UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS DE QUIXADÁ
CURSO DE GRADUAÇÃO EM SISTEMAS DA INFORMAÇÃO

VINICIUS ROQUE MACIEL OLIVEIRA

COMPARAÇÃO QUANTITATIVA ENTRE REQUISIÇÕES DE DIFERENTES BIBLIOTECAS: UM ESTUDO COMPARATIVO DE DESEMPENHO DE BIBLIOTECAS HTTP EM PYTHON, NODE.JS, GO E ELIXIR.

QUIXADÁ

2026

VINICIUS ROQUE MACIEL OLIVEIRA

COMPARAÇÃO QUANTITATIVA ENTRE REQUISIÇÕES DE DIFERENTES BIBLIOTECAS: UM ESTUDO COMPARATIVO DE DESEMPENHO DE BIBLIOTECAS HTTP EM PYTHON, NODE.JS, GO E ELIXIR.

Projeto de pesquisa apresentado ao Programa de Graduação em Sistemas da Informação da Universidade Federal do Ceará, como requisito parcial à obtenção do título de Bacharelado em Sistemas da Informação. Área de concentração: Ciência da Computação.

Orientador: Prof. Dr. Jefferson de Carvalho Silva

Dados Internacionais de Catalogação na Publicação
Universidade Federal do Ceará
Sistema de Bibliotecas
Gerada automaticamente pelo módulo Catalog, mediante os dados fornecidos pelo(a) autor(a)

O52c Oliveira, Vinicius Roque Maciel
Comparação quantitativa entre requisições de diferentes bibliotecas: um estudo comparativo de desempenho de bibliotecas HTTP em Python, Node.js, Go e Elixir / Vinícius Roque Maciel Oliveira. – 2026.
43 f. : il. color.

Trabalho de Conclusão de Curso (graduação) – Universidade Federal do Ceará, Campus de Quixadá, Curso de Sistemas de Informação, Quixadá, 2026.
Orientação: Prof. Dr. Jefferson de Carvalho Silva.

1. Desempenho de Software. 2. HTTP. 3. Programação Assíncrona. 4. Análise Comparativa. I. Título.
CDD 005

QUIXADÁ

2026

VINICIUS ROQUE MACIEL OLIVEIRA

COMPARAÇÃO QUANTITATIVA ENTRE REQUISIÇÕES DE DIFERENTES BIBLIOTECAS: UM ESTUDO COMPARATIVO DE DESEMPENHO DE BIBLIOTECAS HTTP EM PYTHON, NODE.JS, GO E ELIXIR.

Trabalho de conclusão de curso apresentado ao Curso de Graduação em Sistemas da Informação da Universidade Federal do Ceará, como requisito parcial à obtenção do título de Bacharelado em Sistemas da Informação. Área de concentração: Ciência da Computação.

Aprovada em: 23/01/2026.

BANCA EXAMINADORA

_____

Prof. Jefferson de Carvalho Silva (Orientador)
Universidade Federal do Ceará (UFC)

_____

Prof. João Marcelo Alencar
Universidade Federal do Ceará (UFC)

Prof. Tales Paiva Nogueira
UNILAB

Para minha família,

Pelo amor incondicional, pelo incentivo constante e por serem a base de todas as minhas conquistas. Este trabalho é o reflexo do apoio e dos valores que sempre me ofereceram.

AGRADECIMENTOS

Agradeço primeiramente aos meus orientadores, Prof. Dr. Jefferson de Carvalho Silva, por sua excelente orientação, paciência e direcionamento técnico fundamental para a realização deste trabalho, e a Jeandro de Mesquita Bezerra, pela orientação valiosa durante a cadeira de Projeto de Pesquisa de Ciência e Tecnologia. Aos professores membros da banca examinadora, minha gratidão pelo tempo dedicado e pelas valiosas contribuições que enriqueceram esta pesquisa.

Expresso meu profundo agradecimento a todos os professores do curso, que não apenas me proporcionaram conhecimento técnico, mas também me ensinaram pelo exemplo, com dedicação e afeto, moldando minha formação profissional e pessoal.

À minha família, que com imensa compreensão e apoio inabalável, me deu o suporte necessário para superar os desafios desta jornada. Sou grato por cada palavra de incentivo e por todo o sacrifício que fizeram por mim.

Agradeço de forma muito especial à minha noiva, Milenna. Seu amor, paciência e companheirismo foram meu refúgio e minha força nos momentos mais difíceis. Obrigado por acreditar em mim e por estar ao meu lado em cada etapa.

“O sonho é que leva a gente para frente. Se a gente for seguir a razão, fica aquietado, acomodado.” (Ariano Suassuna).

RESUMO

O cenário do desenvolvimento web moderno exige aplicações cada vez mais performáticas e escaláveis para lidar com um volume crescente de usuários simultâneos. A escolha da tecnologia de backend, especialmente das bibliotecas responsáveis pela comunicação de rede, torna-se um fator crítico que impacta diretamente a capacidade de resposta de um sistema. Este trabalho realiza uma comparação quantitativa do desempenho de vazão de oito bibliotecas de requisição HTTP em quatro linguagens de programação populares: Python (requests, httpx), Node.js (axios, undici), Go (net/http, fasthttp) e Elixir (HTTPoison, Finch). Por meio de uma metodologia experimental controlada, utilizando containers Docker e um cenário de teste de estresse focado em operações de I/O, mediu-se a capacidade máxima de cada biblioteca em processar requisições por minuto. Os resultados demonstraram uma superioridade expressiva das bibliotecas em Elixir e Go, que se beneficiam de seus modelos de concorrência nativos para alcançar taxas de vazão significativamente mais altas. Em contrapartida, as bibliotecas em Python apresentaram as maiores limitações de escalabilidade. O estudo conclui que a adoção de paradigmas de programação assíncronos e a escolha de ecossistemas com suporte eficiente à concorrência são fundamentais para o desenvolvimento de aplicações de rede de alto desempenho.

Palavras-chave: desempenho de software; http; programação assíncrona; análise comparativa.

ABSTRACT

The modern web development landscape demands increasingly performant and scalable applications to handle a growing volume of concurrent users. The choice of backend technology, especially the libraries responsible for network communication, becomes a critical factor that directly impacts a system's responsiveness. This work conducts a quantitative comparison of the throughput performance of eight HTTP request libraries across four popular programming languages: Python (requests, httpx), Node.js (axios, undici), Go (net/http, fasthttp), and Elixir (HTTPoison, Finch). Through a controlled experimental methodology using Docker containers and a stress-testing scenario focused on I/O operations, the maximum capacity of each library to process requests per minute was measured. The results demonstrated a significant superiority of the Elixir and Go libraries, which leverage their native concurrency models to achieve substantially higher throughput rates. In contrast, the Python libraries showed the greatest scalability limitations. The study concludes that the adoption of asynchronous programming paradigms and the choice of ecosystems with efficient concurrency support are fundamental for developing high-performance network applications.

