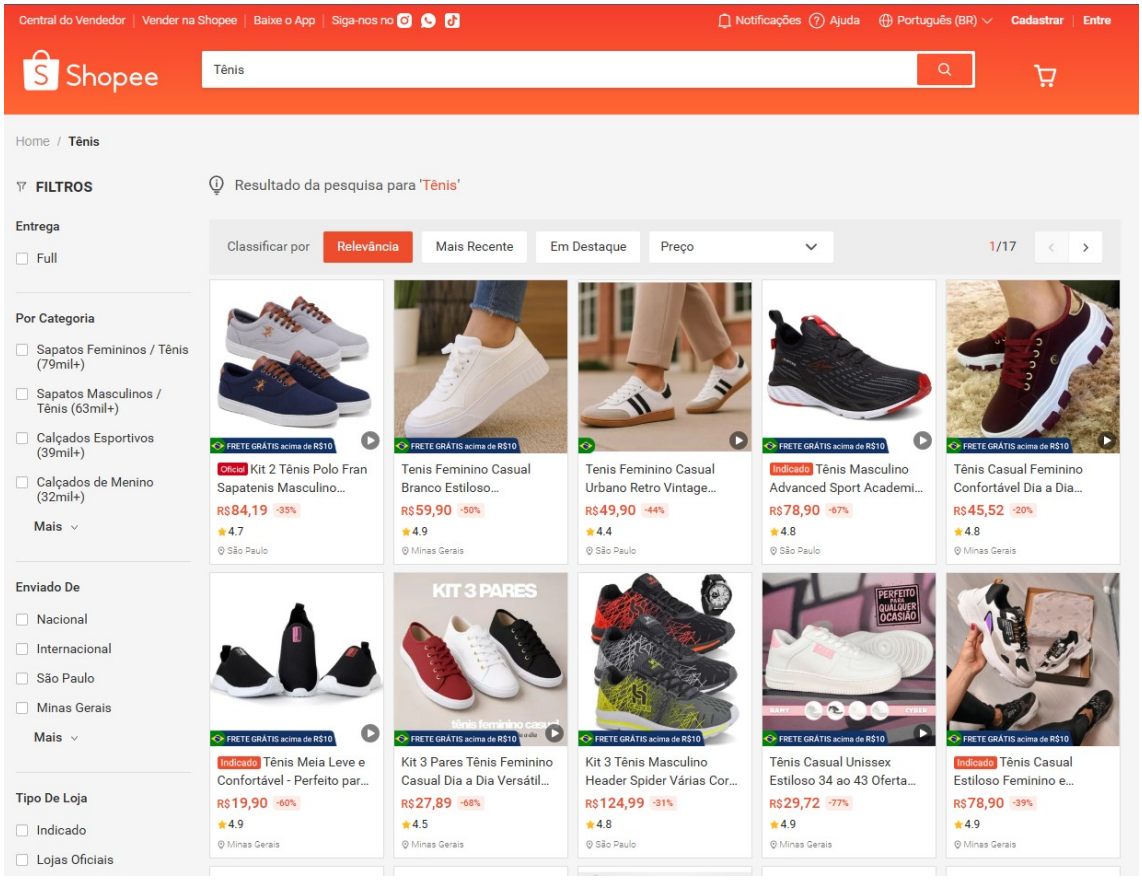
Figura 3.8 – Navegação da página da categoria "Acessórios para Veículos" da Shopee



Fonte: Shopee, 2025.

O sistema de busca da Shopee retorna resultados mais abrangentes, incluindo sugestões relacionadas, produtos similares e múltiplas opções de filtros, como ilustrado na Figura 3.9.

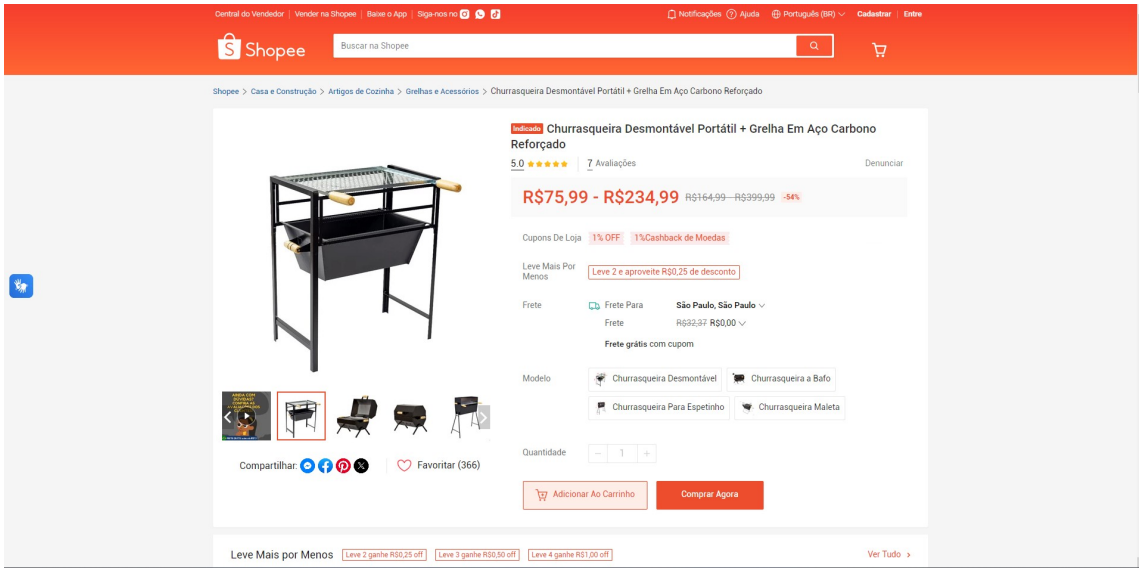
Figura 3.9 – Navegação da página de resultados da busca de um produto da Shopee



Fonte: Shopee, 2025.

A página do produto exibe informações detalhadas da mercadoria selecionada, incluindo nome, imagens, preço, descrição, quantidade em estoque e especificações técnicas. A interface oferece opções para adicionar o item ao carrinho de compras ou proceder diretamente para a finalização da compra, conforme ilustrado na Figura 3.10.

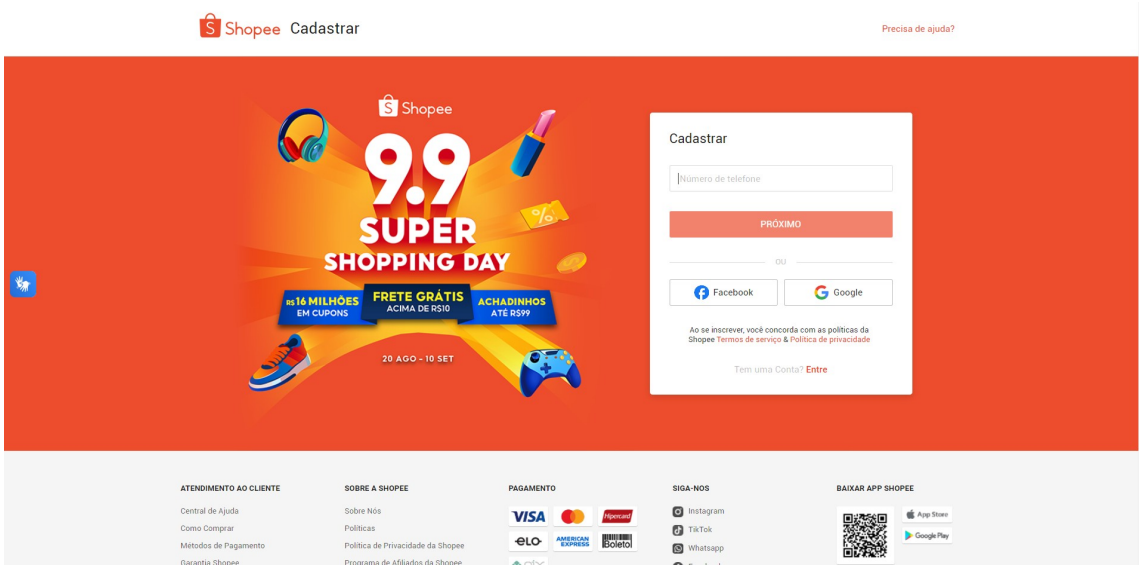
Figura 3.10 – Navegação da página de produto da Shopee



Fonte: Shopee, 2025.

A interface de cadastro da Shopee, trazendo a opção de realizar o cadastro comum, tal como realizar o cadastro via integração com contas da Google e do Facebook, conforme apresentado na Figura 3.11.

Figura 3.11 – Navegação da página de cadastro de usuário da Shopee

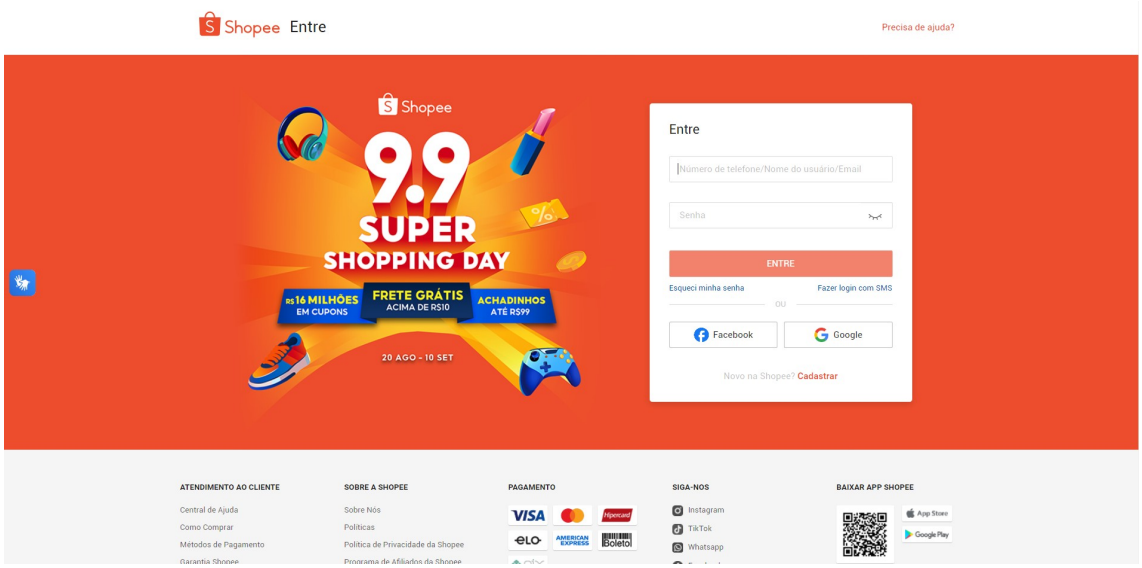


Fonte: Shopee, 2025.

A interface de login da Shopee, se assemelha muito a interface de cadastro, com alterações nos textos da página para o contexto de login e também oferecendo as opções

de login utilizando as contas da Google ou Facebook, ou utilizando o *email* do utilizador, como ilustrado na Figura 3.12.

Figura 3.12 – Navegação da página de login de usuário da Shopee



Fonte: Shopee, 2025.

3.1.3 *Endpoints* implementados nas APIs

Os *endpoints* apresentados na Tabela 3.1 foram organizados por grupos funcionais, cobrindo todas as operações necessárias para simular o fluxo completo de um sistema de *e-commerce*. A estrutura inclui rotas públicas para navegação e consulta, além de rotas protegidas por autenticação, como o gerenciamento de pedidos.

Tabela 3.1 – Lista de *endpoints* disponíveis na API.

Método	Grupo	Caminho	Descrição
GET	Categorias	/categorias	Retorna as categorias dos produtos
POST	Clientes	/clientes/login	Autentica o cliente
POST	Clientes	/clientes	Cria um novo cliente
GET	Home	/home/amazon	Retorna a página inicial da Amazon
GET	Home	/home/shopee	Retorna a página inicial da Shopee
POST	Pedidos	/pedidos	Cria um novo pedido
GET	Produtos	/produtos/	Retorna todos os produtos
GET	Produtos	/produtos/:sku	Retorna um produto específico
GET	Produtos	/produtos/categoria/:categoria	Retorna produtos de uma categoria
GET	Produtos	/produtos/ofertas	Retorna produtos em oferta
GET	Produtos	/produtos/destaques	Retorna produtos em destaque
GET	Produtos	/produtos/nome	Busca produto pelo nome

Fonte: Elaborado pelos autores, 2025.

3.2 MODELAGEM E ESTRUTURA DO BANCO DE DADOS

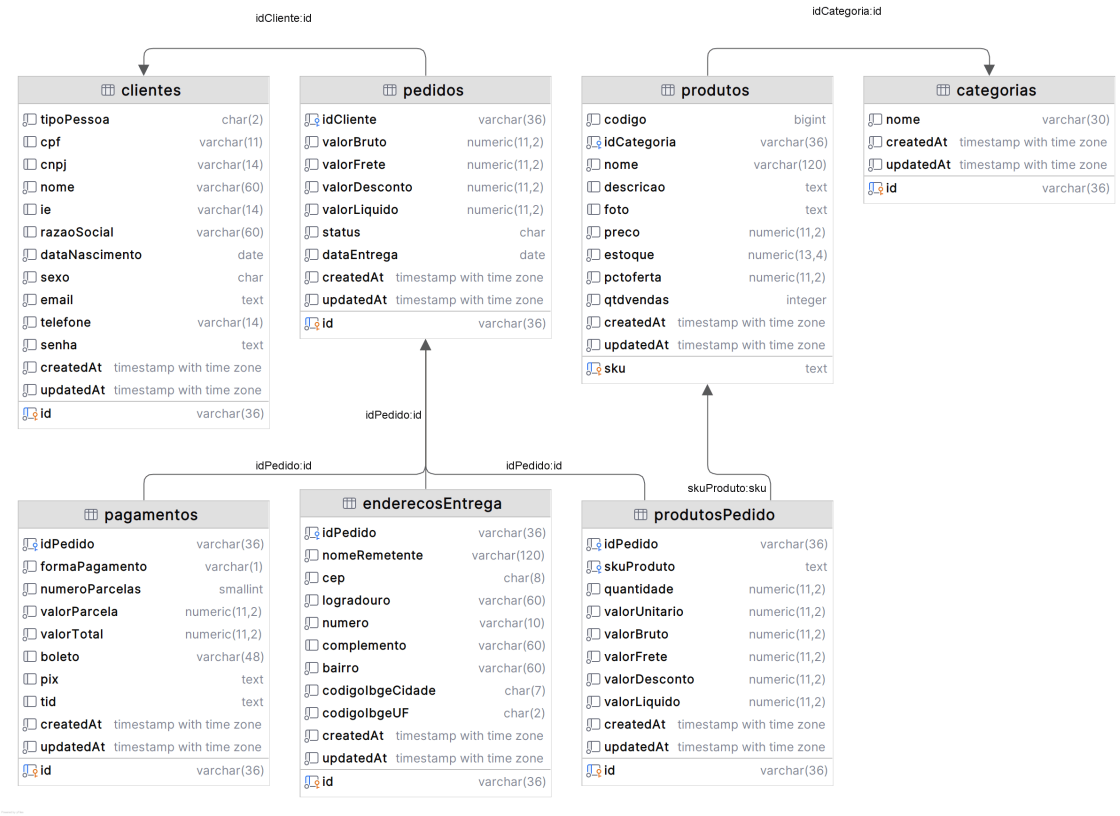
Foi utilizado um único banco de dados compartilhado entre todas as APIs. A modelagem do banco de dados foi orientada a entidades fundamentais de um sistema de *e-commerce*, organizadas em tabelas inter-relacionadas. Todas as entidades utilizam identificadores únicos no formato UUID gerados automaticamente, e os relacionamentos entre tabelas são mantidos por meio de chaves estrangeiras. A seguir, são descritas as entidades que compõe o modelo:

- **CLIENTES**: Armazena os dados de pessoas físicas ou jurídicas;
- **CATEGORIAS**: Contém o nome das categorias de produtos;
- **PRODUTOS**: Representa os produtos disponíveis no sistema;
- **PEDIDOS**: Registra os pedidos realizados pelos clientes. Cada pedido está relacionado a um cliente;
- **PRODUTOS-PEDIDO**: Tabela intermediária que associa os produtos a cada pedido;
- **ENDEREÇOS ENTREGA**: Armazena os dados de entrega dos pedidos;
- **PAGAMENTOS**: Contém as informações da forma de pagamento de cada pedido.

A Figura 3.13 mostra a relação de cada tabela no banco de dados, junto com seus principais campos.

Os dados de produtos foram obtidos a partir de um arquivo CSV extraído da plataforma Kaggle e disponibilizado por Asaniczka (2023), que foi processado e importado para preencher as tabelas “produtos” (mais de 1 milhão de registros) e “categorias” (205 entradas) do banco de dados.

Figura 3.13 – Relação das tabelas do banco de dados



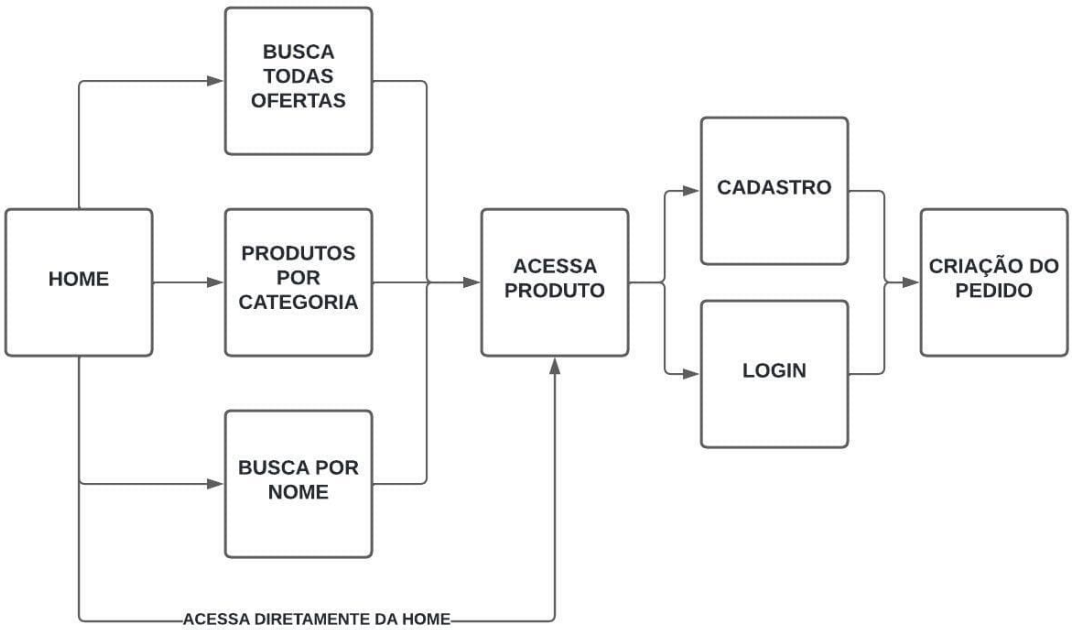
Fonte: Elaborado pelos autores, 2025.

3.3 *SCRIPTS* DE TESTES

Com o objetivo de avaliar o desempenho das APIs em distintas condições de carga, foram elaborados quatro *scripts* de teste. Cada um deles representa um cenário de demanda específico, permitindo analisar o comportamento das aplicações sob diferentes intensidades de uso.

Todos os testes executados foram planejados para reproduzir cenários realistas de navegação em um *e-commerce*. Para tanto, foram definidos fluxos distintos de interação, específicos para a Amazon e para a Shopee, conforme descrito a seguir.

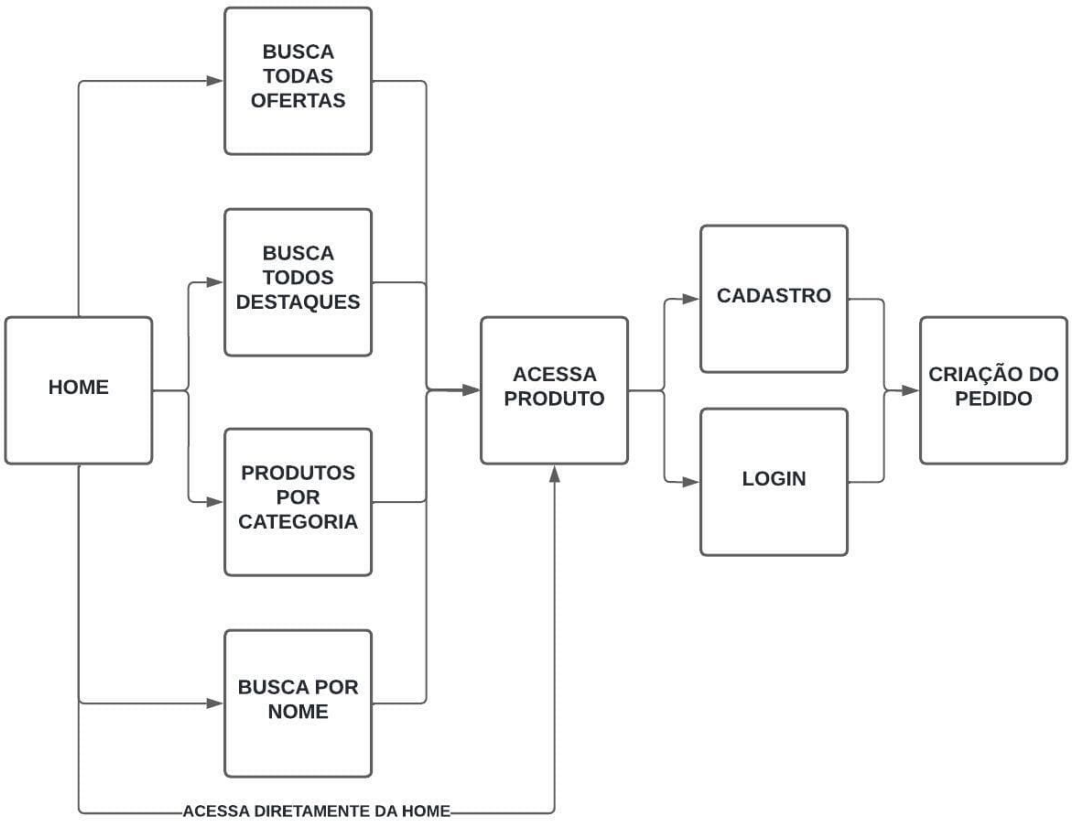
Figura 3.14 – Fluxo - Amazon



Fonte: Elaborado pelos autores, 2025.

O primeiro fluxo, representado na Figura 3.14, refere-se à navegação simulada na plataforma Amazon. O usuário inicia na página inicial (*home*), de onde pode seguir múltiplos caminhos de interação. É possível acessar diretamente a listagem de todas as ofertas disponíveis, realizar uma busca por produtos específicos via nome ou explorar itens por categoria. A partir desses pontos de entrada, o usuário alcança a página de um produto, etapa central do fluxo. Nesse estágio, para prosseguir com a criação de um pedido, torna-se necessário realizar o cadastro ou efetuar o login no sistema, o que viabiliza o acesso à etapa final: a conclusão do pedido.

Figura 3.15 – Fluxo - Shopee



Fonte: Elaborado pelos autores, 2025.

O segundo fluxo, ilustrado na Figura 3.15, descreve o comportamento de navegação na plataforma Shopee. Assim como no fluxo anterior, o ponto de partida é a página inicial, que oferece múltiplas possibilidades de exploração. Nesse caso, além das opções de busca por nome, listagem de ofertas e navegação por categorias, há ainda a consulta aos produtos em destaque, recurso característico da plataforma. Todos esses caminhos convergem para o acesso à página de um produto específico, etapa que, assim como no fluxo anterior, antecede o cadastro ou login do usuário e, em seguida, a etapa de criação do pedido, finalizando sua compra.

3.3.1 Teste de carga

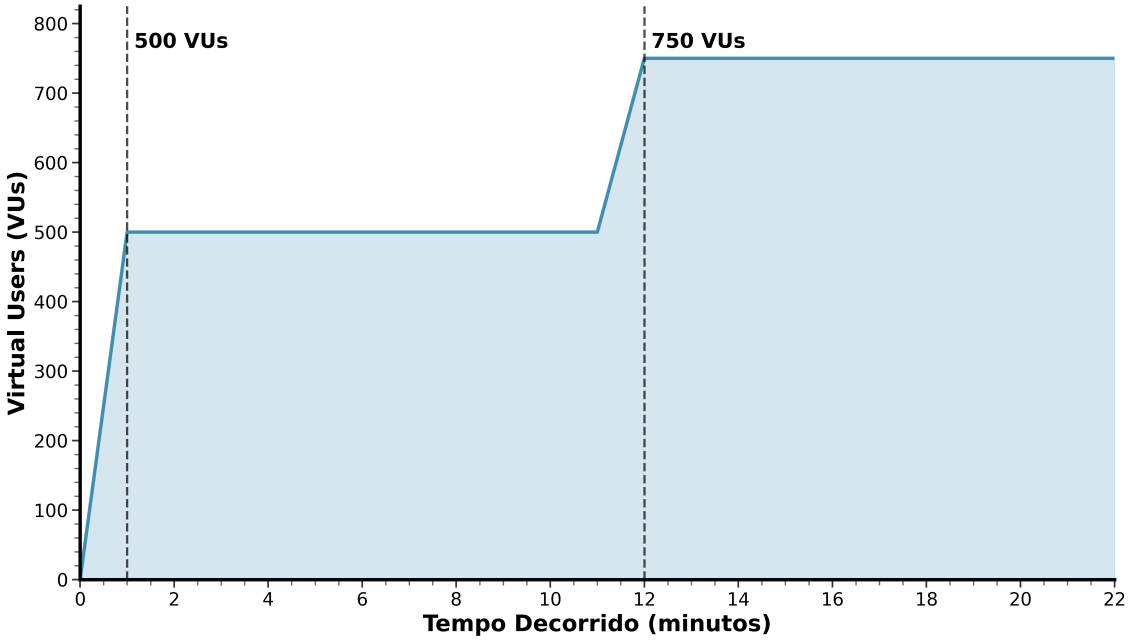
O teste de carga é uma técnica de teste de desempenho que mede a resposta de um sistema sob várias condições de carga. Segundo Hendayun et al. (2023), este teste é essencial para determinar como o *software* se comporta quando múltiplos usuários estão utilizando o sistema simultaneamente. O objetivo principal é simular o uso concorrente de

um site ou aplicação, permitindo verificar se o sistema consegue suportar a carga esperada de usuários em condições normais de operação.

O teste de carga difere-se de outros tipos de teste de performance por focar especificamente em cenários de uso normal, onde o sistema opera dentro dos limites esperados de usuários concorrentes, permitindo assim uma avaliação precisa da capacidade operacional da aplicação em condições típicas de funcionamento.

O teste de carga foi estruturado com duração total de 22 minutos, organizado em dois patamares progressivos de carga. O sistema foi submetido a um *warm-up* de 1 minuto até alcançar 500 VUs, mantidos por 10 minutos, seguido por intensificação para 750 VUs por mais 10 minutos (Figura 3.16).

Figura 3.16 – Evolução dos VUs - Teste de carga



Fonte: Elaborado pelos autores, 2025.

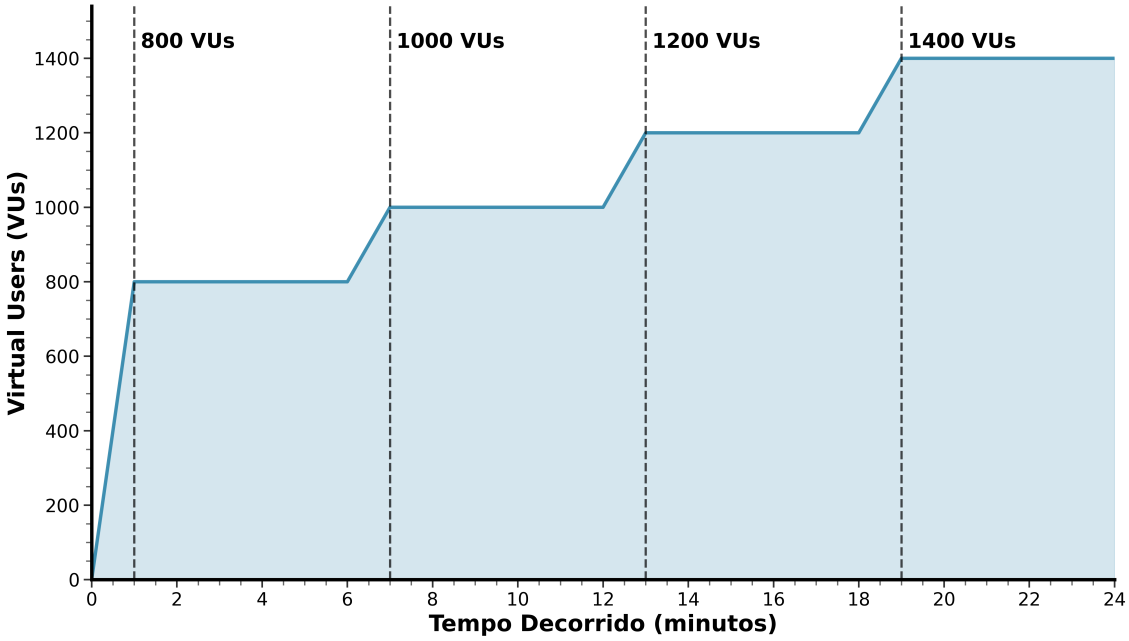
3.3.2 Teste de estresse

O teste de estresse é uma técnica de teste de desempenho que examina os limites superiores de um sistema, submetendo-o a cargas extremas para verificar seu comportamento sob condições adversas. Segundo Kemer e Samli (2019), além da análise de carga, o teste de estresse também examina vazamentos de memória, lentidão, questões de segurança e corrupção de dados.

O teste de estresse foi estruturado com duração total de 24 minutos, organizado em

quatro patamares progressivos (800, 1000, 1200 e 1400 VUs). Cada patamar foi mantido por 5 minutos, com transições de 1 minuto entre os níveis, permitindo identificar gradualmente os pontos de degradação de desempenho (Figura 3.17).

Figura 3.17 – Evolução dos VUs - Teste de estresse



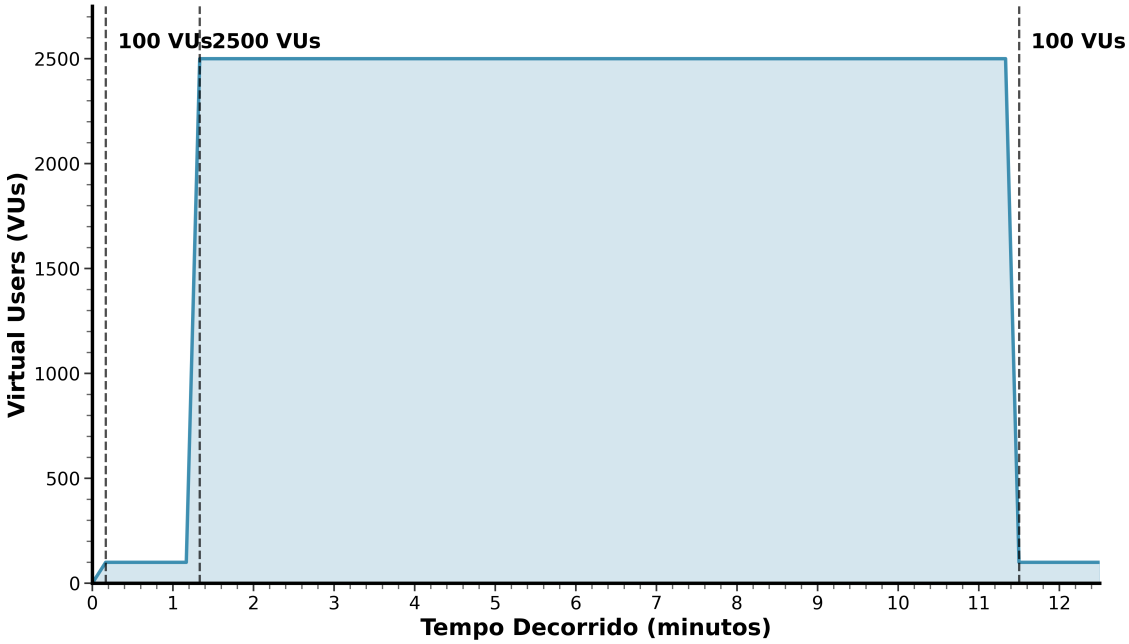
Fonte: Elaborado pelos autores, 2025.

3.3.3 Teste de pico

O teste de pico é uma variação do teste de estresse que se caracteriza por aumentar rapidamente a carga para níveis extremos em um período muito curto de tempo. Segundo Godinho et al. (2024), o objetivo de um teste de pico é avaliar a capacidade do sistema de lidar com surtos súbitos de tráfego e identificar quaisquer gargalos de desempenho ou problemas que possam surgir. Este teste permite a detecção precoce de problemas potenciais antes que ocorram em um ambiente de produção e garante que o sistema possa lidar com os níveis de tráfego antecipados.

O teste de pico foi estruturado com duração total de 12 minutos e 30 segundos, simulando um pico súbito de tráfego. Após um *warm-up* de 10 segundos e 1 minuto em 100 VUs, a carga foi elevada abruptamente para 2500 VUs (aumento de 2400%) e mantida por 10 minutos, avaliando a resposta imediata e a estabilidade sob pressão prolongada (Figura 3.18).

Figura 3.18 – Evolução dos VUs - Teste de pico



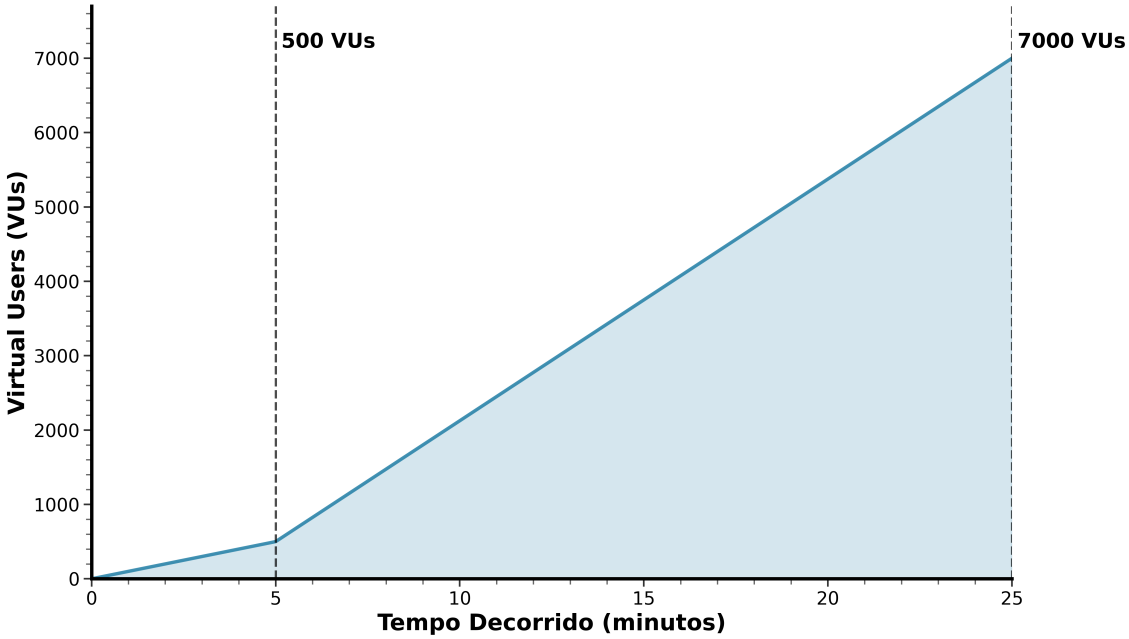
Fonte: Elaborado pelos autores, 2025.

3.3.4 Teste de ruptura

O teste de ruptura (*breakpoint testing*) é uma modalidade de teste de carga que tem como objetivo identificar os limites máximos do sistema. Conforme definido pelo Grafana Labs Team (2024), este tipo de teste aumenta gradualmente a carga até níveis extremamente elevados para determinar o ponto exato onde o sistema começa a falhar ou apresentar degradação significativa de desempenho.

O teste de ruptura foi configurado com duração total de 25 minutos, implementando um estágio inicial de 5 minutos com 500 VUs, seguido por crescimento contínuo até 7.000 usuários virtuais simultâneo, permitindo identificar com precisão o ponto de degradação do sistema (Figura 3.19).

Figura 3.19 – Evolução dos VUs - Teste de ruptura



Fonte: Elaborado pelos autores, 2025.

3.4 AMBIENTE DE EXECUÇÃO

As aplicações foram implantadas em um ambiente local, utilizando tecnologias de conteinerização. O quadro 3.1 apresenta as especificações do contêiner utilizado.

Quadro 3.1 – Especificações do contêiner

Sistema Operacional	*Alpine*
vCPUs	4.0
Memória RAM	4Gb

Fonte: Elaborado pelos autores, 2025.

Cada API RESTful foi conteinerizada utilizando Docker e executada de forma isolada em contêineres independentes hospedados em uma máquina local com as especificações apresentadas no quadro 3.2.

Quadro 3.2 – Especificações do ambiente de execução

Sistema Operacional	Windows 11
Processador	AMD Ryzen 7 5700X3D (8 núcleos, 16 threads)
Placa-mãe	Asus B550
Memória RAM	32 GB
Tecnologia de Virtualização	Docker

Fonte: Elaborado pelos autores, 2025.

O banco de dados utilizado por todas as APIs foi o PostgreSQL, sendo compartilhada por todas as implementações para manter a equivalência das interações com o banco.

3.4.1 Versões de *Software* e Dependências

Para garantir reprodutibilidade dos experimentos, as seguintes versões de *software* foram utilizadas:

Quadro 3.3 – Versões de *software* utilizadas na pesquisa

PostgreSQL	15.03
Go	1.24.1
Node.js	18.19.0
Rust	1.83.0

Fonte: Elaborado pelos autores, 2025.

As dependências do Rust foram especificadas no arquivo `cargo.toml`, apresentadas no Quadro 3.4:

Quadro 3.4 – Dependências do Rust

Crate	**Versão**
actix-web	4.10.2
tokio	1.44.2
diesel	2.2.10
serde_json	1.0.140
jsonwebtoken	9.3.1
bcrypt	0.17.0

Fonte: Elaborado pelos autores, 2025.

As dependências do JavaScript foram registradas no arquivo `package-lock.json`, apresentadas no Quadro 3.5:

Quadro 3.5 – Dependências do JavaScript

express	4.21.2
prisma	6.4.1
jsonwebtoken	9.0.2
bcryptjs	3.0.2
cors	2.8.5
zod	3.24.2

Fonte: Elaborado pelos autores, 2025.

Por fim, as dependências do Go foram definidas no arquivo `go.sum`, apresentadas no Quadro 3.6:

Quadro 3.6 – Dependências do Go

Módulo	Versão
github.com/gin-gonic/gin	1.10.0
gorm.io/gorm	1.25.12
gorm.io/driver/postgres	1.5.11
github.com/golang-jwt/jwt	3.2.2
golang.org/x/crypto	0.36.0
github.com/google/uuid	1.6.0

Fonte: Elaborado pelos autores, 2025.

3.4.2 Configuração da Rede de Testes

A ferramenta k6 foi executada em um contêiner isolado, separado da rede do hospedeiro, estabelecendo conexão exclusiva com os contêineres das APIs através de uma rede Docker dedicada. Esta arquitetura de isolamento garante que o tráfego de teste não sofra interferência do sistema operacional hospedeiro, eliminando variabilidades causadas por outros processos. Os testes foram executados de forma sequencial (um por vez), evitando contenção de recursos entre múltiplas execuções simultâneas e garantindo que o banco de dados compartilhado não representasse um gargalo artificial. A comunicação entre o k6 e as APIs ocorreu através de requisições HTTP nativas ao endereço do contêiner, sem *overhead* de tradução de rede ou camadas adicionais de roteamento.