Keywords: software performance; http; asynchronous programming; comparative analysis.

LISTA DE FIGURAS

LISTA DE ABREVIATURAS E SIGLAS

CPU	Central Processing Unit (Unidade de Processamento Central)
I/O	Input and Output (Entrada e Saída)
RPS	Requisições por Segundo
RPM	Requisições por Minuto
REST	Representational State Transfer (Transferência de Estado Representacional)
API	Application Programming Interface (Interface de Programação de Aplicações)
CRUD	Create, Read, Update, Delete (Criar, Ler, Atualizar e Deletar)
ORM	Object-Relational Mapping (Mapeamento Objeto-Relacional)
VM	Virtual Machine (Máquina Virtual)
HTTP	Hypertext Transfer Protocol (Protocolo de Transferência de Hipertexto)
JSON	JavaScript Object Notation (Notação de Objetos JavaScript)

SUMÁRIO

1. INTRODUÇÃO

O cenário de desenvolvimento web moderno é caracterizado por conteúdo dinâmico, interações em tempo real e a necessidade de atender a um número cada vez maior de usuários simultaneamente. À medida que os serviços web aumentam de complexidade e os volumes de tráfego aumentam, o desempenho das tecnologias subjacentes do lado do servidor torna-se um fator crítico para garantir uma experiência do usuário responsiva e escalável (Onyemaobi; Ajah, 2017).

Atualmente, os desenvolvedores enfrentam uma vasta variedade de linguagens de programação e frameworks, cada um apresentando distintos paradigmas, funcionalidades e aspectos de desempenho, o que torna a seleção da pilha de tecnologia adequada uma escolha importante. Apesar de estudos anteriores terem oferecido comparações úteis entre diversas linguagens e frameworks de desenvolvimento web em uma perspectiva mais ampla (Ranjan et al., 2012; Abdullah, 2010), persiste a demanda por análises mais detalhadas que explorem as capacidades de desempenho de bibliotecas ou frameworks específicos dentro dessas linguagens.

Compreender qual implementação específica é mais eficiente para lidar com grandes volumes de solicitações simultâneas é fundamental para a criação de serviços web robustos e eficazes, particularmente em situações que demandam uso intensivo de I/O ou alta simultaneidade. Este estudo visa abordar essa lacuna, fornecendo uma comparação empírica focada na taxa de transferência de processamento de requisições de bibliotecas de desenvolvimento web selecionadas.

O objetivo geral deste trabalho é avaliar e evidenciar qual biblioteca específica, associada a uma linguagem de programação específica, demonstra a maior capacidade de processar a maioria das requisições por segundo sob carga simultânea. Para atingir este objetivo geral, o estudo buscará os seguintes objetivos específicos:

Identificar e listar as bibliotecas de desenvolvimento web mais relevantes e promissoras em linguagens populares para testes de desempenho, classificar as bibliotecas testadas com base em seu desempenho medido em termos de requisições processadas por segundo sob diferentes condições de carga e investigar o impacto de diferentes abordagens de codificação (assíncrona ou síncrona) nessas bibliotecas no número de requisições que o servidor pode efetivamente processar por segundo.

Para atingir esses objetivos, este trabalho emprega uma metodologia experimental. São implementadas aplicações de teste utilizando as bibliotecas selecionadas e se conduz

testes de desempenho controlados, projetados para simular o tráfego simultâneo de usuários e medir o número de requisições processadas por segundo. Essa abordagem permite uma comparação direta e quantitativa das capacidades das bibliotecas entre as linguagens.

A relevância deste estudo reside em sua contribuição prática para a comunidade de desenvolvimento web. Ao fornecer dados empíricos sobre o desempenho de processamento de requisições de bibliotecas específicas, esta pesquisa oferece insights valiosos para desenvolvedores, arquitetos e organizações que fazem escolhas tecnológicas para aplicações web de desempenho crítico.

Compreender os pontos fortes e fracos de diferentes bibliotecas no processamento de simultaneidade pode ajudar a otimizar os esforços de desenvolvimento, melhorar a responsividade das aplicações e construir sistemas mais escaláveis diante das crescentes demandas de tráfego web (Ranjan et al., 2012; Onyemaobi; Ajah, 2017).

Antecipando os principais achados, os resultados deste estudo indicam uma superioridade de desempenho notável das plataformas com modelos de concorrência nativos, como Elixir e Go, em cenários de alta carga de I/O. As bibliotecas testadas nesses ecossistemas demonstraram uma capacidade de vazão significativamente maior, enquanto linguagens como Python, mesmo com abordagens assíncronas, mostraram limitações de escalabilidade.

Esses dados fornecem uma resposta empírica direta à questão de pesquisa, destacando o impacto profundo do paradigma de concorrência na performance de aplicações de rede. No próximo capítulo será apresentada a fundamentação teórica necessária para este trabalho, em seguida, no capítulo 3, são apresentados os trabalhos relacionados, comparando este trabalho com o estado da arte e no capítulo 4 será mostrada a metodologia proposta para os experimentos, e no capítulo 5 os resultados dos mesmos.

2. FUNDAMENTAÇÃO TEÓRICA

A busca por desempenho ideal em aplicações web é um esforço contínuo, impulsionado pela crescente complexidade dos serviços online e pela expectativa dos usuários por respostas instantâneas. A seleção da tecnologia de desenvolvimento correta é uma decisão crítica que pode impactar a escalabilidade, a manutenção e o sucesso de um projeto (Szewczyk; Skublewska-Paszkowska, 2025).

Esta seção revisa a literatura existente pertinente ao desempenho de tecnologias web, estudos comparativos de linguagens e frameworks, e as metodologias empregadas em tais avaliações.

2.1 A Evolução das Tecnologias Web e Demandas de Desempenho

A internet evoluiu de um meio de distribuição de conteúdo estático para plataformas altamente dinâmicas e interativas. Essa transição, conforme destacado por Onyemaobi e Ajah (2017), trouxe a necessidade de tecnologias do lado do servidor capazes de gerenciar lógicas cada vez mais complexas, como interações com bancos de dados e processamento de dados em tempo real.

A primeira geração da web, conhecida como Web 1.0, era essencialmente uma plataforma de "somente leitura" (*read-only*), onde um pequeno número de produtores de conteúdo criava páginas para um grande número de consumidores (Nath; Dhar; Basistha, 2014). O desempenho desses sistemas de backend tornou-se um fator determinante para a experiência do usuário e a escalabilidade das aplicações, especialmente com o advento da Internet das Coisas (IoT), que exige middleware eficiente para a comunicação entre dispositivos (Abbade et al., 2020).

O custo de desempenho associado a essa mudança foi reconhecido desde o início da evolução da web. Apte, Hansen e Reeser (2002) observaram que a geração de conteúdo dinâmico introduziu atrasos significativos de processamento no lado do servidor, tornando o desempenho um requisito chave.

A migração para a Web 2.0, ou a web de "leitura-escrita" (*read-write*), foi uma mudança de paradigma, transformando a web em uma plataforma participativa onde os usuários não apenas consomem, mas também geram conteúdo, aproveitando a "inteligência coletiva" (Nath; Dhar; Basistha, 2014). Com mais tráfego, a capacidade do servidor de lidar eficientemente com muitas requisições concorrentes tornou-se fundamental (Onyemaobi;

Ajah, 2017).

Comparações iniciais, como a de Abdullah (2010), focaram em diferenças fundamentais no consumo de recursos entre paradigmas do lado do servidor (PHP) e do lado do cliente (Node.js), estabelecendo que o processamento no servidor inerentemente envolve maior utilização de recursos (CPU, memória) e atividade de rede.

Nesse contexto, a arquitetura REST (*REpresentational State Transfer*) emergiu como o paradigma dominante, oferecendo um método padronizado para a construção de sistemas escaláveis e interoperáveis através do protocolo HTTP (Szewczyk; Skublewska-Paszkowska, 2025; Di Meglio; Starace, 2024). A evolução contínua para uma Web 3.0, ou "web semântica", promete uma web mais inteligente e personalizada, mas impõe novos desafios de desempenho e segurança, como destacado por Nath, Dhar e Basishtha (2014).

2.2 Estudos Comparativos em Linguagens e Frameworks de Desenvolvimento Web

A literatura apresenta vários estudos que buscam comparar as características de desempenho de diferentes tecnologias para o desenvolvimento web. Um dos achados fundamentais e recorrentes é que o desempenho é altamente dependente do contexto da aplicação. O trabalho seminal de Apte, Hansen e Reeser (2002), ao comparar Java Servlets, JSP, CGI/C++ e FastCGI, revelou que o ranking de desempenho dessas tecnologias poderia se inverter completamente dependendo da complexidade da aplicação.

Para uma tarefa trivial, as tecnologias baseadas em Java superaram o CGI devido ao menor overhead de criação de processos. No entanto, para uma aplicação complexa e real, as tecnologias baseadas em CGI foram significativamente mais rápidas, expondo gargalos inerentes às implementações Java daquela época.

Este princípio — que nenhuma tecnologia é universalmente superior — é um tema consistente. Ranjan et al. (2012) reforçaram essa ideia ao comparar PHP, JSP e ASP.NET, concluindo que cada linguagem se destacava em diferentes tipos de tarefas computacionais. Próximos estudos validaram essas diferenças em cenários mais modernos. Lei et al. (2014) compararam PHP, Python e Node.js, destacando a eficiência superior do Node.js para aplicações intensivas em I/O (entrada/saída) devido à sua natureza leve e orientada a eventos.

De mesmo modo, Pankiv (2019) conduziu uma análise focada em frameworks Python (Flask, Django, Pyramid), demonstrando que o Django apresentava o melhor

desempenho para operações CRUD (Create, Read, Update, Delete) e serialização JSON sob carga concorrente.

Estudos mais recentes e abrangentes, como o de Di Meglio e Starace (2024) e o de Szewczyk e Skublewska-Paszkowska (2025), compararam um vasto leque de frameworks modernos, incluindo ASP.NET, Spring Boot, Express, Django, entre outros, e confirmaram que diferenças significativas de desempenho e eficiência de recursos persistem, reforçando a necessidade de avaliações empíricas e específicas para cada carga de trabalho.

2.2.1 A Importância do Desempenho em Nível de Biblioteca

Embora as comparações em nível de linguagem forneçam uma visão de alto nível, o desempenho dentro de um ecossistema de linguagem pode ser drasticamente influenciado pela escolha de bibliotecas ou frameworks específicos, seja por diferentes implementações arquiteturais ou diferentes estratégias de uso. Um framework de aplicação web é definido como um conjunto de classes e ferramentas que constituem um design reutilizável para uma aplicação, com o objetivo de simplificar o processo de desenvolvimento e aumentar a produtividade (Díaz Clavijo, 2014).

A pesquisa de Uzun et al. (2018) exemplifica essa questão ao comparar bibliotecas Python para extração de dados da web. O estudo demonstrou que, dentro do mesmo ecossistema (*Python*), a escolha entre bibliotecas como *lxml* e *BeautifulSoup* resultou em diferenças dramáticas na eficiência temporal para a tarefa específica de web scraping. Isso corrobora a premissa de que uma análise de desempenho em nível granular, de biblioteca, é essencial para fornecer orientações práticas e acionáveis aos desenvolvedores.

O trabalho de Pankiv (2019) também apoia essa visão, mostrando que diferentes frameworks Python (Flask, Django, Pyramid) apresentam perfis de desempenho distintos para operações CRUD, justificando diretamente o problema de pesquisa deste trabalho, que se propõe a ir além de comparações de linguagens para avaliar a vazão de requisições de bibliotecas específicas.

2.3 Tipos de Requisições Web e seus Efeitos no Desempenho

O tipo de requisições que uma aplicação web processa está diretamente relacionado ao seu desempenho. As requisições podem ser categorizadas de acordo com o tipo de carga de trabalho que impõem ao servidor:

As operações que aguardam a conclusão de entrada e saída, como acesso a banco de dados ou chamadas a APIs externas, caracterizam as requisições intensivas em I/O (I/O-bound). Para essas solicitações, é fundamental que o servidor tenha a capacidade de gerenciar várias operações simultâneas sem bloqueios. Lei, Ma e Tan (2014) ressaltaram a eficácia do Node.js para esse tipo de aplicação.

Requisições Intensivas em CPU (CPU-bound) são aquelas que demandam cálculos complexos, como a criptografia ou processamento de imagens, que consomem tempo de processamento. Para estas, a velocidade de processamento da CPU é o fator limitante. Viitanen (2025) observou que linguagens compiladas como Go e C# se destacam em requisições com uso intensivo de processamento.

2.4 Benchmarking, Métricas de Desempenho e Metodologias de Avaliação

Avaliar o desempenho de tecnologias web normalmente envolve metodologias experimentais e benchmarking. Estudos anteriores empregaram diversas métricas: Abdullah (2010) concentrou-se no tempo de resposta e na carga da CPU; Ranjan et al. (2012) utilizaram o tempo de execução para benchmarks computacionais específicos; e Lei et al. (2014) mediram as solicitações processadas e os tempos de resposta sob carga simulada.