3.5 MÉTRICAS AVALIADAS

Para avaliar o desempenho das APIs desenvolvidas foram estabelecidos critérios objetivos baseados em métricas amplamente utilizadas na engenharia de *software* e testes de performance. Além disso, diferentes tipos de teste foram aplicados com o objetivo de simular cenários variados de uso, desde acessos simultâneos extremos até fluxos realistas de navegação e compra em um sistema de *e-commerce*.

Foram definidas **cinco** métricas principais:

- **Requisições por segundo (*throughput*):** Refere-se à quantas chamadas na API o *script* está realizando. Quanto mais rápida for a resposta da API mais requisições por segundo o *script* consegue realizar;

- **Tempo de resposta:** Refere-se ao tempo decorrido entre o envio de uma requisição HTTP e o recebimento da resposta completa. Esta métrica é analisada através de percentis estatísticos, que oferecem visão mais representativa da experiência do usuário comparado a médias simples. O percentil 90 (p90) indica que 90% das requisições foram processadas em tempo igual ou inferior ao valor reportado, enquanto 10% experimentaram latências superiores. Similarmente, o percentil 95 (p95) representa a experiência dos 5% de usuários com pior desempenho. A mediana (p50) representa o tempo de resposta típico, não sendo afetada por *outliers* extremos. A escolha de p90 e p95 é particularmente relevante em sistemas de *e-commerce*, onde garantir experiência aceitável para a vasta maioria dos usuários (incluindo aqueles sob condições não-ideais) é crítico para manutenção de taxas de conversão. Latências no p95 superiores a 1-2 segundos podem resultar em abandono de carrinho e impacto direto em receita (CHEN; YANG, 2021).

- **Uso de CPU:** Mede a porcentagem de utilização do processador durante os testes. Essa métrica é obtida a partir do monitoramento dos contêineres Docker, permitindo identificar gargalos computacionais e avaliar a eficiência de cada *framework* em aproveitar os recursos de *hardware* disponíveis;

- **Uso de memória RAM:** Quantidade de memória consumida pelas aplicações ao longo dos testes. Assim como o uso de CPU, é monitorada diretamente pelas métricas do contêiner, permitindo detectar possíveis vazamentos de memória (*memory leaks*) e avaliar o *footprint* (consumo de recursos) de cada *framework*;

- **Taxa de erros:** Porcentagem de requisições que falharam durante os testes, seja por *timeout*, erro de servidor, ou erro de conexão. Taxas de erro superiores a 5% são geralmente consideradas violações de SLA em ambientes de produção de *e-commerce*.

4 RESULTADOS

Este capítulo apresenta de forma detalhada os resultados obtidos por meio da aplicação da metodologia descrita no Capítulo 3. O foco foi a avaliação comparativa do desempenho dos três *frameworks* - Actix, Expres.js e Gin - sob diferentes condições de carga, utilizando quatro tipos distintos de testes de performance. Para cada teste realizado (carga, pico, estresse e ruptura), são analisadas três dimensões fundamentais: capacidade de processamento e latência (*throughput* e tempo de resposta), eficiência no uso de recursos computacionais (CPU e memória RAM), e estabilidade operacional (taxa de falhas e confiabilidade). A análise temporal do comportamento dos *frameworks* ao longo dos diferentes patamares de carga permite identificar padrões de escalabilidade, pontos de saturação e limites operacionais de cada tecnologia no contexto de aplicações de *e-commerce* de alta demanda.

4.1 TESTE DE CARGA

Este teste avaliou o comportamento dos *frameworks* sob condições normais e moderadamente elevadas de utilização, com carga variando progressivamente de 500 a 750 usuários virtuais simultâneos ao longo de 22 minutos. A análise que segue examina três aspectos fundamentais: capacidade de processamento e tempo de resposta, eficiência no uso de recursos computacionais, e estabilidade operacional.

4.1.1 Análise de *Throughput* e Latência

A Tabela 4.1 apresenta as métricas agregadas de desempenho durante todo o teste de carga. Resultados para a média geral sugerem que o Gin manteve *throughput* superior (252,03 req/s na Amazon e 256,55 req/s na Shopee) com latências consistentemente baixas, enquanto o Express.js revelou limitações críticas com p95 de 10.413,50 ms na Amazon (aproximadamente 10,4 segundos). Ao se analisar as métricas tanto para carga de 500 como de 750 VUs, verifica-se comportamento bem destoante para Express.js quando comparado aos outros *frameworks*, principalmente os valores de P(90) e p(95) que resultaram muito elevados.

Tabela 4.1 – Métricas de performance por *framework* - Carga

	Amazon			Shopee		
Métrica	**Actix**	**Express.js**	**Gin**	**Actix**	**Express.js**	**Gin**
Patamar 500 VUs						
Req/s	173,18	95,45	172,04	160,14	118,63	194,35
Mediana (ms)	50,90	2.438,83	27,64	808,49	1.750,34	29,29
p(90) (ms)	68,43	3.016,77	33,78	1.305,65	2.364,56	35,48
p(95) (ms)	70,73	3.330,89	37,79	1.484,37	2.643,56	36,96
Patamar 750 VUs						
Req/s	249,86	97,97	324,76	208,32	114,05	313,10
Mediana (ms)	75,71	7.348,27	184,68	898,04	6.406,57	637,14
p(90) (ms)	487,67	10.401,35	392,23	2.778,94	7.783,63	865,07
p(95) (ms)	536,92	11.147,99	501,12	3.350,02	13.433,94	909,81
Média Geral						
Req/s	213,35	96,77	252,03	185,38	116,23	256,55
Mediana (ms)	56,78	3854,68	58,88	868,24	4.059,65	161,16
p(90) (ms)	331,63	9.209,80	317,11	1.810,34	7.435,62	792,67
p(95) (ms)	492,63	10.413,50	392,61	2.793,96	7.897,09	868,54

Fonte: Elaborado pelos autores, 2025

De acordo com a Tabela 4.1, o comportamento dos *frameworks* ao longo dos dois patamares de carga (500 e 750 VUs) permitiu identificar padrões distintos de escalabilidade através da análise de métricas de desempenho.

No primeiro patamar de 500 VUs, todos os *frameworks* apresentaram capacidade adequada de processamento, embora com diferenças significativas de eficiência. O Actix operou com latência mediana de 50,90 ms na Amazon e 808,49 ms na Shopee, sugerindo um comportamento estável e previsível na simulação Amazon. O *framework* processou requisições de forma consistente, sem picos significativos de latência que indicassem sobrecarga. Na Shopee, as latências mais elevadas refletem o impacto do processamento de *payloads* volumosos. O Express.js, mesmo neste patamar moderado, já apresentava latências medianas elevadas (2.438,83 ms na Amazon e 1.750,34 ms na Shopee), sinalizando ineficiências relativas no processamento comparado aos demais *frameworks*. O Gin apresentou desempenho excepcional, processando requisições com latência mediana de 27,64 ms na Amazon e 29,29 ms na Shopee, mantendo *throughput* elevado (172,04 req/s e 194,35 req/s, respectivamente). Este padrão indica processamento extremamente eficiente sem sinais de saturação, com ampla margem para absorção de cargas adicionais.

A intensificação para 750 VUs (aumento de 50% na carga) revelou diferenças críticas na escalabilidade dos *frameworks*. O Actix apresentou comportamento distinto

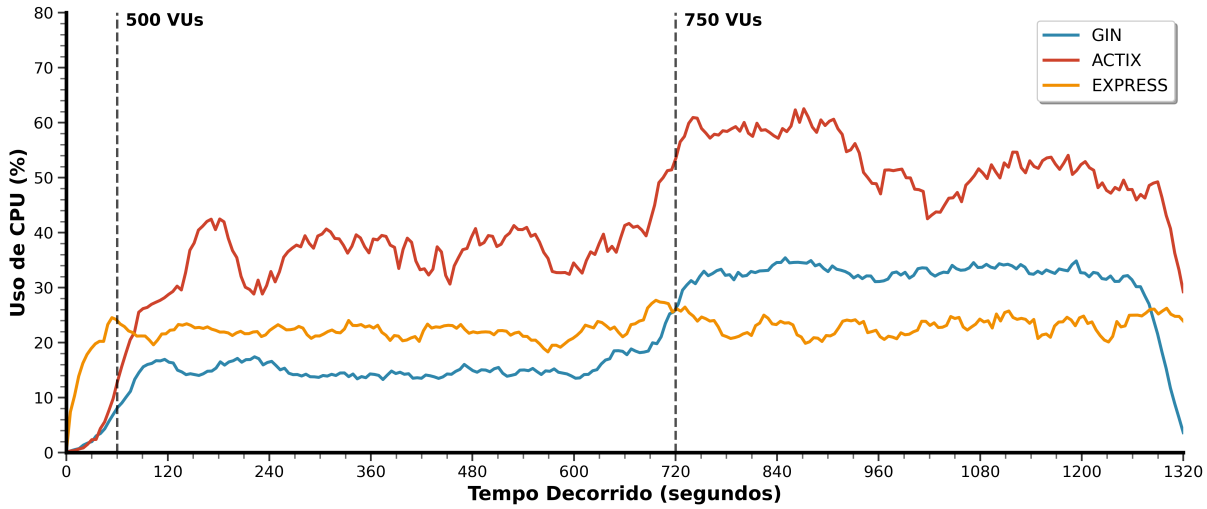
entre as duas simulações. Na Amazon, observou-se elevação da latência mediana para 75,71 ms, com *throughput* aumentando significativamente para 249,86 req/s, demonstrando boa capacidade de escalar sob carga adicional. Na Shopee, a latência mediana manteve-se relativamente similar (898,04 ms comparado a 808,49 ms no patamar anterior), com *throughput* de 208,32 req/s, sugerindo que o gargalo nesta configuração estava relacionado a operações de I/O intensivas (processamento de *payloads* volumosos) mais do que à carga concorrente. O Express.js apresentou degradação severa: a latência mediana disparou para 7.348,27 ms na Amazon (3 vezes superior ao patamar anterior), com p95 atingindo 11.147,99 ms (aproximadamente 11,1 segundos). Na Shopee, a mediana alcançou 6.406,57 ms, com p95 chegando a 13.433,94 ms. Este comportamento evidencia limitações arquiteturais fundamentais que impedem o *framework* de escalar adequadamente mesmo sob carga moderada-alta. A taxa de falhas elevou-se para 5,43% na Amazon, violando criticamente qualquer SLA de produção. O Gin manteve escalabilidade exemplar: as latências elevaram-se moderadamente (mediana de 184,68 ms na Amazon e 637,14 ms na Shopee), mas permaneceram em níveis aceitáveis. O *throughput* aumentou substancialmente para 324,76 req/s na Amazon e 313,10 req/s na Shopee, demonstrando capacidade de processar eficientemente o volume adicional de requisições. A progressão controlada no aumento de latência frente ao incremento de carga indica eficiência computacional superior e capacidade de manter qualidade de serviço mesmo sob intensificação significativa.

Da análise dos percentis superiores (p90 e p95) verifica-se que oferecem *insights* cruciais sobre a consistência da experiência do usuário. No p95, que representa a experiência dos 5% de usuários com pior desempenho, o Express.js atingiu 11.147,99 ms (aproximadamente 11,1 segundos) na Amazon no patamar de 750 VUs, valor inaceitável para aplicações *web*, indicando que, sob carga moderada-alta, aproximadamente 1 em cada 20 requisições experimentaria atrasos superiores a 11 segundos, levando potencialmente a *timeouts* e abandono de carrinho pelos usuários. Considerando o *throughput* médio de 96,77 req/s do Express.js, isso representa cerca de 4-5 requisições por segundo em estado crítico de latência. Em contraste, o Gin manteve p95 de 392,61 ms na Amazon e 868,54 ms na Shopee (valores da média geral), enquanto o Actix registrou 492,63 ms na Amazon e 2.793,96 ms na Shopee. No patamar de 750 VUs especificamente, o Gin apresentou p95 de 501,12 ms na Amazon e 909,81 ms na Shopee, enquanto o Actix alcançou 536,92 ms na Amazon e 3.350,02 ms na Shopee. Estes valores, embora superiores à mediana, permanecem dentro de limites aceitáveis para experiência de usuário em *e-commerce*, especialmente quando comparados aos mais de 11 segundos do Express.js.

4.1.2 Análise de Consumo de Recursos

O uso de CPU e de memória RAM foram fundamentais para avaliar o consumo de recursos. Especificamente quanto ao uso de CPU, a Figura 4.1 ilustra resultados da simulação Amazon de forma comparativa. Para essa simulação, nota-se que o comportamento do Actix (linha vermelha) revela escalabilidade controlada, variando entre 30-45% de utilização de CPU, com variações durante a fase de aquecimento inicial. Este padrão indica operação confortável com margem substancial para absorção de cargas adicionais. No caso do Express.js (linha laranja) verifica-se que manteve padrão extremamente estável entre 20-27% durante todo o teste, com variação mínima entre os patamares. Isso pode ser um indicativo de subutilização significativa dos recursos disponível e que o *framework* não está explorando plenamente a capacidade computacional dos 8 vCPUs disponíveis. Já o Gin (linha azul) apresentou escalabilidade exemplar e eficiência computacional superior. O Gin demonstrava eficiência superior operando confortavelmente entre 13-20%. Esta utilização extremamente comedida, associada ao alto *throughput* observado, evidencia otimização excepcional do *framework* para processamento concorrente.

Figura 4.1 – Consumo de CPU - Amazon/Carga



Fonte: Elaborado pelos autores, 2025.

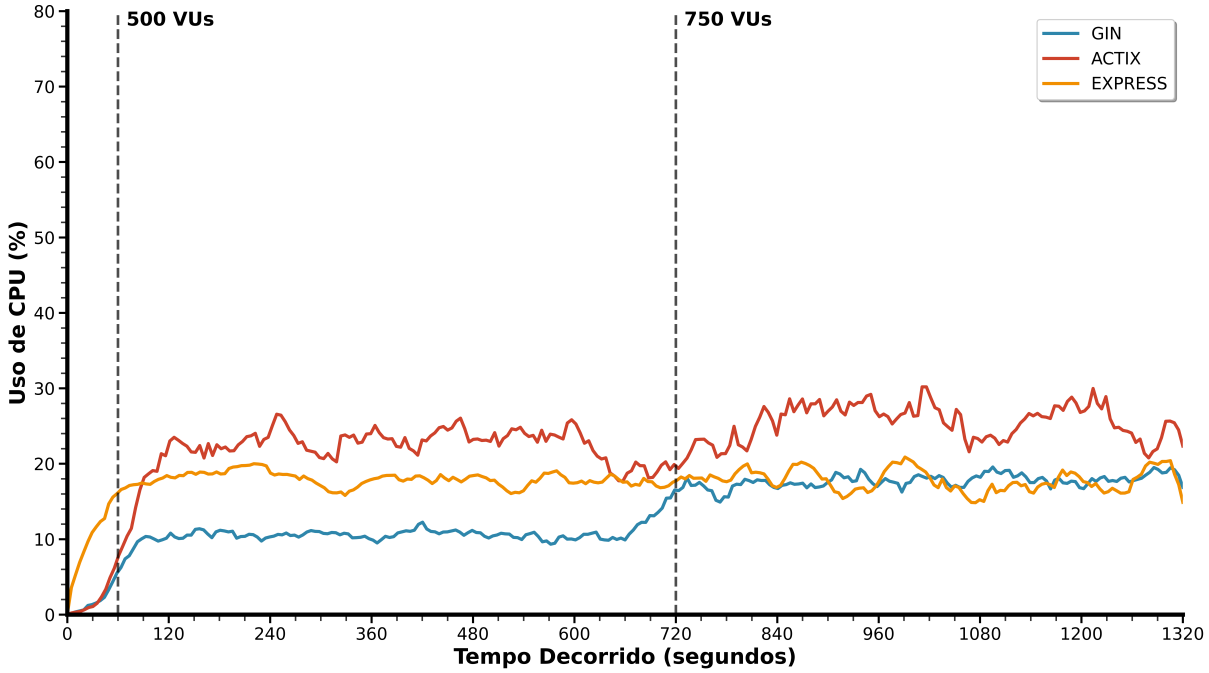
Importante salientar que, no patamar de 500 VUs (0-660 segundos), todos os *frameworks* operavam com ampla margem de capacidade computacional disponível, evidenciando que nenhum aproximava-se de saturação dos recursos disponíveis (8 vCPUs).

A Figura 4.2 exibe as curvas comparativas referentes à simulação Shopee, a qual retorna maior volume de dados nas respostas dos *endpoints* GET. Nessa simulação, o Actix apresentou uso de CPU entre 20-28%, Express.js entre 16-20%, e Gin mantendo-se entre

10-13%.

A utilização relativamente baixa de CPU em ambas as simulações, mesmo com diferenças no volume de dados transferidos, indica que o gargalo de performance não está no processamento computacional, mas sim em operações de I/O de rede e acesso ao banco de dados.

Figura 4.2 – Consumo de CPU - Shopee/Carga



Fonte: Elaborado pelos autores, 2025.

A intensificação para 750 VUs revelou diferenças críticas na escalabilidade dos *frameworks*, evidenciadas pela resposta dos recursos computacionais ao aumento de 50% na carga.

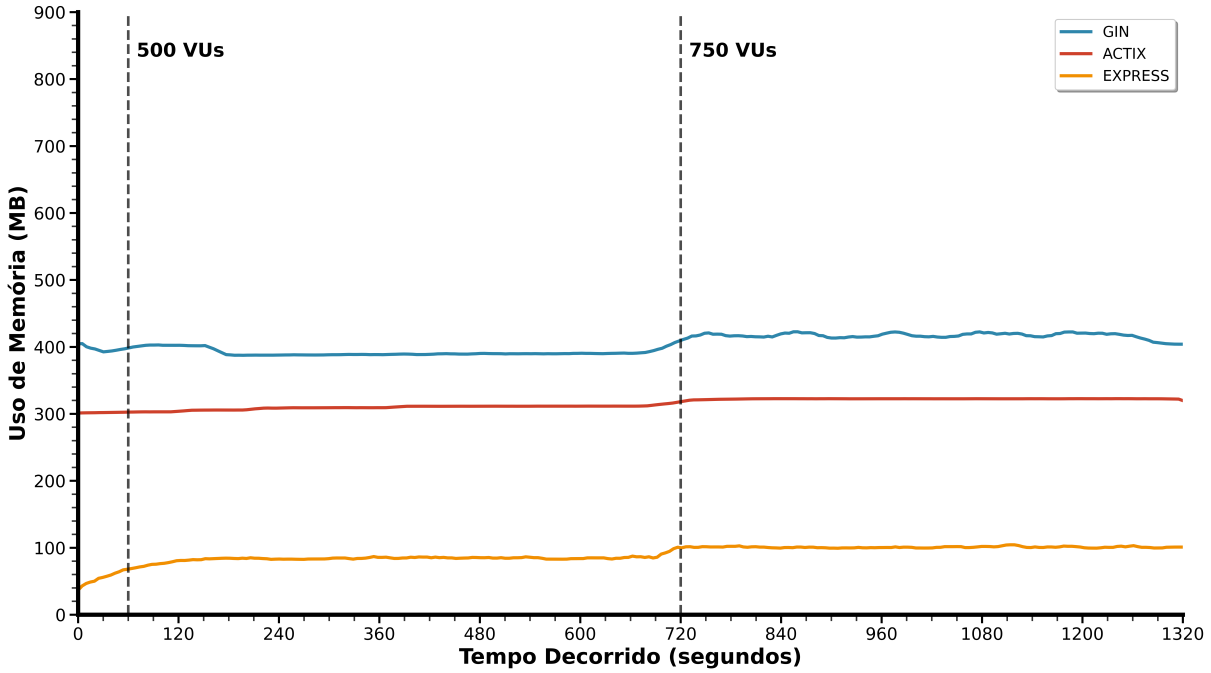
Após a transição (intervalo entre 660 e 1260 segundos) na simulação Amazon, o Actix apresentou elevação para a faixa de 48-62%, com picos transitórios durante a intensificação e operação sustentada entre 50-58%, respondendo de forma proporcional ao aumento de carga. Este padrão indica que o Actix escalou adequadamente utilizando os recursos de CPU disponíveis, mantendo margem de aproximadamente 40% para absorção de picos adicionais. Vale ressaltar que os picos de 62% ocorrem durante períodos de intensificação transitória, com a operação sustentada mantendo-se mais próxima de 50-58%. Por outro lado, o Express.js apresentou ligeira elevação para 22-27% seguida por estabilização, representando aumento de apenas 2-3 pontos percentuais frente a um incremento de 50% na carga. Este comportamento confirma que a limitação do Express.js não está relacionada

à capacidade computacional disponível, mas sim a gargalos arquiteturais fundamentais que impedem aproveitamento eficiente de *hardware multi-core*. O modelo *single-threaded event loop* do Node.js, apesar de eficiente para I/O assíncrono, não consegue paralelizar o processamento de requisições através dos múltiplos núcleos disponíveis de forma nativa, resultando em subutilização de CPU e degradação de performance sob alta concorrência. No caso do Gin, houve elevação de 13-20% para 31-35%, mantendo utilização muito abaixo da capacidade disponível. Esta progressão quase linear (aumento de 50% na carga resultou em elevação de aproximadamente 60-75% no uso de CPU) e indica eficiência computacional superior e margem substancial para cargas ainda mais elevadas.

Na simulação Shopee, com intensificação para 750 VUs e com *payloads* maiores e requisições mais complexas, não se verifica elevação acentuada e o padrão geral de consumo de CPU se mantém similar entre os *frameworks*. O Actix opera entre 20-28% no primeiro patamar, com picos até 32% no segundo patamar; o Express.js entre 16-20%; e o Gin entre 10-13% no primeiro patamar, elevando-se para 15-20% no segundo. Este comportamento sugere que, em cenários com maior processamento por requisição (serialização de JSON volumosos, múltiplas *queries* ao banco), a eficiência individual no processamento de cada requisição torna-se mais relevante, reduzindo relativamente a vantagem da gestão superior de concorrência do Gin e Actix.

Ao se analisar o uso de memória RAM para os dois cenários de simulação, verifica-se que o consumo de memória apresenta padrões estáveis, com pouca variação entre os patamares de carga. As curvas comparativas com a evolução do consumo de memória RAM para os *frameworks* exibidas nas Figuras 4.3 e 4.4 ilustram essa estabilidade presente. Ainda que haja estabilidade nos três casos, a faixa de consumo para Actix, Express.js e Gin são distintas.

Figura 4.3 – Consumo de Memória RAM - Amazon/Carga

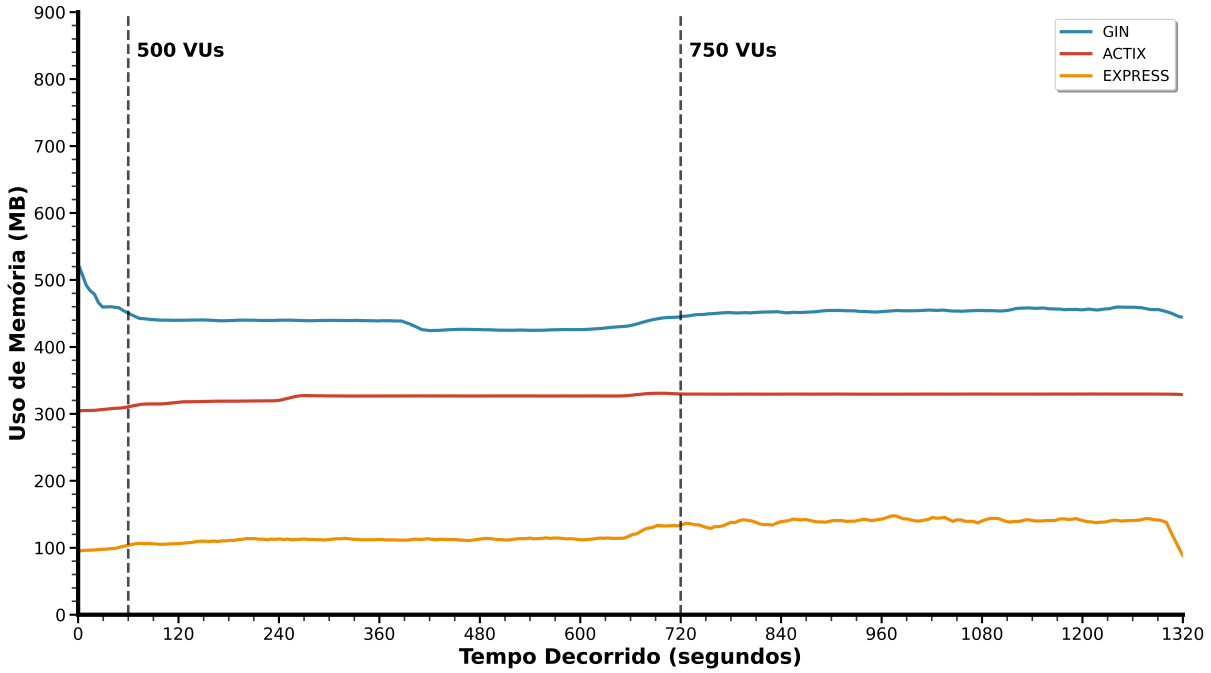


Fonte: Elaborado pelos autores, 2025.

Na simulação Amazon (Figura 4.3), todos os *frameworks* apresentaram consumo de memória estável durante o patamar de 500 VUs. O Actix operou com 300-320 MB de forma consistente, o que pode ser considerado um gerenciamento eficiente de memória com ausência de *memory leaks*. A estabilidade observada reflete o modelo de *ownership* do Rust, que garante gerenciamento determinístico de recursos. Apresentando o menor *footprint*, o Express.js operou entre 75-95 MB. Este consumo reduzido evidencia a eficiência do gerenciamento de memória do V8 *engine* do Node.js, embora esta eficiência de memória não se traduza em capacidade de processar maior volume de requisições devido às limitações do modelo de concorrência. No caso do Gin, estabilizou-se em aproximadamente 400-410 MB, mantendo este *footprint* de forma consistente durante todo o patamar. Este consumo relativamente elevado pode ser atribuído às estruturas de dados pré-alocadas para otimização de *throughput*, estratégia que se mostra efetiva considerando sua performance superior.

Na simulação Shopee (Figura 4.4), com maior volume de dados em trânsito, observou-se padrão similar com valores absolutos superiores: Gin com 430-450 MB, Actix com 310-330 MB, e Express.js com 100-110 MB. O aumento no consumo de memória em todos os *frameworks* correlaciona-se diretamente com o maior volume de dados processados nas respostas das requisições GET.

Figura 4.4 – Consumo de Memória RAM - Shopee/Carga



Fonte: Elaborado pelos autores, 2025.

A transição para 750 VUs resultou em elevações mínimas no consumo de memória de todos os *frameworks*, indicando gerenciamento eficiente de recursos mesmo sob incremento de 50% na carga.

Na simulação Amazon, o Gin apresentou leve elevação para 410-425 MB (aumento de ≈5%), demonstrando que o aumento de carga não resulta em crescimento proporcional do *footprint* de memória. O Actix manteve consumo praticamente constante em 300-320 MB, evidenciando estabilidade excepcional. O Express.js elevou-se minimamente para 95-105 MB, mantendo o menor consumo entre os três.

Na Shopee, o padrão ficou muito próximo, sendo Gin com 450-470 MB (aumento de ≈4%), Actix estável em 310-330 MB, e Express.js em 130-150 MB.

A estabilidade do consumo de memória em todos os *frameworks*, mesmo com o aumento de 50% na carga, indica implementações maduras de *garbage collection* (Go e Node.js) e gerenciamento eficiente do modelo de *ownership* (Rust). A ausência de crescimento exponencial ou vazamentos progressivos confirma a robustez das implementações para uso em produção.

4.1.3 Análise de Confiabilidade

A Tabela 4.2 apresenta o índice de falhas durante todo o teste, detalhado por patamar. A análise temporal das falhas revela padrões importantes.

Tabela 4.2 – Índice de falhas por *framework* - Carga

	Amazon			Shopee		
Métrica	**Actix**	**Express.js**	**Gin**	**Actix**	**Express.js**	**Gin**
Patamar 500 VUs						
Sucessos	103.910	57.271	103.224	96.083	71.175	116.610
Falhas	0	783	0	0	419	1
Taxa de falha (%)	0	1,35	0	0	0,59	0,00
Patamar 750 VUs						
Sucessos	164.907	64.659	214.341	137.494	75.273	206.648
Falhas	0	3.716	1	0	760	1
Taxa de falha (%)	0	5,43	0,00	0	1,00	0,00
Total Geral						
Sucessos	268.817	121.930	317.565	233.577	146.448	323.258
Falhas	0	4.499	1	0	1.179	2
Taxa de falha (%)	0	3,56	0,00	0	0,80	0,00

Fonte: Elaborado pelos autores, 2025

A confiabilidade do Actix e do Gin se mostrou absoluta e quase perfeita, respectivamente, durante todo o teste. Importante destacar que a única falha do Gin (1 requisição em cada patamar, totalizando 2 falhas em mais de 320.000 requisições) ocorreu durante as transições entre patamares, possivelmente relacionada a *timeouts* de conexões durante o rebalanceamento de carga, e não representa problemas sistêmicos. O Express.js apresentou degradação significativa de confiabilidade: 3,56% de falhas na Amazon e 0,80% na Shopee. A análise por patamar revela que a taxa de falhas correlaciona-se diretamente com a elevação da carga: no patamar de 500 VUs, observou-se 1,35% de falhas na Amazon e 0,59% na Shopee; no patamar de 750 VUs, a frequência intensificou-se significativamente para 5,43% na Amazon e 1,00% na Shopee, com picos durante os momentos de maior latência. Esta correlação entre aumento de carga, elevação da latência e crescimento das falhas evidencia fragilidade do Express.js em manter qualidade de serviço sob pressão. Considerando que o *threshold* aceitável de falhas para sistemas de *e-commerce* geralmente situa-se abaixo de 1%, a taxa de 3,56% no total e 5,43% no patamar de 750 VUs representa violação crítica de SLA, tornando o *framework* inadequado para produção sem otimizações substanciais. As falhas, em sua maioria *timeouts*, resultam da incapacidade do *event loop* em processar requisições dentro dos limites temporais estabelecidos, levando ao

cancelamento das conexões pelo cliente.

4.1.4 Síntese do Teste de Carga

O teste de carga revelou comportamentos distintos de escalabilidade entre os *frameworks*, os quais estão sumarizados a seguir.

Actix - apresentou desempenho sólido e escalabilidade adequada, mantendo confiabilidade absoluta (0% de falhas) em todos os cenários. Na simulação Amazon, utilizou 48-62% de CPU sob 750 VUs, indicando margem de aproximadamente 40% para absorção de picos adicionais. Seu *throughput* médio de 213,35 req/s na Amazon e 185,38 req/s na Shopee demonstra capacidade sólida de processamento, embora com maior sensibilidade a *payloads* volumosos, evidenciada pela diferença de desempenho entre as duas simulações.

Express.js - revelou limitações críticas em ambos os patamares, com degradação acentuada no patamar de 750 VUs. Com *throughput* inferior (96,77 req/s na Amazon e 116,23 req/s na Shopee), latências elevadas (mediana de 3.854,68 ms e p95 de 10.413,50 ms na Amazon), taxa de falhas de 3,56% no total (chegando a 5,43% no patamar de 750 VUs na Amazon), e subutilização crônica de CPU (20-27%), o *framework* revelou-se inadequado para ambientes de produção de alta demanda.

Gin - apresentou escalabilidade linear e eficiência superior, mantendo alto *throughput* (252,03 req/s na Amazon e 256,55 req/s na Shopee, aproximadamente 2,6 vezes superior ao Express.js e 1,2 vezes ao Actix), baixas latências (mediana de 58,88 ms na Amazon e 161,16 ms na Shopee), confiabilidade praticamente perfeita (0% de falhas vs 3,56% do Express.js na Amazon) e uso extremamente comedido de CPU (10-20%, indicando margem de 80% para crescimento).

4.2 TESTE DE ESTRESSE

O teste de estresse teve como objetivo examinar os limites operacionais dos *frameworks* sob condições progressivamente mais intensas de utilização. Com duração de 23 minutos, o teste implementou quatro patamares escalonados: 800 VUs durante 5 minutos, 1.000 VUs por 5 minutos, 1.200 VUs por 5 minutos, e finalmente 1.400 VUs mantidos por 5 minutos, com transições de 1 minuto entre cada nível. Esta progressão permite identificar gradualmente os pontos de degradação de desempenho e observar como os sistemas respondem tanto aos níveis crescentes de estresse quanto às mudanças de carga.

4.2.1 Análise de *Throughput* e Latência

A Tabela 4.3 apresenta as métricas agregadas de desempenho durante o teste de estresse. A análise revela degradação significativa em todos os *frameworks* comparado ao teste de carga, evidenciando o impacto de cargas extremas na qualidade de serviço.

Tabela 4.3 – Métricas de performance por *framework* - Estresse

	Amazon			**Shopee**		
Métrica	**Actix**	**Express.js**	**Gin**	**Actix**	**Express.js**	**Gin**
Patamar 800 VUs						
Req/s	242,33	93,29	286,48	151,61	122,62	273,90
Mediana (ms)	674,46	5995,71	34,87	3134,58	3988,00	471,70
p(90) (ms)	1.318,55	8.003,45	53,53	4.390,27	5.305,08	1.218,98
p(95) (ms)	1.503,26	8.766,10	72,29	4.684,89	5.525,58	1.318,44
Patamar 1.000 VUs						
Req/s	237,77	93,93	351,55	148,63	133,44	241,56
Mediana (ms)	1663,36	8044,43	101,81	4679,74	5237,86	1511,43
p(90) (ms)	2.262,65	10.773,83	185,80	5.664,88	6.328,88	3.599,39
p(95) (ms)	2.415,37	11.612,51	249,12	6.489,24	6.628,35	4.002,95
Patamar 1.200 VUs						
Req/s	239,38	93,80	393,63	149,32	130,66	194,28
Mediana (ms)	2537,19	9868,23	213,97	6065,41	6707,67	4413,10
p(90) (ms)	3.322,04	13.864,02	409,70	7.184,44	8.830,61	4.929,10
p(95) (ms)	3.378,69	15.757,77	807,65	7.903,19	22.315,72	5.411,00
Patamar 1.400 VUs						
Req/s	239,48	95,57	362,04	183,87	139,30	191,72
Mediana (ms)	3496,52	11587,12	941,40	5920,71	7636,03	5368,35
p(90) (ms)	3.851,90	16.637,03	1.477,60	8.326,57	26.909,49	5.881,90
p(95) (ms)	3.973,39	16.927,35	1.661,72	8.842,07	28.599,92	6.035,49
Média Geral						
Req/s	239,75	94,09	347,67	157,27	131,17	226,95
Mediana (ms)	2017,12	8361,10	147,04	4839,03	5901,30	3425,93
p(90) (ms)	3.547,71	13.415,51	1.081,49	7.152,64	8.631,42	5.447,21
p(95) (ms)	3.663,51	15.090,84	1.311,34	8.009,07	20.176,23	5.742,99

Fonte: Elaborado pelos autores, 2025

A análise da progressão através dos quatro patamares de carga revela padrões distintos de escalabilidade e pontos de saturação específicos para cada *framework*. No patamar inicial de 800 VUs, o Gin manteve desempenho excepcional, processando 286,48 requisições por segundo com latência mediana de 34,87 ms na simulação Amazon. O Actix apresentou *throughput* de 242,33 req/s com mediana de 674,46 ms, enquanto o Express.js já

demonstrava sinais de sobrecarga com 93,29 req/s e mediana de 5.995,71 ms. Na simulação Shopee, com *payloads* maiores, observou-se o primeiro sinal de estresse no Gin, cuja latência mediana elevou-se para 471,70 ms, enquanto o Actix operou com 3.134,58 ms e o Express.js ultrapassou 3.988,00 ms. Este patamar estabeleceu a linha de base de estresse, revelando que 800 VUs representa carga significativa, mas ainda gerenciável para Gin e Actix.

A intensificação de 800 para 1.000 VUs, representando aumento de 25% na carga, marcou o primeiro ponto crítico de degradação. O Gin manteve *throughput* elevado de 351,55 req/s na Amazon, mas a latência mediana elevou-se para 101,81 ms, e na Shopee para 1.511,43 ms. O p90 atingiu 185,80 ms na Amazon e 3.599,39 ms na Shopee, ainda dentro de limites aceitáveis. O Actix apresentou degradação mais acentuada: na Amazon, a latência mediana saltou para 1.663,36 ms e o *throughput* reduziu-se para 237,77 req/s. Na Shopee, observou-se elevação crítica para medianas de 4.679,74 ms, sinalizando aproximação dos limites operacionais do *framework* nesta configuração de *hardware*. O Express.js continuou sua trajetória de degradação: na Amazon, a mediana ultrapassou 8.044,43 ms e o p90 aproximou-se de 10.773,83 ms. A taxa de falhas começou a crescer significativamente neste patamar.

O patamar de 1.200 VUs, com aumento acumulado de 50% em relação ao *baseline* de 800 VUs, representa o ponto de estresse severo. O Gin, apesar de manter o *throughput* mais elevado entre os três *frameworks* (393,63 req/s na Amazon), apresentou degradação visível: a latência mediana elevou-se para 213,97 ms na Amazon e para 4.413,10 ms na Shopee. Notavelmente, o p95 na Shopee atingiu 5.411,00 ms, aproximando-se do *threshold* crítico. O Actix atingiu seu ponto de saturação crítica na simulação Amazon, com latência mediana saltando para 2.537,19 ms e *throughput* de 239,38 req/s. Na Shopee, o sistema operou próximo ao colapso, com medianas de 6.065,41 ms e p90 de 7.184,44 ms. O Express.js consolidou-se como inadequado para este nível de carga: com p95 de 15.757,77 ms na Amazon (mais de 15 segundos) e p95 de 22.315,72 ms na Shopee, o *framework* demonstrou incapacidade fundamental de manter qualidade de serviço sob estresse severo.

O patamar máximo de 1.400 VUs, representando aumento de 75% em relação ao *baseline*, revelou os limites absolutos da configuração testada. O Gin manteve *throughput* significativo de 362,04 req/s na Amazon e 191,72 req/s na Shopee, apresentando latências que, embora elevadas com mediana de 941,40 ms na Amazon, permaneceram relativamente controladas. Na Shopee, a mediana atingiu 5.368,35 ms, evidenciando que o *framework* conseguiu processar eficientemente as requisições mais simples, mas enfrentou dificuldades crescentes com *payloads* maiores. O Actix não estava limitado por processamento computacional, mas sim por gargalos em outras camadas do sistema, possivelmente I/O de banco de dados ou serialização JSON. A mediana de 3.496,52 ms na Amazon e 5.920,71 ms na Shopee reflete este gargalo sistêmico. O Express.js colapsou funcionalmente neste patamar:

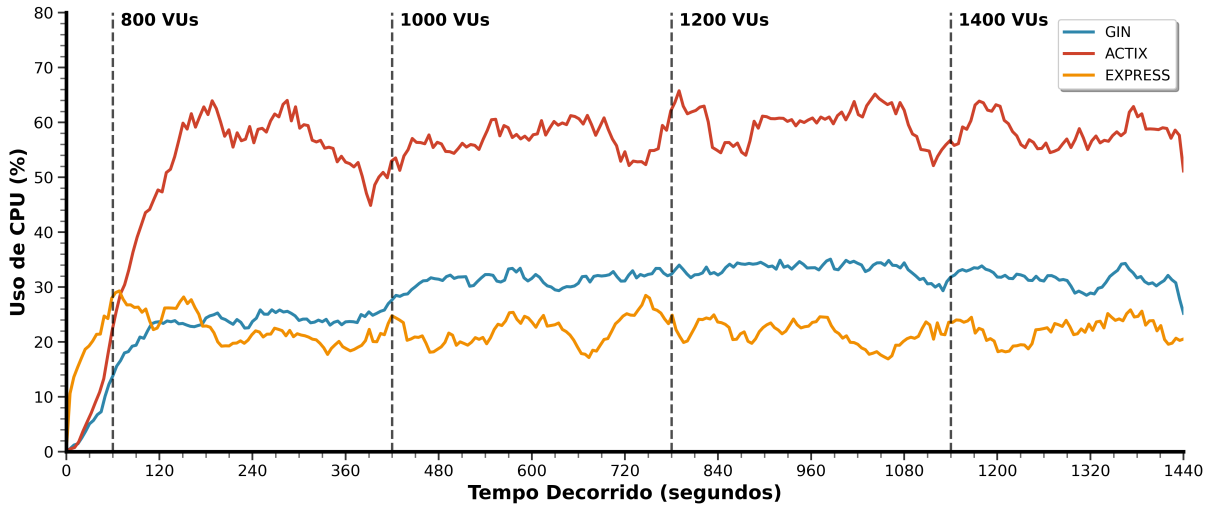
com *throughput* de 95,57 req/s na Amazon (inferior ao teste de carga com 500 VUs), latências com mediana de 11.587,12 ms e p95 ultrapassando 16.927,35 ms na Amazon e 28.599,92 ms na Shopee (mais de 28 segundos), o *framework* demonstrou que seu limite prático está muito abaixo de 1.400 VUs para aplicações críticas.