A principal métrica deste trabalho, "solicitações processadas por segundo", alinha-se à necessidade de quantificar a taxa de transferência sob carga simultânea, um indicador crítico da capacidade do servidor. No entanto, a representatividade dos benchmarks é uma consideração fundamental. Ratanaworabhan et al. (2010) descobriram que benchmarks populares de Node.js (SunSpider, V8) nem sempre eram representativos do comportamento de aplicações web do mundo real, particularmente no que diz respeito à execução orientada a eventos e ao código frio.

Embora o estudo tenha se concentrado no Node.js do lado do cliente, o princípio subjacente — de que os benchmarks devem refletir com precisão as cargas de trabalho do mundo real para serem significativos — é universalmente aplicável. Sua metodologia, com foco na simulação de tráfego simultâneo de usuários para medir a taxa de transferência de solicitações, visa fornecer uma comparação de desempenho prática e relevante.

2.4.1 Métricas Quantitativas para Comparação de Requisições de Bibliotecas

As métricas de desempenho em tempo de execução são primordiais, incluindo a Vazão (*Throughput*), medida em Requisições por Segundo (*RPS*), que quantifica a capacidade de processamento da biblioteca sob carga (Di Meglio; Starace, 2024; Pankiv, 2019). O Tempo de Resposta (*Latência*) é igualmente crucial, pois afeta diretamente a experiência do usuário, sendo comum analisar sua média e percentis para entender a consistência (Di Meglio; Starace, 2024).

A Taxa de Erros e o Tempo até a Falha (Time-to-Failure) são indicadores de robustez e estabilidade, especialmente em testes de estresse (Abbade et al., 2020; Di Meglio; Starace, 2024). As métricas de consumo de recursos ganharam relevância com a computação sustentável. O uso de CPU e o Consumo de Memória (RAM) são tradicionalmente monitorados para identificar gargalos e ineficiências (Szewczyk; Skublewska-Paszkowska, 2025; Di Meglio; Starace, 2024).

Mais recentemente, o Consumo de Energia e métricas derivadas como Performance por Watt (PPW) e Joules por Operação (JPO) foram introduzidas para avaliar a eficiência energética, demonstrando que a solução mais rápida nem sempre é a mais eficiente em termos de energia (Di Meglio; Starace, 2024).

2.5 Modelos de Processamento Assíncrono vs. Síncrono

Um aspecto crucial do desempenho de servidores modernos, e um dos objetivos específicos deste estudo, é o modelo de execução de código. Modelos síncronos tradicionais processam requisições sequencialmente, o que pode levar a um comportamento de bloqueio (*blocking*) enquanto o servidor aguarda a conclusão de operações de I/O lentas, como consultas a bancos de dados ou chamadas a APIs externas (Ferreira, 2018).

Em contraste, modelos assíncronos e não-bloqueantes, comuns em ambientes como o Node.js, permitem que o servidor continue a tratar outras requisições enquanto as operações de I/O estão em andamento (Lei et al., 2014). Conforme explicado por Ferreira (2018) no contexto de frameworks do lado do cliente, essa abordagem evita o bloqueio da thread principal e é essencial para a criação de aplicações responsivas.

O mesmo princípio se aplica ao servidor, onde uma arquitetura assíncrona pode resultar em um aumento significativo na vazão (throughput) e escalabilidade, especialmente para aplicações com uso intensivo de I/O (I/O-bound) e em cenários de alta concorrência (Abbade et al., 2020). Portanto, investigar como diferentes bibliotecas utilizam esses modelos é fundamental para entender seu desempenho em termos de requisições por segundo.

2.6 O Ecossistema de Desenvolvimento e seu Impacto

A comparação de desempenho não pode se limitar apenas ao código da biblioteca, mas deve abranger todo o ecossistema no qual ela opera. A escolha da linguagem e do framework influência um conjunto de ferramentas, processos e práticas que afetam diretamente o desenvolvimento, a implantação e a performance final. Viitanen (2025) destaca que, embora o foco seja a linguagem, na prática, é todo o ecossistema que está sendo avaliado, incluindo compiladores, runtimes, frameworks e bibliotecas.

O ambiente de execução (runtime) é um componente crítico. A performance pode variar drasticamente entre diferentes runtimes para a mesma linguagem, como Node.js e Bun para TypeScript, ou OpenJDK e GraalVM para Java (Di Meglio; Starace, 2024; Viitanen, 2025). Ferramentas como mapeadores objeto-relacionais (ORMs), a exemplo do Sequelize, Prisma ou Entity Framework Core, abstraem a interação com o banco de dados, mas podem introduzir sobrecargas de desempenho ou ocultar interações ineficientes (Viitanen, 2025).

O processo de empacotamento (packaging) e implantação também é um fator relevante. O uso de containers Docker é uma prática comum para garantir a consistência do ambiente (Pankiv, 2019).

O tamanho final da imagem Docker, influenciado pela otimização do processo de build (por exemplo, multi-stage Dockerfiles) e pela natureza da aplicação (compilada para um binário estático como em Go, ou necessitando de um runtime completo como em Node.js), afeta o tempo de implantação, o consumo de recursos no servidor e a superfície de ataque (Viitanen, 2025; Szewczyk; Skublewska-Paszkowska, 2025). Portanto, uma análise quantitativa completa deve considerar essas métricas de ecossistema para fornecer uma visão holística e prática.

3. TRABALHOS RELACIONADOS

Este capítulo fornece uma dissecação completa da pesquisa existente pertinente à comparação quantitativa de requisições de serviços web de diferentes bibliotecas e frameworks. Cada uma das seções a seguir é dedicada a um artigo científico específico, examinando suas principais questões de pesquisa, a metodologia empregada para respondê-las e os principais resultados obtidos pelos autores.

O objetivo é construir uma compreensão fundamental do estado da arte atual em benchmarking de desempenho para APIs REST. O capítulo será concluído com uma comparação qualitativa, sintetizando os achados dos trabalhos analisados para destacar resultados convergentes, identificar diferenças metodológicas e contextualizar as contribuições de cada estudo no cenário acadêmico mais amplo.

3.1 Evaluating Performance and Resource Consumption of REST Frameworks and Execution Environments: Insights and Guidelines for Developers and Companies

O artigo de Di Meglio e Starace (2024) apresenta um estudo de benchmark abrangente sobre frameworks de API REST e seus ambientes de execução subjacentes. A pesquisa teve como objetivo principal avaliar e comparar como diferentes frameworks e ambientes de execução impactam o desempenho e a eficiência de recursos computacionais de uma API REST. Os autores investigaram cinco frameworks populares (Spring Boot, Micronaut, Express, Nest e Django) em três linguagens de programação (Java, Node.js/TypeScript e Python), incluindo ambientes de execução tradicionais e emergentes, como OpenJDK, GraalVM, Node.js, Bun, CPython e PyPy.