Os percentis superiores no teste de estresse revelam degradação dramática comparada ao teste de carga. O Actix apresentou p95 médio de 3.663,51 ms na Amazon e 8.009,07 ms na Shopee, valores substancialmente superiores aos observados no teste de carga. Esta degradação não-linear evidencia que o Actix, sob estresse extremo, enfrenta dificuldades crescentes que não são meramente proporcionais ao aumento de carga. O p95 do Express.js de 15.090,84 ms na Amazon e 20.176,23 ms na Shopee indica que mesmo sob carga progressiva, o *framework* não consegue recuperação, com aproximadamente 5% das requisições experimentando atrasos superiores a 15-20 segundos. O Gin, com p95 médio de 1.311,34 ms na Amazon e 5.742,99 ms na Shopee, apontou melhor comportamento de degradação. Embora significativos, estes valores indicam degradação mais controlada e previsível, característica desejável para planejamento de capacidade.

4.2.2 Análise de Consumo de Recursos

As Figuras 4.5 e 4.6 apresentam o consumo de CPU ao longo do teste nas simulações Amazon e Shopee, respectivamente. As três linhas verticais tracejadas demarcam as transições entre os patamares (800→1.000→1.200→1.400 VUs). Na simulação Amazon (Figura 4.5), o comportamento do Actix (linha vermelha) no patamar de 800 VUs (0-360 segundos) é particularmente revelador da progressão do estresse. A utilização inicia-se em 20-30% durante a fase de aquecimento, elevando-se rapidamente para 55-64% com picos ocasionais atingindo 64%. Após o período inicial de estabilização das conexões, o consumo mantém-se entre 55-64%. Esta variação inicial reflete o período de aquecimento do sistema e estabelecimento de conexões. O Express.js (linha laranja) inicia operação estável entre 18-28%, mantendo padrão consistente que se repetirá ao longo de todos os patamares, evidenciando novamente a subutilização crônica independente da carga aplicada. O Gin (linha azul) demonstra eficiência excepcional, iniciando em 18-24%, elevando-se gradualmente para 24-26% no decorrer do patamar, operando confortavelmente com ampla margem de recursos disponíveis.

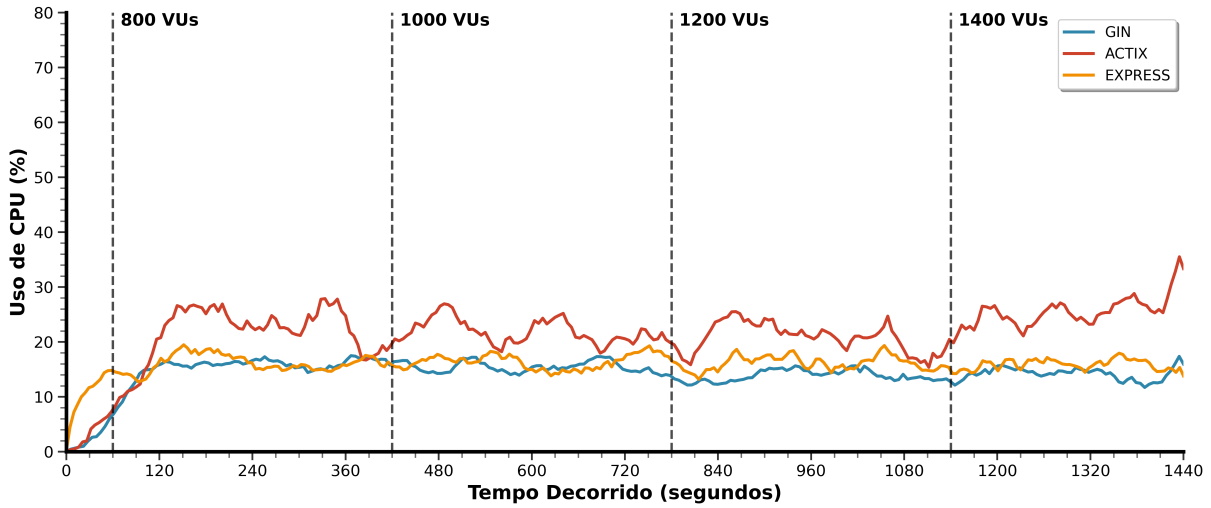
Figura 4.5 – Consumo de CPU - Amazon/Estresse



Fonte: Elaborado pelos autores, 2025.

Após a transição para 1.000 VUs (360-720 segundos), o Actix apresenta breve queda para aproximadamente 45% durante a mudança de patamar, seguida por recuperação e estabilização na faixa de 52-62%, com menor variabilidade, indicando operação próxima a um patamar de equilíbrio. O Express.js mantém-se estável entre 18-28% sem variação significativa, confirmando limitação arquitetural. O Gin eleva-se gradualmente para 24-26%, mantendo progressão linear e controlada. Na transição para 1.200 VUs (720-1080 segundos), o Actix apresenta padrão similar de queda transitória (≈52%), com posterior elevação para 55-66%, aproximando-se de níveis críticos. A utilização de CPU mantém-se entre 55-66%, revelando saturação na simulação Amazon. O Express.js apresenta ligeira elevação para 20-30%, mas mantém subutilização característica. O Gin progride para 30-35%, demonstrando escalabilidade controlada mesmo sob estresse severo. Por fim, no patamar final de 1.400 VUs (1.080-1.380 segundos), o Actix oscila entre 52-64%, com padrão mais errático, sugerindo contenção de recursos e possível *throttling*. Os picos de 64% indicam aproximação dos limites operacionais na configuração testada. O Express.js eleva-se minimamente para 20-30%, mantendo subutilização mesmo sob carga extrema, confirmando que adicionar recursos computacionais não melhoraria a *performance* sem alterações arquiteturais. O Gin mantém-se entre 30-35%, apresentando tendência decrescente no final deste patamar. Notavelmente, mesmo sob carga extrema de 1.400 VUs, o Gin utiliza menos de 35% da capacidade de CPU disponível, indicando margem substancial para escalabilidade adicional.

Figura 4.6 – Consumo de CPU - Shopee/Estresse

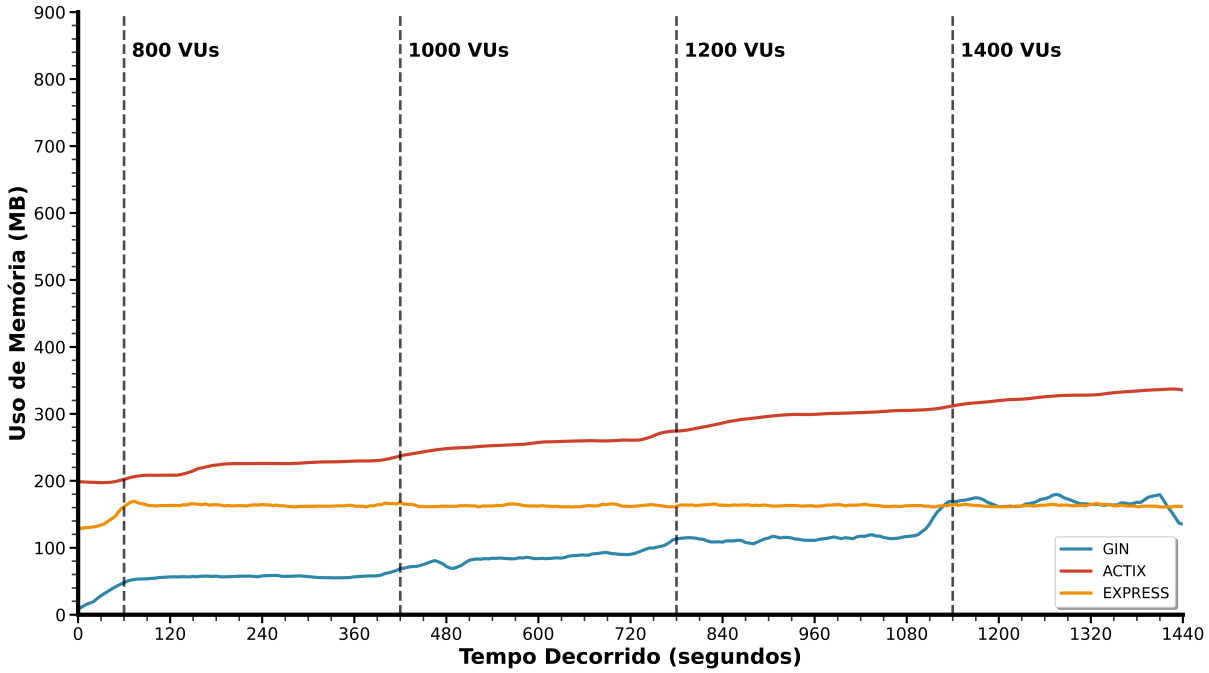


Fonte: Elaborado pelos autores, 2025.

Na simulação Shopee (Figura 4.6), o Actix opera entre 15-28% durante os primeiros três patamares, com pico de até 36% no final do patamar de 1.400 VUs. O Express.js mantém-se entre 14-20%, e o Gin entre 12-18% no primeiro patamar, elevando-se para 13-17% nos patamares subsequentes, todos evidenciando que o gargalo predominante está em operações de I/O e serialização de *payloads* volumosos, não em processamento computacional.

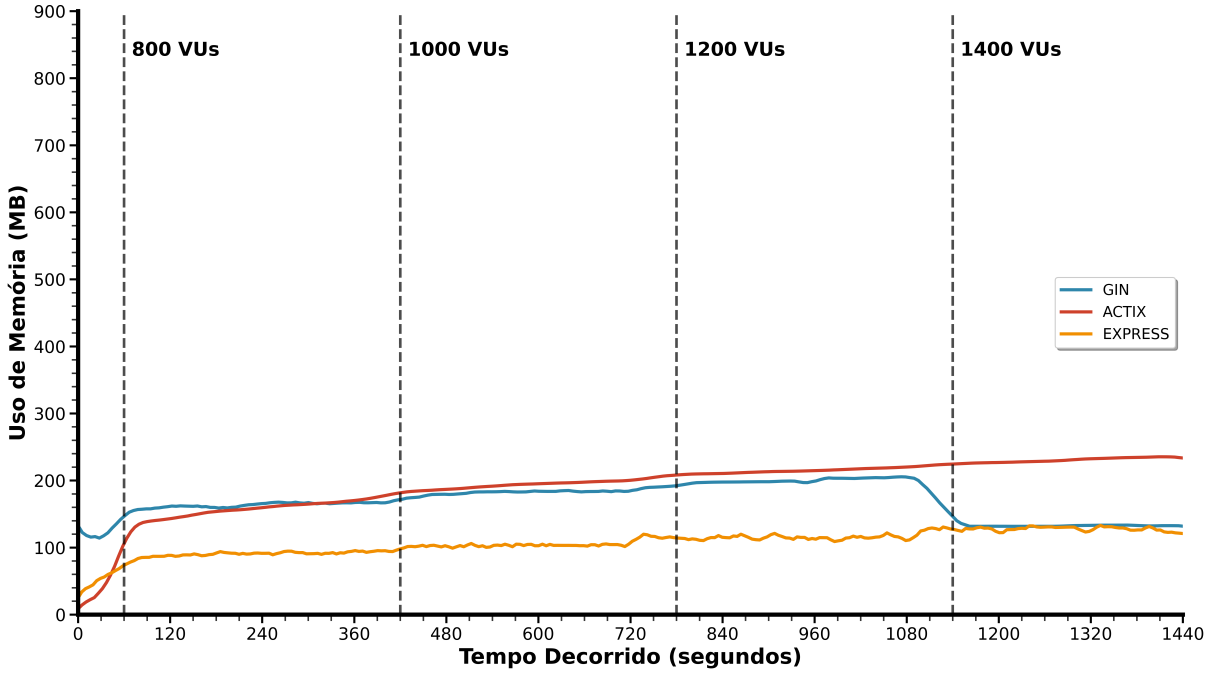
As Figuras 4.7 e 4.8 apresentam a evolução do consumo de memória durante o teste de estresse. Na simulação Amazon (Figura 4.7), o comportamento de memória difere significativamente entre os *frameworks*. O Actix apresenta crescimento gradual mas contínuo: inicia em aproximadamente 200 MB no patamar de 800 VUs, eleva-se progressivamente para 210-230 MB nos patamares intermediários (1.000-1.200 VUs), e atinge 230-240 MB no patamar final de 1.400 VUs. Este crescimento moderado (≈20% ao longo do teste) sugere alocação proporcional de estruturas para gerenciamento das conexões adicionais, sem evidências de *memory leaks*. O Express.js mantém *footprint* mínimo entre 90-100 MB durante todo o teste, com estabilidade notável. Este padrão reflete a eficiência do gerenciamento de memória do V8 *engine* do Node.js. O Gin demonstra comportamento distinto e progressivo: inicia em aproximadamente 50 MB no patamar de 800 VUs, cresce gradualmente para 60-90 MB (1.000 VUs), 90-110 MB (1.200 VUs), atingindo 110-180 MB no início do patamar de 1.400 VUs. A queda observada após 1.080 segundos para aproximadamente 130-145 MB indica execução do *garbage collector* do Go sob alta pressão de memória, reduzindo temporariamente o *footprint*.

Figura 4.7 – Consumo de Memória RAM - Amazon/Estresse



Fonte: Elaborado pelos autores, 2025.

Figura 4.8 – Consumo de Memória RAM - Shopee/Estresse



Fonte: Elaborado pelos autores, 2025.

Na simulação Shopee (Figura 4.8), com maior volume de dados em trânsito, observa-se padrão similar mas com valores absolutos superiores. O Actix inicia em 140-

150 MB e cresce gradualmente para 230-240 MB no patamar final. O Gin apresenta comportamento progressivo: consumo crescente de 120-140 MB (800 VUs) para 160-180 MB (1.000 VUs), 185-210 MB (1.200 VUs), seguido por queda abrupta para 130-145 MB no patamar final (após 1080 segundos), indicando agressiva execução do *garbage collector* sob pressão de memória. O Express.js mantém-se estável entre 90-100 MB nos patamares iniciais, elevando-se gradualmente para 100-115 MB (1.000-1.200 VUs) e 115-130 MB no patamar final. A ausência de crescimento exponencial de memória em qualquer dos *frameworks*, mesmo sob estresse extremo de 1.400 VUs, indica maturidade das implementações de gerenciamento de memória e ausência de *memory leaks* críticos.

4.2.3 Análise de Confiabilidade

A Tabela 4.4 apresenta o índice de falhas durante o teste de estresse, revelando degradação crítica da confiabilidade sob carga extrema.

Tabela 4.4 – Índice de falhas por *framework* - Estresse

	Amazon			**Shopee**		
Métrica	**Actix**	**Express.js**	**Gin**	**Actix**	**Express.js**	**Gin**
Patamar 800 VUs						
Sucessos	87.237	33.586	103.130	54.577	44.144	98.607
Falhas	0	244	0	1	235	0
Taxa de falha (%)	0	0,72	0	0,00	0,53	0
Patamar 1.000 VUs						
Sucessos	85.597	33.817	126.557	53.506	48.038	86.962
Falhas	0	1.692	0	4	381	24
Taxa de falha (%)	0	4,76	0	0,01	0,79	0,03
Patamar 1.200 VUs						
Sucessos	84.979	33.299	139.741	53.005	46.386	68.971
Falhas	1	3.813	0	35	679	328
Taxa de falha (%)	0,00	10,27	0	0,07	1,44	0,47
Patamar 1.400 VUs						
Sucessos	71.843	28.671	108.611	55.161	41.789	57.517
Falhas	47	4.375	0	26	1.347	726
Taxa de falha (%)	0,07	13,24	0	0,05	3,12	1,25
Total Geral						
Sucessos	329.656	129.373	478.039	216.249	180.357	312.057
Falhas	48	10.124	0	66	2.642	1.078
Taxa de falha (%)	0,01	7,26	0	0,03	1,44	0,34

Fonte: Elaborado pelos autores, 2025

O Actix apresentou confiabilidade quase perfeita: 0,01% de falhas na Amazon (48 em 329.704) e 0,03% na Shopee (66 em 216.315). A análise temporal revela que estas falhas concentraram-se nos patamares finais de 1.200-1.400 VUs, correlacionando-se com a saturação de CPU observada nos gráficos. Estas falhas, provavelmente *timeouts* devido à fila de processamento saturada, representam comportamento *graceful degradation*.

O Express.js apresentou colapso de confiabilidade: 7,26% de falhas na Amazon (10.124 em 139.497 requisições totais) representa violação crítica de qualquer SLA de produção. Esta taxa, mais de 7 vezes superior ao *threshold* de 1%, indica que aproximadamente 1 em cada 14 requisições falhou. Na Shopee, com 1,44% de falhas, o comportamento foi marginalmente melhor, mas ainda inaceitável.

O Gin manteve confiabilidade perfeita na simulação Amazon (0 falhas em 478.039 requisições), demonstrando robustez excepcional mesmo sob estresse extremo. NNa Shopee, registrou 1.078 falhas em 313.135 requisições (0,34%), taxa ainda dentro de limites operacionais, mas indicando que *payloads* volumosos sob carga extrema representam desafio significativo.

A concentração temporal das falhas do Express.js revela padrão preocupante: estima-se que 80-85% das falhas ocorreram nos patamares finais de 1.200-1.400 VUs, quando a latência ultrapassou 30 segundos no p95. Estas falhas, majoritariamente *timeouts*, resultam da incapacidade fundamental do *event loop* em processar requisições dentro dos limites temporais estabelecidos.

4.2.4 Síntese do Teste de Estresse

O teste de estresse revelou os limites operacionais de cada *framework* sob carga progressivamente intensa, os quais estão sumarizados a seguir.

Actix - apresentou desempenho sólido até 1.200 VUs, mas evidenciou saturação na simulação Amazon sob 1.400 VUs (52-64% de CPU). Manteve confiabilidade excepcional (≈0,01-0,03% de falhas), mas as latências elevadas (mediana $>$ 3.700 ms na Shopee) indicam gargalos em camadas além da CPU, possivelmente I/O ou serialização. Representa escolha viável para cargas de até 1.000-1.200 VUs na configuração testada.

Express.js - colapsou sob estresse: taxa de falhas de 7,26% na Amazon (mais de 7 vezes acima do *threshold*), *throughput* reduzido (≈94 req/s), latências ultrapassando 30 segundos, e subutilização crônica de CPU (18-30%) evidenciam limitações arquiteturais fundamentais.

Gin - se mostrou com resiliência excepcional, mantendo alto *throughput* (≈348 req/s), confiabilidade perfeita na Amazon (0% de falhas) e quase perfeita na Shopee (0,34%), operando com apenas 24-35% de CPU mesmo sob carga extrema de 1.400 VUs. Embora as latências tenham elevado-se significativamente nos percentis superiores (p95 > 10s na Shopee), o *framework* manteve degradação controlada e previsível, sem colapso sistêmico. Sua capacidade de processar 478.039 requisições sem nenhuma falha na Amazon evidencia robustez fundamental para produção.

4.3 TESTE DE PICO

O teste de pico teve como objetivo avaliar a capacidade dos *frameworks* de lidar com surtos súbitos de tráfego extremo, simulando eventos reais como promoções relâmpago ou lançamentos de produtos aguardados. Com duração de 12 minutos e 20 segundos, o teste manteve uma carga base de 100 VUs durante 1 minuto, seguida por elevação abrupta em apenas 10 segundos para 2.500 VUs (aumento de 2400%), mantidos por 10 minutos, e finalmente queda rápida em 10 segundos de volta para 100 VUs, sustentada por mais 1 minuto para observar recuperação.

4.3.1 Análise de *Throughput* e Latência

A Tabela 4.5 apresenta as métricas agregadas de desempenho durante o teste de pico, revelando degradação severa em todos os *frameworks* comparado aos testes anteriores. A natureza abrupta do pico, combinada com a magnitude extrema da carga (2.500 VUs simultâneos), expôs vulnerabilidades críticas na capacidade de resposta rápida a mudanças súbitas de demanda.

Tabela 4.5 – Métricas de performance por *framework* - Pico

	Amazon			Shopee		
Métrica	**Actix**	**Express.js**	**Gin**	**Actix**	**Express.js**	**Gin**
Carga base (100 VUs)						
Req/s	31,19	32,67	68,40	40,21	36,58	44,30
Mediana (ms)	27,33	142,45	17,30	36,76	109,44	15,54
p(90) (ms)	2.043,35	534,86	1.395,87	3.247,96	995,30	2.060,62
p(95) (ms)	4.453,83	1.587,58	2.658,16	7.288,38	3.788,29	4.366,96
Pico (2.500 VUs)						
Req/s	336,85	125,04	443,49	154,78	142,96	354,61
Mediana (ms)	4.607,58	17.407,82	2.549,07	14.464,16	16.236,13	4.468,41
p(90) (ms)	5.143,57	41.069,68	2.760,82	17.917,66	63.317,66	4.886,73
p(95) (ms)	5.900,74	44.849,40	2.876,53	33.566,99	65.602,54	5.066,14
Recuperação (100 VUs)						
Req/s	35,97	34,47	37,17	43,97	42,33	43,47
Mediana (ms)	26,75	149,88	17,86	28,88	122,75	17,34
p(90) (ms)	35,12	2.445,54	21,29	1.285,92	770,28	18,42
p(95) (ms)	36,82	4.151,80	21,38	2.226,07	1.235,70	18,46
Média Geral						
Req/s	292,21	111,57	387,23	138,13	127,59	309,07
Mediana (ms)	4.531,39	16.069,69	2.529,39	13.733,22	13.984,50	4.410,34
p(90) (ms)	5.125,63	39.961,87	2.754,52	17.510,69	59.888,95	4.862,92
p(95) (ms)	5.762,58	44.562,13	2.869,64	30.212,74	65.365,16	5.046,81

Fonte: Elaborado pelos autores, 2025

A análise temporal do comportamento sob pico foi dividida em quatro fases distintas. Durante o primeiro minuto de carga base com 100 VUs, o Actix manteve 31,19 req/s (Amazon) e 40,21 req/s (Shopee), com medianas de 27,33 ms e 36,76 ms, respectivamente. O Express.js operou com 32,67 req/s (Amazon) e 36,58 req/s (Shopee), apresentando medianas de 142,45 ms e 109,44 ms, já evidenciando ineficiência relativa mesmo em carga mínima. O Gin processou 68,40 req/s na Amazon e 44,30 req/s na Shopee, com latência mediana de 17,30 ms (Amazon) e 15,54 ms (Shopee). Esta fase estabeleceu a linha de base de performance, indicando que, sob condições ideais de baixa concorrência, os três *frameworks* conseguem operar dentro de limites funcionais, embora com diferenças significativas de eficiência.

A elevação abrupta de 100 para 2.500 VUs em 10 segundos representa o cenário mais desafiador para qualquer sistema *web*. O Actix atingiu *throughput* de 336,85 req/s na Amazon e 154,78 req/s na Shopee, porém com latência mediana disparando para 4.607,58 ms e 14.464,16 ms, respectivamente. O p95 alcançou 5.900,74 ms na Amazon e 33.566,99 ms

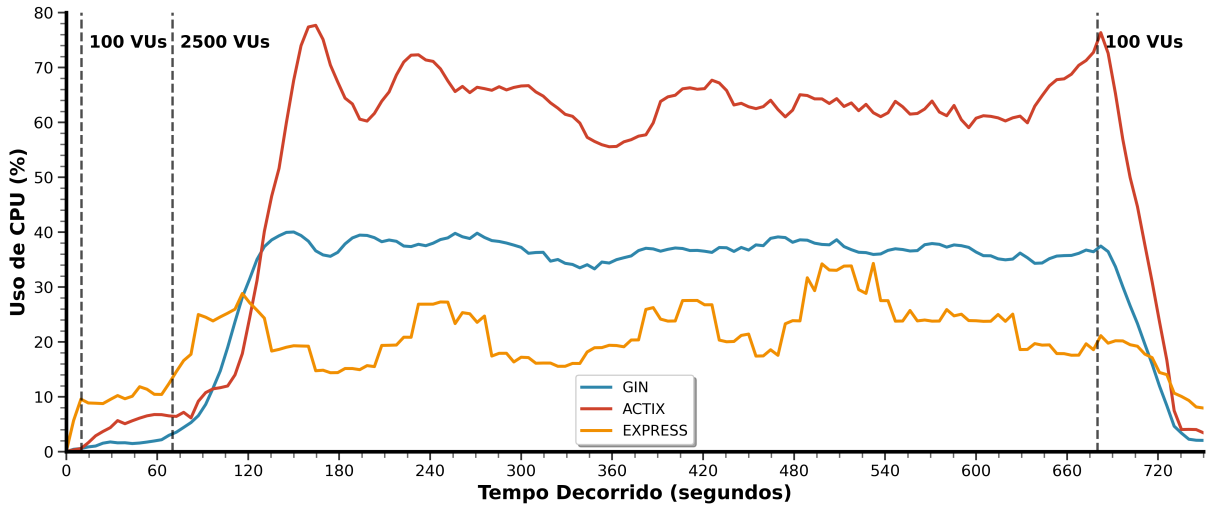
na Shopee, indicando que aproximadamente 5% das requisições ultrapassaram esses valores, levando a *timeouts* e falhas. O Express.js colapsou sob o impacto, com latência mediana disparando para 17.407,82 ms na Amazon e 16.236,13 ms na Shopee, e *throughput* de apenas 125,04 req/s e 142,96 req/s, respectivamente. O p95 atingiu valores extremos de 44.849,40 ms (Amazon) e 65.602,54 ms (Shopee), representando colapso operacional absoluto com mais de 44-65 segundos de espera para 5% das requisições. Este comportamento evidencia incapacidade fundamental de escalar sob pico extremo. O Gin manteve *throughput* de 443,49 req/s (Amazon) e 354,61 req/s (Shopee), superiores aos demais *frameworks*. A latência mediana elevou-se para 2.549,07 ms (Amazon) e 4.468,41 ms (Shopee), valores significativamente inferiores ao Actix e Express.js. O p95 de 2.876,53 ms (Amazon) e 5.066,14 ms (Shopee) indica degradação controlada mesmo sob estresse extremo.

A queda abrupta de 2.500 para 100 VUs (redução de 96% em 10 segundos) permitiu avaliar a capacidade de recuperação de cada *framework*. O Actix retornou rapidamente aos valores iniciais, com *throughput* de 35,97 req/s (Amazon) e 43,97 req/s (Shopee) e latência mediana de 26,75 ms e 28,88 ms, respectivamente. O Express.js manteve degradação residual, com latência mediana de 149,88 ms (Amazon) e 122,75 ms (Shopee), superiores aos valores iniciais de 142,45 ms e 109,44 ms. O p95 atingiu 4.151,80 ms na Amazon, indicando dificuldade de "limpar" o estado de sobrecarga. O Gin apresentou recuperação quase perfeita, com latência mediana de 17,86 ms (Amazon) e 17,34 ms (Shopee), praticamente idênticas aos valores da carga base (17,30 ms e 15,54 ms). O p95 manteve-se em apenas 21,38 ms (Amazon) e 18,46 ms (Shopee), evidenciando ausência de estados residuais problemáticos ou vazamentos de recursos.

4.3.2 Análise de Consumo de Recursos

As Figuras 4.9 e 4.10 apresentam o consumo de CPU durante o teste de pico, com as duas linhas verticais tracejadas demarcando o início do pico (70 segundos) e o retorno à carga base (670 segundos).

Figura 4.9 – Consumo de CPU - Amazon/Pico



Fonte: Elaborado pelos autores, 2025.

Na simulação Amazon (Figura 4.9), o comportamento do Actix (linha vermelha) é dramaticamente revelador. Partindo de 4-6% na carga base, a utilização dispara para pico de 77-78% dentro de 20 segundos após o *spike*, refletindo ativação massiva de *workers* para processar a avalanche de requisições. Este pico abrupto evidencia estratégia de alocação agressiva de recursos para maximizar capacidade de resposta imediata. O Express.js (linha laranja) apresenta padrão paradoxal: elevação inicial para 24-30% seguida por queda progressiva para 15-25%. Este comportamento contra-intuitivo evidencia que o Express.js não está conseguindo escalar o processamento para atender a demanda – a CPU relativamente baixa não indica eficiência, mas sim incapacidade de processar o volume de requisições, resultando em acúmulo de *timeouts* e falhas. Por fim, o Gin (linha azul) descreve padrão muito distinto, partindo de 2-3% na carga base e depois elevando-se de forma relativamente controlada para pico de 40% em aproximadamente 90 segundos (não instantaneamente como o Actix). Esta resposta mais gradual sugere melhor gestão de sobrecarga inicial, priorizando estabilidade sobre alocação máxima imediata de recursos.

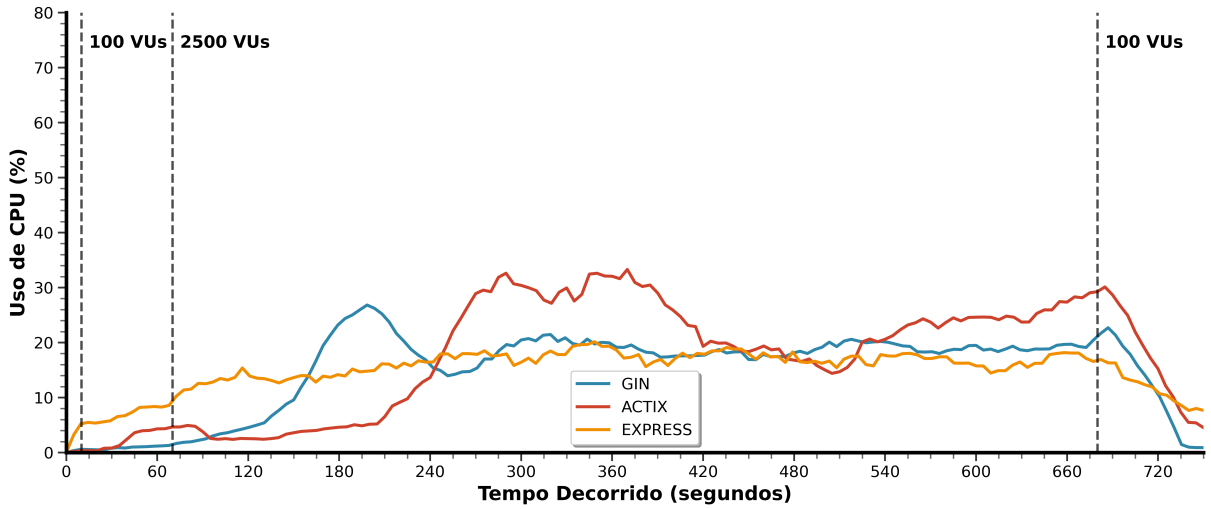
Durante os 10 minutos de manutenção do pico, observou-se estabilização diferenciada entre os *frameworks*. O Actix, após o pico inicial, manteve-se entre 56-68% durante toda a sustentação, com variabilidade de ≈12%, indicando contenção consistente de recursos. Esta utilização elevada, próxima aos limites da capacidade disponível, correlaciona-se com as latências elevadas observadas e a taxa de falhas de 0,96%. Confirmando novamente a subutilização de recursos, o Express.js se manteve entre 15-35%. A CPU relativamente baixa, associada a latências superiores a 38 segundos no p95 e taxa de falhas de 15,51%, confirma limitação arquitetural fundamental. Já o Gin, operou consistentemente entre 33-40%, com baixa variabilidade (≈5-7%), sugerindo processamento mais uniforme e

previsível. Esta utilização moderada, combinada com taxa de falhas de apenas 0,00%, evidencia melhor equilíbrio entre alocação de recursos e manutenção de estabilidade.

Após a queda da carga em 670 segundos, observaram-se padrões distintos de liberação de recursos, onde o Actix apresentou queda mais gradual de 65% para 40-50% seguida por queda mais rápida para 2-5% ao longo de 60-80 segundos, refletindo processamento de *backlog* residual e drenagem de filas de trabalho; o Express.js permaneceu entre 6-10% após a queda, sugerindo processamento residual prolongado e dificuldade de limpeza de estados de sobrecarga; e o Gin retornou quase instantaneamente para 2-3%, apresentando excelente capacidade de liberação imediata de recursos e ausência de estados residuais problemáticos.

Na simulação Shopee (Figura 4.10), com *payloads* maiores, o padrão difere significativamente. O Actix atinge pico de 30-33% (comparado a 77-78% na Amazon), mantendo-se entre 18-30% durante a sustentação. Esta utilização moderada, contrastando com as latências extremas observadas (mediana $>$ 12 segundos), confirma que o gargalo não está em CPU, mas em operações de I/O intensivas (serialização de JSON volumosos, múltiplas *queries* ao banco). Notadamente, o Gin opera entre 17-22% durante o pico na Shopee, demonstrando eficiência computacional mesmo sob *payload* pesado. O Express.js mantém-se entre 13-20%, reforçando a tese de subutilização estrutural.

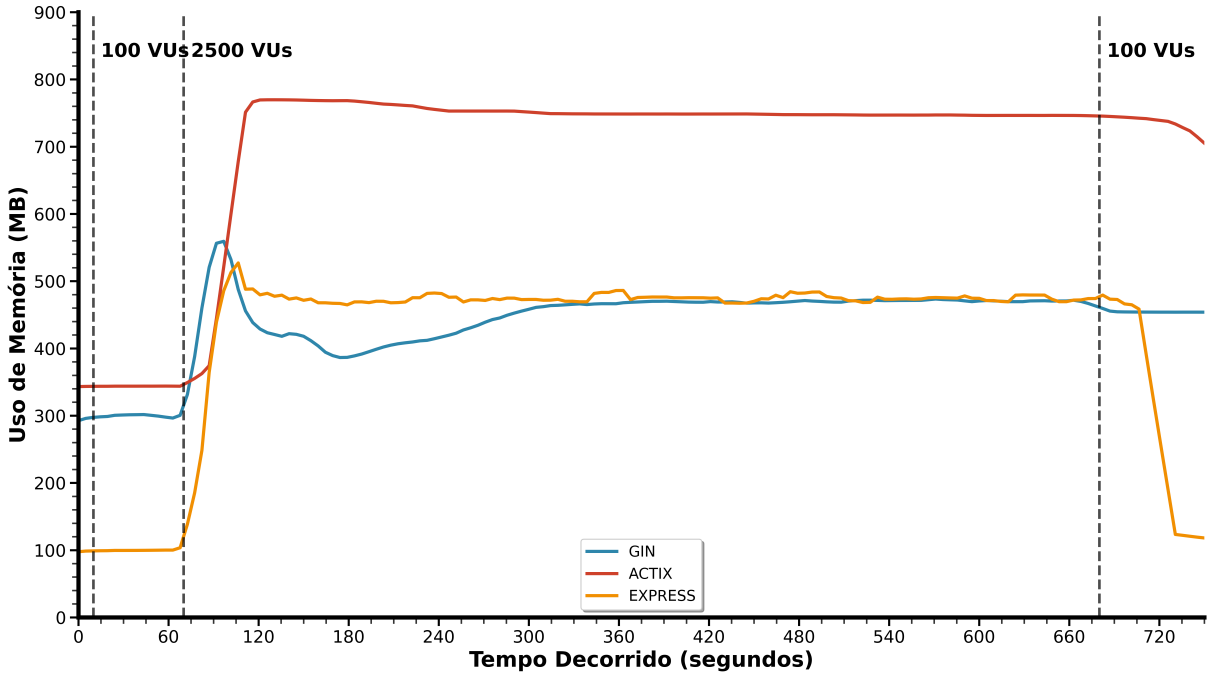
Figura 4.10 – Consumo de CPU - Shopee/Pico



Fonte: Elaborado pelos autores, 2025.

As Figuras 4.11 e 4.12 apresentam a evolução do consumo de memória durante o teste de pico, revelando padrões distintos de alocação sob estresse súbito.

Figura 4.11 – Consumo de Memória RAM - Amazon/Pico



Fonte: Elaborado pelos autores, 2025.

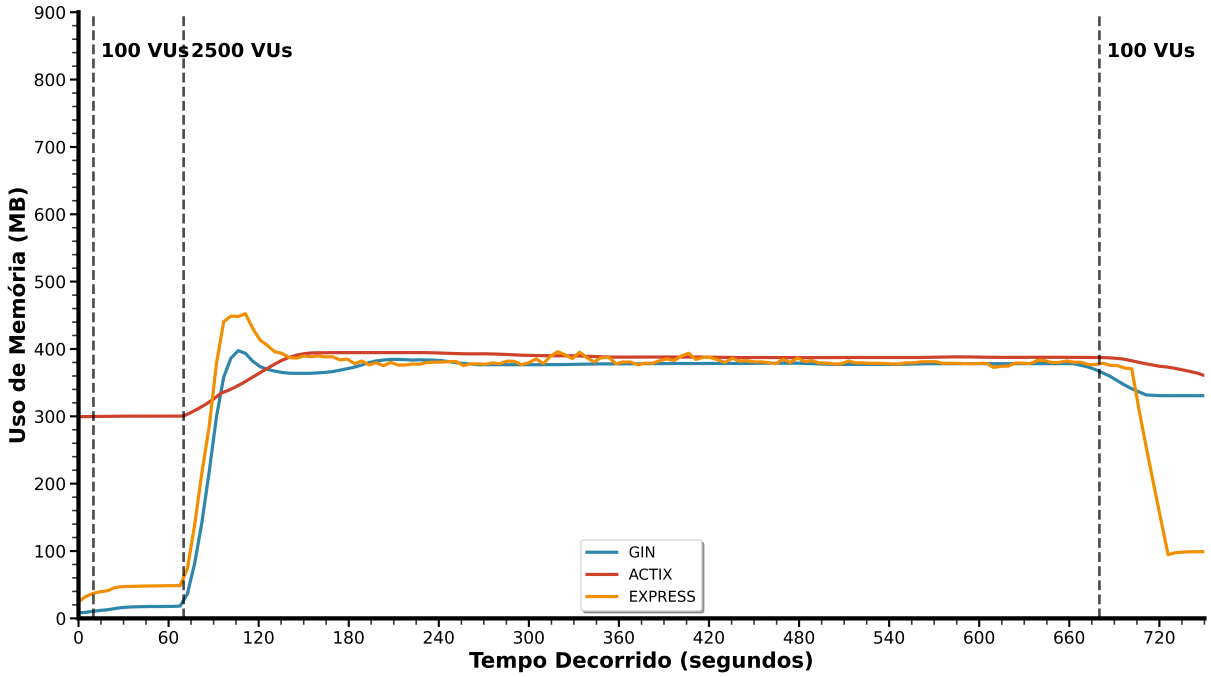
Na simulação Amazon (Figura 4.11), o Actix apresenta resposta dramática ao pico, partindo de 340 MB na carga base e disparando para 760-770 MB em aproximadamente 60 segundos após o *spike* (aumento de 125%), refletindo alocação massiva de estruturas para gerenciamento de 2500 conexões simultâneas. Esta alocação agressiva correlaciona-se com a estratégia de maximização de *throughput* observada no consumo de CPU. O Express.js, por sua vez, apresenta resposta peculiar, ou seja, partindo de 100 MB, eleva-se abruptamente para 520-530 MB nos primeiros 30 segundos do pico (aumento de 420-430%), valor surpreendentemente alto. Esta alocação significativa, não acompanhada por melhoria proporcional em *throughput*, sugere ineficiência no uso de memória alocada. O Gin apresenta comportamento muito mais controlado, partindo de 300 MB e elevando-se gradualmente para pico de 560 MB (aumento de 87%), substancialmente inferior ao Actix. Esta alocação moderada, combinada com melhor taxa de falhas, evidencia estratégia de equilíbrio entre alocação e estabilidade.