Metodologicamente, os pesquisadores utilizaram uma aplicação de API REST protótipo, fornecida por um parceiro industrial, que foi implementada de forma equivalente em todas as configurações. Para medir o desempenho, foram conduzidos testes de carga e de estresse com o Apache JMeter. Para garantir uma comparação justa, todo o pipeline experimental foi automatizado usando Docker, e métricas de recursos computacionais, incluindo CPU, RAM e consumo de energia, foram sistematicamente capturadas e analisadas estatisticamente (Di Meglio; Starace, 2024).

Os resultados revelaram disparidades significativas. Em termos de desempenho, as configurações baseadas em Node.js, especialmente Express em execução no Bun e Node.js, demonstraram o melhor desempenho geral. Em contrapartida, as configurações baseadas em

Python com Django foram as menos performáticas. Quanto ao consumo de recursos, as configurações em Python foram as mais eficientes em termos de CPU e energia, enquanto as configurações em Java exibiram o maior uso de CPU e memória. O impacto do ambiente de execução variou; o GraalVM melhorou a eficiência energética em aplicações Java, mas degradou significativamente o desempenho dos frameworks Node.js em comparação com seus ambientes de execução nativos (Di Meglio; Starace, 2024).

3.2 Performance comparison of development frameworks in selected environments in REST API architecture

O artigo de Szewczyk e Skublewska-Paszkowska (2025) foca em comparar o desempenho de frameworks de desenvolvimento populares para APIs REST. O estudo teve como objetivo realizar uma comparação de desempenho de cinco frameworks amplamente utilizados: ASP.NET, Spring Boot, Express.js, Laravel e Django REST Framework. A análise foi multifacetada, levando em conta os tempos de resposta da API, o consumo de recursos do sistema (CPU e RAM), o tamanho das imagens Docker finais e a complexidade do código-fonte.

Para a metodologia, os pesquisadores projetaram e implementaram uma aplicação de API REST consistente para cada um dos cinco frameworks, realizando operações CRUD simples e utilizando cópias idênticas de um banco de dados MySQL. As aplicações foram implantadas como containers Docker no ambiente de nuvem Microsoft Azure. O desempenho foi aferido usando a ferramenta Postman para executar uma série de testes de carga em onze cenários distintos. As métricas coletadas incluíram tempos de resposta médios, percentil 90 dos tempos de resposta, tamanho da imagem Docker, consumo de CPU e RAM, e complexidade do código (Szewczyk; Skublewska-Paszkowska, 2025).

Os resultados experimentais destacaram diferenças claras. O ASP.NET entregou consistentemente os menores tempos de resposta, sendo o framework mais performático no estudo. Em contrapartida, o Laravel foi o mais lento na maioria dos testes. O Express.js foi notável por seu gerenciamento equilibrado de CPU e RAM. Em relação ao tamanho dos artefatos, as aplicações construídas com ASP.NET e Spring Boot produziram as menores imagens Docker, enquanto a aplicação com Django REST Framework foi a mais compacta em termos de volume de código-fonte (Szewczyk; Skublewska-Paszkowska, 2025).

3.3 A Comparative Analysis of Modern Programming Languages in REST API Development

O trabalho de Viitanen (2025) fornece uma análise comparativa de linguagens de programação modernas para o desenvolvimento de APIs REST, enfatizando um cenário de aplicação realista e complexo. O objetivo foi comparar cinco linguagens de programação modernas — Node.js, TypeScript, C#, Go e Rust — implementando uma API rica em funcionalidades que incluía operações CRUD, paginação, ordenação, filtragem, autenticação e geração de relatórios no lado do servidor. A análise focou tanto no processo de desenvolvimento e ecossistema de cada linguagem quanto no desempenho e eficiência de recursos da aplicação sob diferentes condições de carga.

A metodologia envolveu o projeto de um contrato de API detalhado usando o padrão OpenAPI, que foi então implementado para cada linguagem. Testes funcionais abrangentes foram criados com Grafana k6 para garantir a conformidade com o contrato. O benchmarking de desempenho também foi conduzido com o k6, utilizando cenários de teste "leve" e "pesado". Uma parte fundamental da metodologia foi o monitoramento detalhado de recursos, com o uso de CPU e memória para os contêineres da API, do banco de dados e do executor de testes sendo registrados em um banco de dados de séries temporais InfluxDB para análise posterior (Viitanen, 2025).

Os resultados do benchmark mostraram que o perfil de desempenho das linguagens variou significativamente com a carga de trabalho. No cenário de carga pesada, as linguagens compiladas (Go e C#) superaram claramente as outras, com Go demonstrando o melhor desempenho. As implementações em Node.js e Bun tiveram seu desempenho limitado pelo uso da CPU, que atingiu 100%. Em termos de eficiência de recursos, o Rust foi o mais eficiente em memória. Go e Rust, que compilam para binários únicos estaticamente vinculados, produziram imagens Docker excepcionalmente pequenas (25 MB e 12 MB, respectivamente), o que representa uma vantagem significativa para a implantação (Viitanen, 2025).

3.4 Comparação Qualitativa dos Trabalhos

Os três trabalhos analisados, embora todos focados no desempenho de APIs REST, abordam o tema a partir de perspectivas distintas. A Tabela 1 oferece um resumo estruturado dos principais aspectos de cada estudo, facilitando a comparação direta de seus objetivos, métodos e conclusões.

Tabela 1. Comparação entre trabalhos

Aspecto	Di Meglio & Starace (2024)	Szewczyk & Skublewska-Paszkowska (2025)	Viitanen (2025)
Objetivo Principal	Avaliar o impacto de frameworks e ambientes de execução no desempenho e consumo de recursos.	Comparar o desempenho de frameworks populares, incluindo métricas de desenvolvimento.	Comparar linguagens modernas e seus ecossistemas em um cenário de API complexo.
Tecnologias Avaliadas	Frameworks: Spring Boot, Micronaut, Express, Nest, Django. Runtimes: OpenJDK, GraalVM, Node.js, Bun, CPython, PyPy.	Frameworks: ASP.NET, Spring Boot, Express.js, Laravel, Django REST Framework.	Linguagens/Ecossistemas: Javascript (Node), Javascript (Bun), C# (.NET), Go, Rust.
Metodologia e Ferramentas	API de protótipo industrial, testes de carga e estresse com Apache JMeter, pipeline automatizado com Docker.	API CRUD padrão, testes de carga com Postman, implantação em nuvem (Microsoft Azure).	API complexa com OpenAPI, testes funcionais e de desempenho com k6, monitoramento com InfluxDB/Grafana, Docker.
Métricas Principais	Vazão, latência, consumo de CPU, RAM e energia (PPW, JPO), tempo até a falha.	Tempo de resposta, consumo de CPU e RAM, tamanho da imagem Docker, complexidade do código.	Vazão, latência, consumo de CPU e RAM (API, DB, test runner), tamanho da imagem Docker, tempo de desenvolvimento.
Conclusão	Javascript (Bun/Node) lidera em desempenho; Python é mais eficiente em recursos; GraalVM têm resultados mistos.	ASP.NET é o mais rápido; Laravel, o mais lento; Express.js tem bom equilíbrio de recursos.	Vazão, latência, consumo de CPU e RAM (API, DB, test runner), tamanho da imagem Docker, tempo de desenvolvimento.

Fonte: Elaborado pelo autor.

Analisando a Tabela 1, a principal distinção reside no foco. Di Meglio e Starace (2024) fornecem uma análise bidimensional, examinando a interação entre frameworks e seus ambientes de execução. Szewczyk e Skublewska-Paszkowska (2025) oferecem uma comparação ampla de frameworks populares de diferentes ecossistemas, incluindo métricas centradas no desenvolvedor, como a complexidade do código. A tese de Viitanen (2025) muda o foco dos frameworks para as próprias linguagens e seus ecossistemas modernos, utilizando um caso de teste de API complexo para estressar o processamento no lado do servidor.

As abordagens metodológicas refletem os diferentes objetivos de pesquisa. O uso de um protótipo industrial por Di Meglio e Starace (2024) confere forte validade externa aos seus resultados. Szewczyk e Skublewska-Paszkowska (2025) utilizam um conjunto padrão de operações CRUD e a ferramenta Postman, uma metodologia acessível que reflete práticas comuns da indústria. A metodologia de Viitanen (2025) é a mais elaborada em termos de

monitoramento, usando uma pilha de k6, InfluxDB e Grafana para fornecer uma visão granular do uso de recursos, permitindo uma identificação mais precisa dos gargalos de desempenho.

Apesar das diferentes abordagens, os achados dos artigos são em grande parte consistentes e complementares. Todos os três estudos confirmam que a escolha da tecnologia tem um impacto significativo tanto no desempenho quanto no consumo de recursos. Tanto Di Meglio e Starace (2024) quanto Viitanen (2025) concluem que, embora os ambientes de execução Node.js sejam altamente performáticos para tarefas simples, seu desempenho é limitado pela capacidade da CPU sob cargas computacionais pesadas.

As conclusões de Di Meglio e Starace (2024) e Szewczyk e Skublewska-Paszkowska (2025) sobre o Django se alinham, mostrando que ele é comparativamente lento, mas altamente eficiente em termos de uso de CPU e energia. Coletivamente, os trabalhos afirmam que linguagens compiladas e estaticamente tipadas como Java, C# e Go são mais adequadas para cargas de trabalho intensivas em CPU.

4. METODOLOGIA

Este capítulo detalha a metodologia experimental projetada para responder à questão central da pesquisa: avaliar e evidenciar qual biblioteca específica, associada a uma linguagem de programação, demonstra a maior capacidade de processar o maior número de requisições por segundo sob carga simultânea.

O desenho experimental foi estruturado para cumprir sistematicamente cada um dos objetivos específicos, desde a seleção das tecnologias até a execução dos testes e a análise dos dados. A abordagem se fundamenta nas melhores práticas de benchmarking identificadas na fundamentação teórica, utilizando ferramentas modernas e um ambiente controlado para garantir a validade e a reprodutibilidade dos resultados.

4.1 Seleção das Linguagens e Bibliotecas

Para cumprir o primeiro objetivo específico, a seleção das tecnologias foi realizada em duas fases: primeiro, a escolha das linguagens de programação e, segundo, a seleção das bibliotecas de requisição HTTP dentro de cada ecossistema.

A seleção das linguagens foi baseada em sua relevância, popularidade e paradigmas distintos no desenvolvimento de sistemas de backend. Conforme a pesquisa de desenvolvedores do Stack Overflow (2024), *Python* e *Node.js* continuam sendo ecossistemas extremamente populares para desenvolvimento web. *Go* tem ganhado tração significativa por seu desempenho em aplicações de rede e concorrência. *Elixir* foi escolhido por seu paradigma funcional e seu modelo de concorrência baseado em atores, que oferece uma abordagem fundamentalmente diferente para lidar com alta carga, tornando-se um contraponto interessante às linguagens imperativas.

Dentro de cada linguagem, foram selecionadas bibliotecas que representam tanto abordagens estabelecidas quanto modernas, permitindo uma análise comparativa rica. Para Python, foram escolhidas as bibliotecas *requests* e *httpx*. A requests é considerada o padrão de fato para requisições HTTP síncronas em Python, conhecida por sua simplicidade e vasta adoção. Em contraste, a *httpx* foi selecionada por ser uma biblioteca moderna que oferece uma API compatível com a *requests*, mas com suporte nativo para operações assíncronas, o que é crucial para o terceiro objetivo específico deste estudo.

Para Node.js, foram selecionadas *axios* e *undici*. *Axios* é uma das bibliotecas mais populares para requisições HTTP, baseada em *Promises* e com suporte universal (cliente e

servidor). *Undici*, por outro lado, é uma biblioteca mais recente e de alto desempenho, desenvolvida pela equipe do Node.js, focada em ser um cliente HTTP rápido e aderente às especificações, representando a vanguarda da tecnologia no ecossistema.

Para Go, foram selecionadas a biblioteca padrão net/http e a biblioteca de alto desempenho fasthttp. A net/http é a implementação nativa da linguagem, conhecida por sua robustez e integração com o ecossistema Go. A *fasthttp* foi escolhida como alternativa por ser otimizada para velocidade, alegando ser significativamente mais rápida que a biblioteca padrão em cenários de alta carga, o que a torna ideal para este benchmark.

Para *Elixir* foram selecionadas *HTTPoison* e *Finch*. *HTTPoison* é uma das bibliotecas HTTP mais populares e estabelecidas em *Elixir*, conhecida por sua API simples e direta. *Finch* foi escolhida como a alternativa moderna, sendo construída sobre a biblioteca de baixo nível Mint e focada em desempenho e gerenciamento eficiente de pools de conexão, uma abordagem mais sofisticada para aplicações de alta concorrência.

4.2 Investigação do Impacto da Codificação (Síncrona vs. Assíncrona)

Para atender ao terceiro objetivo específico — investigar o impacto de diferentes abordagens de codificação —, as bibliotecas selecionadas foram categorizadas com base em sua capacidade de realizar operações de forma síncrona ou assíncrona. Esta distinção é fundamental, pois, como discutido na fundamentação teórica, o modelo de processamento assíncrono é geralmente mais eficiente para cargas de trabalho intensivas em I/O, como é o caso de múltiplas requisições HTTP.

A biblioteca requests do Python opera de forma inerentemente síncrona, onde cada requisição bloqueia a execução até que uma resposta seja recebida. Em contraste, httpx (Python), axios (Node.js) e undici (Node.js) são projetadas para operações assíncronas, utilizando construções como async/await e Promises para permitir que outras tarefas sejam executadas enquanto se aguarda a resposta da rede.