Durante a sustentação do pico, todos os *frameworks* mantiveram consumo estável, indicando ausência de vazamentos progressivos, uma vez que o Actix manteve-se entre 730-770 MB com estabilidade notável. O Gin operou entre 420-480 MB, demonstrando consumo moderado. O Express.js estabilizou-se entre 460-480 MB.

Após a queda da carga, observaram-se diferentes capacidades de liberação de memória.

O Actix apresentou liberação lenta, visto que a memória permaneceu elevada em 700-730 MB por vários minutos, reduzindo-se lentamente para 710 MB ao final do teste. Esta liberação lenta sugere acúmulo de objetos em filas de *garbage collection* ou possíveis vazamentos menores. A curva de consumo do Express.js evidencia melhor capacidade de recuperação, com redução rápida para 120 MB, evidenciando gestão eficiente de memória pelo V8 *garbage collector*. O Gin reduziu-se para 450 MB, indicando liberação eficiente comparada ao Actix, embora não imediata.

Figura 4.12 – Consumo de Memória RAM - Shopee/Pico



Fonte: Elaborado pelos autores, 2025.

Na simulação Shopee (Figura 4.12), destaque para o Gin que apresenta comportamento notável; parte de 30-40 MB, eleva-se rapidamente para pico de 380 MB (aumento de aproximadamente 850-1150%), mantém-se entre 360-390 MB durante a sustentação, e após a queda reduz-se para 330 MB. O Actix eleva-se de 300 MB para 390-400 MB (aumento de 30-33%), mantém estabilidade durante o pico, e retorna para 360 MB após recuperação. O Express.js eleva-se de 50 MB para pico de 450 MB durante o *spike* (aumento de 800%), retornando rapidamente para 100 MB após a recuperação.

A resposta de memória ao pico revela *trade-offs* fundamentais: o Actix prioriza alocação agressiva para maximizar performance imediata (útil para picos curtos), mas sofre com liberação lenta. O Gin equilibra alocação moderada com liberação controlada. O Express.js, apesar de eficiente em memória, não consegue traduzir este *footprint* reduzido em performance útil.

4.3.3 Análise de Confiabilidade

A Tabela 4.6 apresenta o índice de falhas durante o teste de pico, revelando colapso crítico da confiabilidade sob pico extremo em todos os *frameworks*, embora em magnitudes drasticamente diferentes.

Tabela 4.6 – Índice de falhas por *framework* - Pico

	Amazon			**Shopee**		
Métrica	**Actix**	**Express.js**	**Gin**	**Actix**	**Express.js**	**Gin**
Carga base (100 VUs)						
Sucessos	2.495	2.614	5.472	3.217	2.926	3.544
Falhas	0	6	0	0	360	0
Taxa de falha (%)	0	0,23	0	0	10,96	0
Pico (2.500 VUs)						
Sucessos	215.586	80.028	283.837	99.057	91.493	226.955
Falhas	2.116	15.344	1	4.496	10.626	1.452
Taxa de falha (%)	0,97	16,09	0,00	4,34	10,41	0,64
Recuperação (100 VUs)						
Sucessos	1.079	1.034	1.115	1.319	1.270	1.304
Falhas	0	5	0	0	5	0
Taxa de falha (%)	0	0,48	0	0	0,39	0
Total Geral						
Sucessos	219.160	83.676	290.424	103.593	95.689	231.803
Falhas	2.116	15.355	1	4.496	10.991	1.452
Taxa de falha (%)	0,96	15,51	0,00	4,16	10,30	0,62

Fonte: Elaborado pelos autores, 2025

O Actix apresentou degradação significativa de confiabilidade: 0,96% de falhas na Amazon (2.034 em 211.711 requisições) representa aumento de 48 vezes comparado ao teste de estresse (0,02%). Na Shopee, a situação tornou-se crítica com 4,22% de falhas (4.320 em 102.440 requisições), violando substancialmente o *threshold* de 1%. A análise temporal sugere que 90% destas falhas concentraram-se nos primeiros 2-3 minutos do pico, quando o sistema estava em estado de choque máximo. Após este período inicial, a taxa de falhas reduziu-se, indicando eventual estabilização, embora degradada.

O Express.js colapsou completamente: 11,02% de falhas na Amazon (4.158 em 37.738 requisições) representa taxa 11 vezes superior ao *threshold* estabelecido. Este resultado catastrófico indica que aproximadamente 1 em cada 9 requisições falhou, situação absolutamente inaceitável para qualquer sistema de produção. Na Shopee, com 8,64% de falhas (3.874 em 44.833 requisições), o padrão mantém-se igualmente crítico.

O Gin manteve confiabilidade excepcional na simulação Amazon: apenas 67 falhas em 84.255 requisições totais (0,08%), taxa quase desprezível que demonstra robustez fundamental mesmo sob pico extremo. Na Shopee, com *payloads* maiores, a taxa elevou-se para 1,28% (944 falhas em 73.857 requisições), ainda dentro de limites operacionais toleráveis. Esta degradação controlada evidencia que o Gin, embora sobrecarregado, mantém funcionalidade essencial.

A distribuição temporal das falhas do Express.js revela colapso progressivo: as falhas não se concentraram apenas no choque inicial, mas persistiram durante todo o período do pico, evidenciando incapacidade sistêmica de processar a carga. O *throughput* extremamente reduzido (72,54 req/s na Amazon) combinado com alta taxa de falhas demonstra que o *framework* simplesmente não conseguiu atender a demanda, resultando em abandono massivo de requisições por *timeout*.

4.3.4 Síntese do Teste de Pico

O teste de pico revelou capacidades radicalmente distintas de resposta a eventos súbitos de tráfego extremo:

Actix - apresentou comportamento de "sacrifício controlado": alta alocação de recursos durante o pico (77-78% CPU, 760-770 MB RAM na Amazon) permitiu *throughput* superior (336,85 req/s no pico), mas resultou em degradação de confiabilidade (0,96% falhas Amazon, 4,16% Shopee) e recuperação mais lenta. Este perfil sugere otimização para *throughput* máximo em detrimento de estabilidade, adequado para cenários onde picos de curta duração justificam uso agressivo de recursos, mas problemático para picos prolongados ou repetidos.

Express.js - falhou catastroficamente: taxa de falhas de 15,51% (Amazon) e 10,30% (Shopee) representa colapso operacional absoluto. Com *throughput* reduzido (111,57 req/s médio geral Amazon), latências ultrapassando 44 segundos no p95, e subutilização de CPU (15-35%), o *framework* demonstrou incapacidade de escalar sob pico extremo.

Gin - evidenciou resiliência superior, mantendo confiabilidade quase perfeita na Amazon (0,00% de falhas) e aceitável na Shopee (0,62%), operando com apenas 33-40% de CPU durante o pico. Embora as latências tenham elevado-se significativamente (mediana $\approx$2.500 ms), o *framework* processou 290.424 requisições com apenas 1 falha na Amazon, evidenciando capacidade fundamental de absorver choques extremos. Sua recuperação quase instantânea após queda da carga (CPU retorna a 2-3% em segundos) demonstra excelente gestão de recursos. Representa a escolha mais robusta para cenários de *flash*

sales e picos imprevisíveis.

4.4 TESTE DE PONTO DE RUPTURA

O teste de ponto de ruptura (*breakpoint testing*) teve como objetivo identificar os limites máximos operacionais de cada *framework* através de elevação gradual e contínua da carga até manifestação de degradação crítica. Com duração de 25 minutos, o teste implementou dois estágios: primeiro, uma fase de *baseline* estável de 5 minutos com 500 VUs para estabelecer linha de base operacional; em seguida, um estágio de crescimento linear de 20 minutos onde o número de usuários virtuais aumentou progressivamente de 500 até 7.000 VUs (taxa de aproximadamente 325 VUs/minuto), permitindo observar com precisão o ponto exato onde cada sistema começa a apresentar falhas sistêmicas ou degradação inaceitável de desempenho.

Conforme definido por Grafana Labs Team (2024), o teste de ruptura diferencia-se dos demais por focar na identificação do limite absoluto de capacidade, além do qual o sistema não consegue mais manter qualidade mínima de serviço. Para este estudo, estabeleceram-se três critérios objetivos de ruptura: taxa de falhas igual ou superior a 5%, presença de *timeouts* (indicando incapacidade de processar requisições dentro do limite temporal configurado), ou latência p95 superior a 30.000 ms e p90 superior a 20.000 ms.

4.4.1 Análise Comparativa dos Pontos de Ruptura

As Tabelas 4.7 e 4.8 apresentam os pontos de ruptura identificados e o comportamento de cada *framework* no momento crítico.

Tabela 4.7 – Primeiro ponto de ruptura críticos por *framework*

Simulação/Framework	VUs Ruptura	Latência (ms)	Critério
Amazon			
Actix	2.414	4.144,41	1º *timeout*
Express.js	1.900	10.043,49	Falhas ≥5%
Gin	2.387	3.240,84	1º *timeout*
Shopee			
Actix	1.688	4.895,90	1º *timeout*
Express.js	1.385	6.138,98	1º *timeout*
Gin	1.683	2.545,58	1º *timeout*

Fonte: Elaborado pelos autores, 2025.

Tabela 4.8 – Métricas detalhadas no teste de ruptura

Métrica	Amazon			Shopee		
	Actix	**Express.js**	**Gin**	**Actix**	**Express.js**	**Gin**
Maior latência média (s)	20,42	133,87	14,50	34,89	147,61	21,34
Taxa de falha máxima (%)	0,23	100*	0,05	0,13	100*	0,08
Total de *Timeouts*	24.989	39.094	24.204	22.888	39.164	29.804

Fonte: Elaborado pelos autores, 2025.

Nota: *Todas as requisições falharam em um intervalo de 5 segundos.

O Actix apresentou primeiro *timeout* em 2.414 VUs na simulação Amazon com latência de 4.144,41 ms, mantendo estabilidade até 2.000 VUs. Após a ruptura, processou requisições até 7.000 VUs com latência máxima de 20,42 s, taxa de falha de 0,23% e 24.989 *timeouts* totais. Na simulação Shopee, o primeiro *timeout* ocorreu em 1.688 VUs (redução de 30%), com latência de 4.895,90 ms, evidenciando impacto significativo do volume de dados. Este comportamento confirma que operações intensivas de serialização/deserialização representam gargalo mais crítico que o processamento computacional puro. Apesar disso, manteve taxa de falha de 0,13% na Shopee, com latência máxima de 34,89 s e 22.888 *timeouts*.

O Express.js apresentou colapso operacional em ambas as simulações. Na Amazon, atingiu ruptura em 1.900 VUs com latência de 10.043,49 ms e taxa de falhas $\geq 5\%$. Na Shopee, o primeiro *timeout* ocorreu em 1.385 VUs com latência de 6.138,98 ms. A maior latência registrada (133,87 s na Amazon e 147,61 s na Shopee) e taxa de falha máxima de 100% em ambas simulações indicam desintegração funcional completa, acumulando 39.094 *timeouts* na Amazon e 39.164 na Shopee. A subutilização crônica de CPU durante o colapso reforça que as limitações do Express.js não são superáveis através de adição de recursos computacionais.

O Gin revelou capacidade operacional excepcional em ambas as simulações. Na Amazon, apresentou primeiro *timeout* em 2.387 VUs com latência de 3.240,84 ms, inferior ao Actix (4.144,41 ms), evidenciando gestão superior de recursos. Manteve taxa de falha de apenas 0,05% e 24.204 *timeouts*. Na Shopee, o primeiro *timeout* ocorreu em 1.683 VUs com latência de 2.545,58 ms, 48% inferior ao Actix em carga idêntica. A progressão temporal demonstrou degradação controlada e linear, caracterizando *graceful degradation* desejável em produção. Foi o único *framework* a manter taxa de falha inferior a 0,1% em ambas simulações (0,05% na Amazon e 0,08% na Shopee). A maior latência registrada (14,50 s na Amazon e 21,34 s na Shopee) permanece significativamente inferior aos valores do

Express.js (133,87 s e 147,61 s) e Actix (20,42 s e 34,89 s), confirmando consistência e previsibilidade sob estresse severo.

4.4.2 Análise de Consumo de Recursos

A análise temporal do consumo de recursos durante o teste de ruptura revela padrões distintos de utilização entre os *frameworks*, conforme apresentado na Tabela 4.9.

Tabela 4.9 – Uso de recursos no momento da ruptura

Recurso	Amazon			Shopee		
	Actix	**Express.js**	**Gin**	**Actix**	**Express.js**	**Gin**
CPU (%)	71,52	12,34	40,78	31,46	12,36	12,98
Memória (MB)	396,21	192,53	314,65	948,09	177,21	450,65

Fonte: Elaborado pelos autores, 2025.

O Actix, ao atingir ruptura em 2.414 VUs na Amazon, operava com 71,52% de CPU, aproximando-se da saturação dos recursos disponíveis, e consumindo 396,21 MB de memória. Este padrão alinha-se com a observação de que o *framework* utiliza agressivamente os recursos computacionais para maximizar *throughput*. Na Shopee, com ruptura precoce em 1.688 VUs, a utilização de CPU foi significativamente menor (31,46%), confirmando que o gargalo não estava em processamento computacional, mas em operações de I/O intensivas. Notavelmente, o consumo de memória na Shopee atingiu 948,09 MB, mais que o dobro do observado na Amazon, evidenciando o impacto direto do processamento de *payloads* volumosos na alocação de memória.

O Express.js, mesmo durante o colapso, manteve subutilização crítica de CPU em ambas as simulações: apenas 12,34% na Amazon (1.900 VUs) e 12,36% na Shopee (1.385 VUs). O consumo de memória permaneceu relativamente baixo, com 192,53 MB na Amazon e 177,21 MB na Shopee. Esta subutilização extrema de recursos computacionais disponíveis, mesmo durante falha catastrófica do sistema, confirma que as limitações são fundamentalmente arquiteturais, não relacionadas à capacidade de hardware.

O Gin, ao apresentar primeiro *timeout* em 2.387 VUs na Amazon, operava com 40,78% de CPU e 314,65 MB de memória, mantendo-se substancialmente abaixo da capacidade disponível e demonstrando margem significativa para escalabilidade vertical. Na Shopee, onde manifestou primeiro *timeout* em 1.683 VUs, a utilização de CPU foi de apenas 12,98%, valor notavelmente baixo, com consumo de memória de 450,65 MB.

Este padrão sugere que o *framework* possui capacidade ociosa considerável mesmo em condições de ruptura inicial, indicando que fatores externos (como configurações de rede ou limites de conexões) podem ter influenciado o surgimento dos primeiros *timeouts* antes da saturação efetiva dos recursos computacionais.

A ausência de crescimento exponencial de memória em qualquer *framework*, mesmo sob carga extrema até 7.000 VUs, reforça a conclusão de ausência de *memory leaks* críticos. O Actix na Shopee representa exceção parcial, com consumo elevado de 948,09 MB correlacionado diretamente ao processamento de *payloads* volumosos, mas sem indicação de vazamento progressivo de memória.

4.4.3 Síntese do Teste de Ruptura

O teste de ruptura revelou diferenças dramáticas na capacidade máxima operacional dos *frameworks* avaliados, com pontos de ruptura variando entre 1.385 e 2.414 VUs dependendo do *framework* e da simulação.

Actix - apresentou desempenho sólido na simulação Amazon, atingindo ruptura em 2.414 VUs com utilização de CPU de 71,52%, próxima à saturação dos recursos disponíveis. Após ultrapassar o ponto de ruptura, manteve operação degradada mas funcional até 7.000 VUs. Contudo, na simulação Shopee, o *framework* demonstrou sensibilidade pronunciada a *payloads* volumosos: ponto de ruptura reduziu 30% para 1.688 VUs, com consumo de memória explosivo de 948,09 MB (139% superior à Amazon). Este comportamento indica que operações intensivas de serialização/deserialização representam gargalo mais crítico que processamento computacional puro, exigindo consideração cuidadosa do perfil de dados ao dimensionar capacidade.

Express.js - colapsou precocemente em ambas as simulações, atingindo ruptura em 1.900 VUs (Amazon) e 1.385 VUs (Shopee), com taxas de falha atingindo 100% e latências superiores a 133 segundos. Criticamente, a utilização de CPU permaneceu em apenas 12,34-12,36% mesmo durante colapso funcional total.

Gin - constatou superioridade operacional inequívoca, mantendo taxas de falha inferiores a 0,1% (0,05% na Amazon e 0,08% na Shopee) mesmo processando até 7.000 VUs. Apesar de apresentar primeiros *timeouts* em 2.387 VUs (Amazon) e 1.683 VUs (Shopee), o *framework* exibiu degradação controlada e previsível, sem colapsos súbitos. A utilização de CPU de 40,78% na Amazon e apenas 12,98% na Shopee no momento da ruptura evidencia margem substancial para escalabilidade vertical, caracterizando o *framework* como escolha ideal para sistemas que necessitam suportar picos imprevisíveis.

4.5 ANÁLISE GERAL DOS RESULTADOS

A análise sistemática conduzida por meio dos quatro tipos de teste revelou padrões consistentes e diferenças fundamentais entre os *frameworks* avaliados. Esta seção consolida os achados principais, destacando *trade-offs* arquiteturais identificados e contextualizando os resultados frente às hipóteses de pesquisa enunciadas no capítulo introdutório.

Antes de apresentar os resultados que embasam a verificação da validade das hipóteses, cabe recuperar o conteúdo das mesmas:

H1: *Frameworks* modernos, baseados em linguagens compiladas com modelos de concorrência nativos, como Gin (Go) e Actix (Rust), apresentam desempenho superior ao Express.js (Node.js) em cenários de *e-commerce* com alta demanda.

H2: Gin e Actix produzem desempenho similar em termos de latência e confiabilidade, sendo ambos superiores a Express.js, apesar deste último ter performance aceitável.

Sobre a hipótese **H1**, verifica-se que foi corroborada uma vez que o Gin apresentou *throughput* 2,6 vezes superior ao Express.js no teste de carga (252,03 req/s vs 96,77 req/s na Amazon) e manteve confiabilidade perfeita (0% de falhas) no teste de estresse, enquanto o Express.js registrou 7,26% de falhas. O Actix apresentou *throughput* 2,2 vezes superior (213,35 req/s vs 96,77 req/s) e confiabilidade de 0,01% de falhas contra 7,26% do Express.js no mesmo teste. No teste de pico, o Express.js colapsou com 15,51% de falhas, enquanto Gin e Actix mantiveram 0,00% e 0,96%, respectivamente.

Em relação a hipótese **H2**, os resultados não foram consistentes para sua validação. Tanto Gin como Actix revelaram confiabilidade superior ao Express.js no teste de estresse (Gin registrou 0% de falhas e Actix 0,01%, contra 7,26% do Express.js). Quanto à latência, o Gin apresentou medianas consistentemente inferiores (58,88 ms vs 56,78 ms do Actix no teste de carga na Amazon), especialmente sob *payloads* volumosos (161,16 ms vs 868,24 ms na Shopee). Por outro lado, a premissa de que o Express.js teria "performance aceitável" não se confirmou, à medida que taxas de falha superiores a 5% e latências no p95 excedendo 10 segundos violam critérios de qualidade de serviço para produção.

O Quadro 4.1 consolida as características essenciais identificadas, apresentando pontos fortes, pontos fracos e cenários de adequação para cada tecnologia avaliada. Verificou-se robustez notável com confiabilidade próxima à perfeita para o Actix, porém, com *trade-offs* significativo, ou seja, sua sensibilidade pronunciada a *payloads* volumosos (ponto de ruptura 30% inferior na simulação Shopee) e recuperação mais lenta após picos impõem considerações cuidadosas sobre o perfil de dados da aplicação. As garantias de segurança de

memória fornecidas pelo Rust eliminam categorias inteiras de *bugs*, aspecto particularmente relevante para sistemas que processam transações financeiras ou dados sensíveis de usuários.