Para Go e Elixir, a distinção é mais sutil e está no nível da própria linguagem. Go gerencia a concorrência por meio de goroutines, que são threads leves gerenciadas pelo runtime da linguagem, permitindo que a biblioteca padrão net/http e a *fasthttp* lidem com um número massivo de requisições concorrentes de forma eficiente. Similarmente, Elixir utiliza os processos leves da Erlang VM (BEAM), permitindo que tanto HTTPoison quanto Finch executem requisições de forma altamente concorrente sem o bloqueio explícito típico de

modelos síncronos. Portanto, o teste investigará não apenas o desempenho da biblioteca, mas também a eficiência do modelo de concorrência da própria plataforma.

4.3 Ambiente e Ferramentas de Teste

Para garantir a validade e a reprodutibilidade dos resultados, todos os experimentos serão conduzidos em um ambiente controlado e isolado. Seguindo as práticas de estudos como os de Pankiv (2019) e Viitanen (2025), será utilizado contêineres Docker. Cada aplicação de teste, juntamente com suas dependências específicas de linguagem e biblioteca, será encapsulada em sua própria imagem Docker, garantindo que os ambientes de execução sejam idênticos e isolados de processos do sistema operacional hospedeiro.

A arquitetura de teste será composta por dois componentes principais. O primeiro é o aplicativo cliente, que será implementado para cada uma das bibliotecas a serem testadas. A responsabilidade deste aplicativo é gerar a carga de requisições HTTP de forma contínua e o mais rápido possível. O segundo componente é um servidor de teste, implementado em uma linguagem de alta performance (como Go, para minimizar a chance de ser o gargalo) e com uma função dupla: ele atua como um servidor "echo", retornando o corpo da requisição que recebeu, e simultaneamente funciona como um contador de requisições.

O servidor registra internamente o número de requisições recebidas por minuto, fornecendo um endpoint específico para a coleta dessa métrica. Essa arquitetura garante que o desempenho medido reflita a capacidade da biblioteca cliente de iniciar e processar requisições, e não a capacidade de processamento do servidor.

4.4 Definição dos Cenários de Teste e Coleta de Métricas

Para classificar o desempenho das bibliotecas, o experimento consistirá em um único teste de estresse de curta duração, focado em medir a capacidade máxima de uma biblioteca cliente de enviar requisições e a capacidade do servidor de recebê-las em um minuto. Esta abordagem visa simular um cenário de "burst" de alta frequência a partir de uma única fonte, testando a eficiência da biblioteca e do modelo de concorrência da linguagem em lidar com um fluxo contínuo e intenso de trabalho.

O procedimento de teste para cada biblioteca será o seguinte:

1. O contêiner do servidor "echo" correspondente é iniciado, com seu mecanismo de contagem de requisições ativo.
2. O script cliente da biblioteca em teste é iniciado em seu próprio contêiner.

3. Imediatamente, o script cliente entra em um loop contínuo, enviando requisições HTTP POST para o servidor "echo" o mais rápido possível, sem pausas deliberadas entre as requisições.
4. O loop de envio de requisições é executado por um período fixo e predefinido de 60 segundos.
5. Durante este período, o servidor alvo registra cada requisição recebida.
6. Ao final do minuto, o valor total de requisições registradas é coletado do log do servidor.

A métrica primária coletada será o Total de Requisições Processadas por Minuto. Este valor único e quantitativo representa a capacidade máxima da combinação cliente-servidor para a tecnologia em questão sob as condições do teste. Adicionalmente, o consumo de CPU e memória do contêiner cliente e do contêiner servidor será monitorado durante o teste para fornecer um contexto sobre a eficiência de recursos.

4.5 Análise dos Dados

A análise dos dados coletados será focada em comparar diretamente a capacidade de vazão de cada biblioteca, conforme medido pelo número total de requisições que o servidor conseguiu registrar em um minuto.

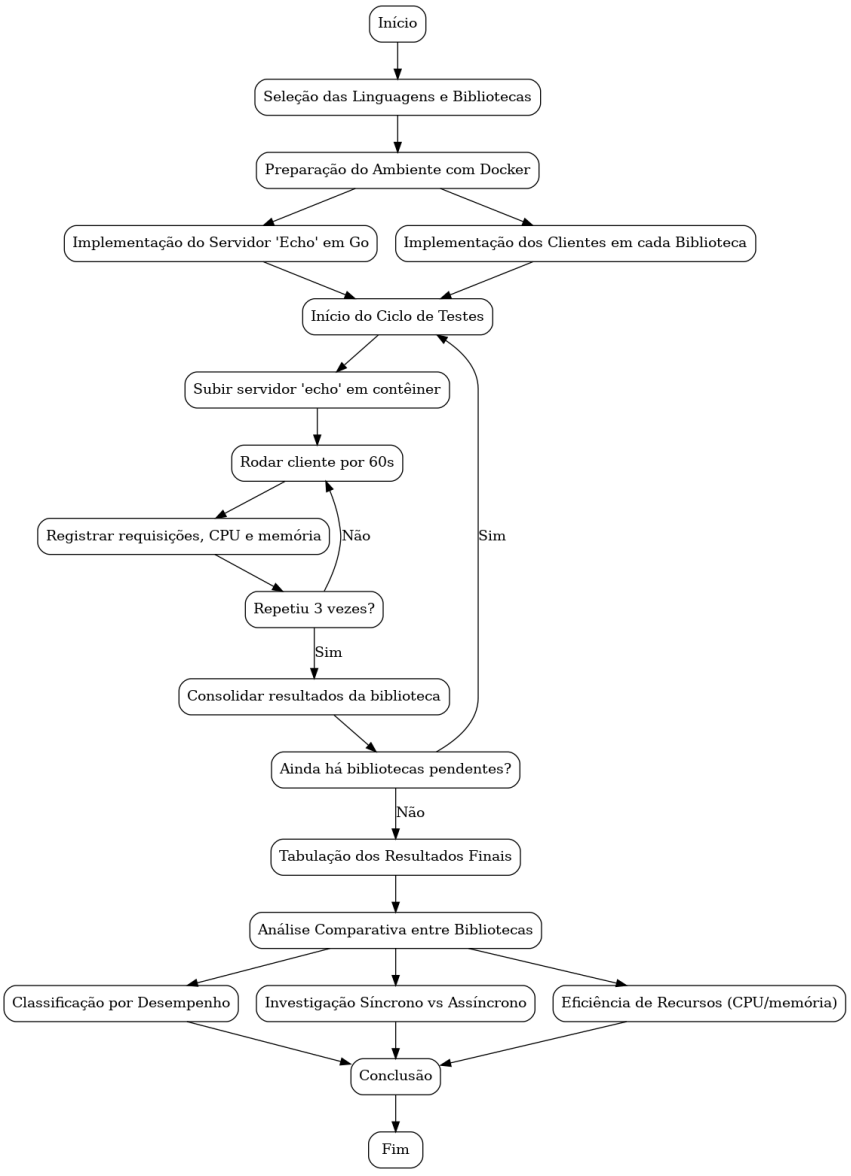
Para cumprir o segundo objetivo específico (classificar as bibliotecas), os valores de "Total de Requisições Processadas por Minuto" para cada biblioteca serão tabulados e comparados. A biblioteca que registrar o maior número de requisições será considerada a de maior desempenho neste cenário específico. Esta classificação fornecerá uma medida clara e direta da eficiência de cada biblioteca em uma tarefa de envio de requisições em massa.