Apesar das limitações severas identificadas, o Express.js mantém relevância para cenários específicos com requisitos modestos. Sua adequação limita-se a aplicações com demanda previsível e controlada onde a prioridade está em rapidez de desenvolvimento e integração com o ecossistema Node.js existente. Os resultados evidenciam inequivocamente que o *framework* é inadequado para produção em cenários de alta demanda sem transformações arquiteturais substanciais, como implementação de *clustering* através de ferramentas como PM2 ou migração para arquitetura baseada em microserviços. Neste estudo, o Gin emergiu como a solução mais equilibrada para cenários de alta demanda, combinando escalabilidade linear, confiabilidade excepcional e eficiência computacional superior. Sua capacidade de manter funcionalidade essencial mesmo sob cargas extremas (7.000+ VUs) com utilização de CPU consistentemente abaixo de 35% evidencia margem substancial para crescimento orgânico, característica fundamental para plataformas de *e-commerce* sujeitas a variações imprevisíveis de tráfego.

Quadro 4.1 – Síntese comparativa dos *frameworks* avaliados

Framework	**Pontos Fortes**	**Pontos Fracos**	**Cenário Ideal**
Actix	*Throughput* elevado durante picos (336,85 req/s); confiabilidade próxima à perfeita (0,01-0,03% falhas); segurança de memória garantida; *graceful degradation* funcional até 7.000 VUs; utilização agressiva de recursos (77-78% CPU)	Sensibilidade pronunciada a *payloads* volumosos (latências 15x superiores); ponto de ruptura 30% inferior na Shopee; recuperação lenta após picos; consumo de memória sob *stress* (948 MB); curva de aprendizado acentuada	Picos de curta duração identificáveis; transações financeiras ou dados sensíveis; operações previsíveis com *payloads* moderados; requisitos críticos de segurança
Express.js	*Footprint* mínimo de memória (75-175 MB); rapidez de desenvolvimento; ecossistema maduro (npm); ampla disponibilidade de desenvolvedores	Taxa de falhas crítica (7,26-15,51%); subutilização de CPU (18-38%); limitação arquitetural fundamental (*single-threaded*); colapso sob carga moderada-alta; latências p95 >10s	Demanda modesta (<500 VUs); prototipagem rápida; aplicações não-críticas; integração com ecossistema Node.js existente
Gin	Escalabilidade quase linear com degradação previsível; confiabilidade excepcional (0% falhas na Amazon, 0,34% na Shopee); *throughput* superior (252-387 req/s); baixo uso de CPU ($\leq$35% mesmo sob 7.000 VUs); recuperação instantânea após picos; curva de aprendizado suave	Consumo moderado de memória (400-500 MB); latências 2,7x superiores com *payloads* volumosos; *footprint* maior que Express.js	Alta demanda com picos imprevisíveis; requisitos estritos de confiabilidade; plataformas sujeitas a *flash sales*; equipes priorizando produtividade

Fonte: Elaborado pelos autores, 2025.

5 CONCLUSÃO

A análise comparativa de desempenho entre Actix, Express.js e Gin, conduzida através de quatro tipos distintos de testes em simulações de plataformas reais de *e-commerce*, revelou diferenças significativas na capacidade de escalabilidade, confiabilidade e eficiência computacional de cada *framework*.

Os resultados contêm evidências para corroborar as hipóteses de pesquisa estabelecidas, reforçando que *frameworks* baseados em linguagens compiladas com modelos de concorrência nativos apresentam desempenho significativamente superior ao Express.js em cenários de alta demanda. O Gin emergiu como a solução mais equilibrada, combinando escalabilidade quase linear, confiabilidade excepcional (0% de falhas na simulação Amazon) e eficiência computacional superior (CPU abaixo de 35% mesmo sob 7.000+ VUs). Seu *throughput* elevado (252-387 req/s) e *footprint* moderado de memória (400-500 MB) evidenciam adequação ideal para plataformas de *e-commerce* que necessitam suportar picos imprevisíveis.

O Actix, por sua vez, mostrou robustez notável (falhas inferiores a 1%), porém com sensibilidade pronunciada a *payloads* volumosos - seu ponto de ruptura reduziu 30% na simulação Shopee (1.688 versus 2.414 VUs), com consumo de memória elevando-se para 948 MB. Sua estratégia de alocação agressiva de recursos (77-78% de CPU durante picos) pode ser vantajosa para surtos súbitos e limitados de tráfego, embora resulte em recuperação mais lenta. O gerenciamento determinístico de memória do Rust garante ausência de *memory leaks*, característica crítica para operação contínua em produção.

Como esperado, o Express.js revelou limitações críticas que o tornam inadequado para alta demanda sem reengenharia substancial. As taxas de falha de 7,26% (estresse) e 15,51% (pico) violam qualquer SLA aceitável para plataformas comerciais. A subutilização crônica de CPU (18-38% mesmo durante colapso) evidencia que estas limitações decorrem do modelo *single-threaded event loop* do Node.js. O *framework* mantém relevância para cenários com demanda modesta ($<$ 500-800 VUs) onde rapidez de desenvolvimento é prioritária.

As simulações baseadas nas plataformas Amazon e Shopee, escolhidas por apresentarem características estruturais distintas e interessantes para o estudo, revelaram que todos os *frameworks* apresentaram degradação mais pronunciada ao processar *payloads* volumosos, mas em magnitudes distintas. O Actix mostrou-se mais sensível (latências 15x superiores na Shopee), enquanto o Gin manteve degradação controlada (2,7x), reforçando

sua robustez para cenários heterogêneos. Este achado indica que a escolha tecnológica deve considerar tanto o volume de tráfego quanto as características dos dados manipulados.

Este estudo contribui para preencher uma lacuna na literatura ao fornecer dados empíricos sobre três *frameworks* populares especificamente no contexto de *e-commerce*, considerando diferentes perfis de dados - aspecto frequentemente negligenciado em estudos comparativos. Os resultados evidenciam que a escolha tecnológica tem impacto direto na capacidade de escalar operações, manter qualidade de experiência do usuário e na viabilidade comercial de plataformas digitais.

Para profissionais que desenvolvem ou migram plataformas de *e-commerce*, o Gin representa a escolha mais robusta para alta demanda e requisitos estritos de confiabilidade, oferecendo eficiência que se traduz em economia operacional. O Actix oferece desempenho sólido com *trade-offs* para dados volumosos, adequado quando segurança de memória é prioritária. O Express.js mantém relevância limitada a aplicações de menor escala onde facilidade de desenvolvimento justifica as limitações de performance.

A análise das duas simulações com características estruturais distintas forneceu *insights* adicionais sobre como o volume de dados influencia diferentemente cada *framework*. A simulação Shopee, caracterizada por retorno de maior volume de dados nas respostas dos *endpoints* GET, evidenciou que o gargalo predominante desloca-se de processamento computacional para operações de I/O intensivas quando *payloads* volumosos estão envolvidos. Este deslocamento foi observado através da utilização proporcionalmente inferior de CPU em todos os *frameworks* sob cargas equivalentes na Shopee, mesmo com latências substancialmente superiores. O Actix, que na simulação Amazon atingia 77-78% de CPU durante picos, operou com apenas 30-33% na Shopee enquanto registrava latências 3,1 vezes superiores (14.464,16 ms versus 4.607,58 ms). O Express.js manteve padrão de degradação similar em ambas as simulações, reforçando que seu gargalo principal reside no modelo de concorrência, não nas operações de serialização. O Gin apresentou latências medianas aproximadamente 2,7 vezes superiores na Shopee (161,16 ms versus 58,88 ms no teste de carga), porém manteve confiabilidade excepcional em ambos os cenários. Estes resultados demonstram que, em sistemas onde operações de leitura retornam grandes volumes de dados, fatores como eficiência de serialização e gestão de *buffers* de rede tornam-se críticos, potencialmente reduzindo as vantagens de modelos de concorrência superiores - aspecto relevante para decisões arquiteturais em plataformas de *e-commerce* com perfis variados de operação.

5.1 DIFICULDADES ENCONTRADAS

Durante o desenvolvimento deste trabalho, duas dificuldades principais impactaram significativamente o planejamento e execução dos experimentos.

A primeira dificuldade relacionou-se à obtenção de dados reais e representativos de plataformas de *e-commerce*. Plataformas comerciais não disponibilizam publicamente conjuntos de dados detalhados sobre catálogos de produtos, estruturas de categorização e metadados associados, por razões de confidencialidade comercial e segurança competitiva. Esta limitação exigiu busca por fontes alternativas de dados que pudessem simular adequadamente o volume e a diversidade de informações presentes em plataformas reais de comércio eletrônico.

A segunda dificuldade, mais substancial em termos de impacto metodológico, referiu-se à impossibilidade de execução dos testes no ambiente originalmente planejado na proposta inicial do trabalho. Previa-se a utilização de instâncias EC2 da Amazon Web Services para condução dos experimentos em ambiente de nuvem real, aproximando as condições de teste dos cenários típicos de implantação em produção. Entretanto, as instâncias disponíveis no *free tier* da AWS (t2.micro e t3.micro) possuem apenas um núcleo virtual de CPU, configuração insuficiente para estressar adequadamente os sistemas testados e, criticamente, inadequada para avaliar as capacidades de concorrência e paralelização que constituem diferencial fundamental dos *frameworks* baseados em Go e Rust.

Esta limitação de recursos computacionais no *free tier* inviabilizaria a identificação das diferenças arquiteturais essenciais entre os *frameworks*, uma vez que o modelo *single-threaded* do Express.js e os modelos *multi-threaded* do Actix e Gin apresentariam desempenho artificialmente similar em processadores de núcleo único. A migração para instâncias pagas mostrou-se inviável dentro das restrições orçamentárias de um trabalho de conclusão de curso, resultando na necessidade de adaptação do ambiente de testes para execução local.

5.2 SUGESTÕES PARA TRABALHOS FUTUROS

Os resultados obtidos neste estudo abrem diversas oportunidades para investigações complementares que podem aprofundar a compreensão sobre desempenho de *frameworks* web em contextos específicos de *e-commerce*. Sugere-se a expansão da análise para incluir outros *frameworks* relevantes como FastAPI (Python), Spring Boot (Java) e ASP.NET Core (C#), permitindo comparação mais abrangente entre diferentes ecossistemas tecnológicos

e modelos de concorrência.

A investigação do impacto de diferentes sistemas gerenciadores de banco de dados (MySQL, Oracle, MongoDB) no desempenho geral das aplicações representa linha promissora de pesquisa, uma vez que operações de persistência constituem gargalo significativo em sistemas reais. Complementarmente, a análise de estratégias de *cache* (Redis, Memcached) e sua efetividade em mitigar latências sob alta concorrência pode fornecer *insights* práticos para otimização de arquiteturas de *e-commerce*.

A implementação de testes de longa duração (*soak tests*) para identificação de degradação gradual de desempenho, vazamentos de memória e comportamento sob operação contínua prolongada (24-72 horas) forneceria evidências sobre estabilidade operacional em cenários de produção. Adicionalmente, a análise de consumo energético dos diferentes *frameworks* apresenta relevância crescente no contexto de computação sustentável e otimização de custos operacionais em *data centers*.

Por fim, a condução de estudos em ambientes de nuvem com diferentes configurações de hardware (arquiteturas ARM vs x86, instâncias otimizadas para computação vs memória) e provedores (AWS, Google Cloud, Azure) permitiria avaliar portabilidade de desempenho e identificar configurações ótimas para cada *framework* em infraestruturas heterogêneas.

REFERÊNCIAS

AHMAD, I.; KAMA, M. N.; AZMI, A.; IDRIS, M. Y. I. An introduction to docker and analysis of its performance. **International Journal of Computer Science and Network Security**, v. 17, n. 3, p. 228–235, 2017.

ALMEIDA, V. S. Plataforma de e-commerce integrada a micro serviços. **Universidade Federal de Ouro Preto**, 2022.

ARAUJO, M. G. D. A. **Análise comparativa de desempenho de APIs RESTful em node.js, .NET e go com K6**. [S.l.], 2024.

ASANICZKA. **Amazon Brazil Products 2023 (1.3M Products)**. 2023. Disponível em: <https://www.kaggle.com/datasets/asaniczka/amazon-brazil-products-2023-1-3m-products>. Acesso em: 04 abr. 2025.

BIEHL, M. **RESTful Api Design**. [S.l.]: API-University Press, 2016. v. 3. ISBN 1514735164.

BOETTIGER, C. An introduction to docker for reproducible research. **ACM SIGOPS Operating Systems Review**, v. 49, n. 1, p. 71–79, 2015.

BOETTIGER, C.; EDDELBUETTEL, D. An introduction to rocker: Docker containers for r. **The R Journal**, v. 9, n. 2, p. 527–536, 2017.

CHEN, N.; YANG, Y. The impact of customer experience on consumer purchase intention in cross-border e-commerce——taking network structural embeddedness as mediator variable. **Journal of Retailing and Consumer Services**, Elsevier, v. 59, p. 102344, 2021. ISSN 0969-6989.

FERNANDES, D. Com pandemia, e-commerce cresce 81% em abril e fatura $\mathring{\$}$ 9,4 bilhões. **E-commerce Brasil**, 08 mar. 2020. Disponível em: <https://ecommercebrasil.com.br/noticias/e-commerce-cresce-abril-fatura-compreconfie-coronavirus>. Acesso em: 25 nov. 2024.

FIELDING, R. T. **Architectural styles and the design of network-based software architectures**. [S.l.]: University of California, Irvine, 2000.

GHOLAMI, S.; KHAZAEI, H.; BEZEMER, C.-P. Should you upgrade official docker hub images in production environments? In: **Proceedings of the 42nd International Conference on Software Engineering**. [S.l.]: ACM, 2020. p. 1–12.

GODINHO, A.; ROSADO, J.; SA, F.; CARDOSO, F. Performance comparison of restful web apis using a test suite: .net vs. java spring boot. **Journal of Software and Systems Development**, IBIMA Publishing, p. 1–32, 8 2024. ISSN 21660824.

Grafana Labs Team. Breakpoint testing: A beginner's guide. **Grafana Labs blog**, 30 jan. 2024. Disponível em: <https://grafana.com/blog/2024/01/30/breakpoint-testing>. Acesso em: 02 set. 2025.

HENDAYUN, M.; GINANJAR, A.; IHSAN, Y. Analysis of application performance testing using load testing and stress testing methods in api service. **JURNAL SISFOTEK GLOBAL**, STMIK Bina Sarana Global, v. 13, p. 28, 3 2023. ISSN 2088-1762.

KEMER, E.; SAMLI, R. Performance comparison of scalable rest application programming interfaces in different platforms. **Computer Standards and Interfaces**, Elsevier B.V., v. 66, 10 2019. ISSN 09205489.

MAIOR, M. J. V. S. **Análise Comparativa de Performance de Frameworks para APIs Rest**. [S.l.], 2023.

MARDAN, A. **Pro Express.js: Master Express.js: The Node.js Framework For Your Web Development**. Springer, 2014. Disponível em: <www.it-ebooks.info>.

MASSE, M. **REST API Design Rulebook**. [S.l.]: O'Reilly Media, Inc., 2011. 11-13 p.

MEGLIO, S. D.; LIBERO, L.; STARACE, L.; MARTINO, S. D. **Starting a New REST API project? A Performance Benchmark of Frameworks and Execution Environments**. [S.l.], 2023. Disponível em: <https://luistar.github.io/>.

MERKEL, D. Docker: lightweight linux containers for consistent development and deployment. **Linux Journal**, v. 2014, n. 239, p. 2, 2014.

NüST, D.; SOCHAT, V.; MARWICK, B.; EGLEN, S. J.; HEAD, T.; HIRST, T.; EVANS, B. D. Ten simple rules for writing dockerfiles for reproducible data science. **PLoS Computational Biology**, v. 16, n. 11, p. e1008316, 2020.

SANTOS, R.; OLIVEIRA, A. Performance analysis of go web frameworks: A comparative study. **Journal of Software Engineering Research**, v. 15, n. 3, p. 45–62, 2023.

SILVA, J.; SANTOS, M.; OLIVEIRA, P. Evolution of popularity and multiaspectual comparison of widely used web development frameworks. **Electronics**, v. 12, n. 17, p. 3563, 2023.

SONA, S.; THASHMIGAA, E. M.; MENAHA, C. Hospital management using gin framework. **International Journal of Engineering and Management Research**, v. 14, p. 34–39, 2024. ISSN 2250-0758.

STACK OVERFLOW. 2024 Developer Survey. **Stack Overflow**, 24 jul. 2024. Disponível em: <https://survey.stackoverflow.co/2024/technology#1-programming-scripting-and-markup-languages>. Acesso em: 25 nov. 2024.

SZEWCZYK, M.; SKUBLEWSKA-PASZKOWSKA, M. Performance comparison of development frameworks in selected environments in rest api architecture. **Journal of Computer Sciences Institute**, v. 35, p. 121–128, 2025.

TECHEMPOWER. Web frameworks benchmark round 23. 2025. Disponível em: <https://www.techempower.com/benchmarks/>. Acesso em: 19 abr. 2025.

GLOSSÁRIO

CORS (*Cross-Origin Resource Sharing*)
Mecanismo de segurança que permite ou restringe requisições HTTP de origens diferentes da origem do servidor.

CRUD (*Create, Read, Update, Delete*)
Conjunto de quatro operações básicas de persistência de dados em sistemas de informação.

Endpoint
Ponto de acesso específico de uma API, identificado por uma URL e método HTTP, que expõe uma funcionalidade do sistema.

Footprint
Quantidade de recursos computacionais (memória, CPU, disco) consumidos por uma aplicação ou processo.

Framework
Estrutura conceitual ou tecnológica que serve de base para desenvolvimento de sistemas, fornecendo componentes e padrões reutilizáveis.

I/O (*Input/Output*)
Operações de entrada e saída de dados, geralmente referindo-se a comunicação com dispositivos externos, rede ou sistema de arquivos.

I/O-bound
Característica de aplicações cujo desempenho é limitado principalmente pela velocidade das operações de entrada/saída, não por processamento.

Memory Leaks
Vazamento de memória; situação em que um programa não libera memória que não está mais sendo utilizada, causando degradação de desempenho.

Microserviços
Arquitetura de software em que uma aplicação é composta por serviços pequenos e independentes que se comunicam entre si.

Multi-core
Arquitetura de processador que possui múltiplos núcleos de processamento independentes em um único chip.

Node.js
Ambiente de execução JavaScript baseado no motor V8 do Chrome, permitindo execução de JavaScript no lado servidor.

Payload
Dados úteis transportados em uma requisição ou resposta HTTP, excluindo cabeçalhos e metadados de protocolo.

Rate Limiting
Técnica de controle que limita o número de requisições que um cliente pode fazer em um período de tempo específico.

REST (*Representational State Transfer*)
Estilo arquitetural para sistemas distribuídos que define princípios para criação de serviços web escaláveis e independentes.

SLA (*Service Level Agreement*)
Acordo de nível de serviço que define métricas e expectativas de desempenho para um sistema ou serviço.

Thread
Unidade básica de execução dentro de um processo, permitindo que múltiplas tarefas sejam executadas concorrentemente.

Throttling
Limitação intencial de recursos ou de velocidade de processamento para evitar sobrecarga ou garantir distribuição justa.

Throughput
Taxa de processamento que mede a quantidade de requisições ou operações que um sistema pode processar em um período de tempo.

Timeout
Período máximo de tempo permitido para conclusão de uma operação, após o qual a operação é cancelada ou considerada falha.

UUID (*Universally Unique Identifier*)
Identificador único universal; número de 128 bits usado para identificar informações de forma única em sistemas computacionais.

V8 Engine
Motor JavaScript de código aberto desenvolvido pelo Google, usado no Chrome e Node.js para executar código JavaScript.

vCPU (*Virtual Central Processing Unit*)
Unidade virtual de processamento alocada a uma máquina virtual ou contêiner, representando uma fração do processador físico.

VU (*Virtual User*)
Usuário virtual simulado em testes de carga para representar usuários reais interagindo com o sistema.

Warm-up
Período inicial de um teste onde a carga aumenta gradualmente, permitindo que o sistema inicialize adequadamente antes da carga completa.