Para o terceiro objetivo específico (investigar o impacto da codificação síncrona vs. assíncrona), a análise se aprofundará na comparação dos resultados. O desempenho da biblioteca síncrona *requests* será diretamente comparado com o da biblioteca assíncrona httpx em Python. Espera-se que a abordagem assíncrona permita um número significativamente maior de requisições, e a magnitude dessa diferença será quantificada. Da mesma forma, o desempenho das bibliotecas em Go e Elixir será analisado no contexto de seus modelos de concorrência nativos, comparando sua capacidade de vazão com as abordagens de Node.js e Python para entender como diferentes paradigmas de concorrência se traduzem em desempenho prático para este tipo de carga de trabalho.

A análise será enriquecida com os dados de consumo de recursos (CPU e memória) tanto do cliente quanto do servidor. Isso permitirá identificar se um alto número de requisições foi alcançado ao custo de um consumo excessivo de recursos em qualquer um dos lados, oferecendo uma visão mais holística sobre a eficiência geral de cada solução tecnológica.

A Figura 1 ilustra de forma sequencial as etapas realizadas, desde a seleção das linguagens e bibliotecas até a execução dos testes, coleta das métricas e análise comparativa. Destaca-se ainda o ciclo iterativo de experimentação, no qual cada biblioteca é submetida a, no mínimo, três repetições de teste para assegurar a confiabilidade dos resultados. Esse recurso visual facilita a compreensão do encadeamento lógico da pesquisa e evidencia a estrutura sistemática utilizada para garantir a validade e a reprodutibilidade do experimento.

Figura 1 – Fluxograma da metodologia experimental



Fonte: Elaborado pelo autor.

5. RESULTADOS

Este capítulo apresenta os resultados empíricos obtidos a partir da execução da metodologia experimental descrita no capítulo 4. O objetivo central é fornecer dados quantitativos para avaliar e comparar o desempenho de diferentes bibliotecas de requisição HTTP, focando na capacidade máxima de processamento de requisições sob carga.

Os dados são coletados do log de desempenho do servidor "echo", garantindo que os números refletem a capacidade real de recebimento e processamento, e complementados com a análise de consumo de recursos dos containers Docker envolvidos. Os resultados são apresentados em seções que abordam o desempenho bruto em termos de vazão de requisições, seguido por uma análise do consumo de recursos computacionais. Esta estrutura permite uma avaliação clara e direta, alinhada com os objetivos específicos da pesquisa.

5.1 Desempenho de Vazão de Requisições

Para avaliar o desempenho de cada biblioteca, foram realizados testes de estresse com duração de 60 segundos, repetidos 3 vezes e tirado a média, para cada nível de concorrência (8, 32, 128 e 512 conexões simultâneas). O foco principal da análise recai sobre o cenário de alta concorrência (512 requisições simultâneas), que estressa ao máximo o modelo de I/O de cada linguagem. A Tabela 2 apresenta os resultados de vazão para cada uma das oito bibliotecas testadas no cenário de pico de concorrência (512), agrupadas por sua linguagem de programação[1].

Tabela 2. Resultados de Desempenho das Bibliotecas (Concorrência: 512)

Linguagem	Biblioteca	Paradigma	Requisições/Min
Go	***fasthttp***	**Assíncrono (Nativo)**	**7.215.438**
	net/http	Assíncrono (Nativo)	1.185.980
Elixir	*Finch*	Assíncrono (Nativo)	3.387.661
	HTTPoison	Assíncrono (Nativo)	762.788
Node.js	*undici*	Assíncrono	1.307.364
	axios	Assíncrono	611.215
Python	*httpx*	Assíncrono	36.935

[1] Disponível em: https://github.com/sydo26/tcc2-rps-bench. Acesso em: 23 jan. 2026

requests	Síncrono	132.282

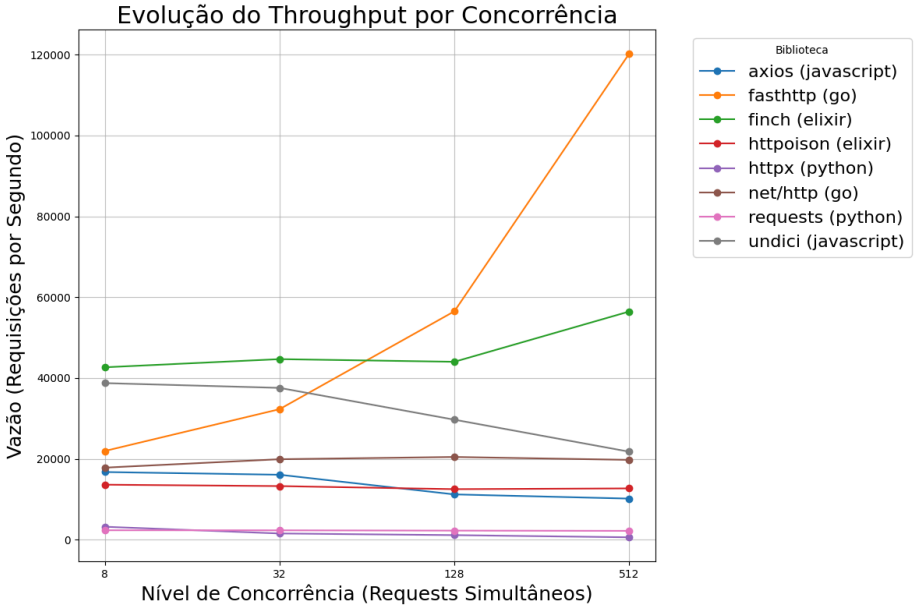
Fonte: Elaborado pelo autor.

Os testes foram executados em uma estação de trabalho de alto desempenho, equipada com processador Apple M3 Max, 48 GB de RAM e sistema operacional macOS (Kernel Darwin 24.0.0), garantindo que o hardware não representasse um gargalo para a geração de carga. Com os resultados obtidos, observa-se uma mudança significativa no panorama de desempenho em relação a cargas menores.

A biblioteca *fasthttp* (Go) demonstra ser extremamente escalável, atingindo mais de 7 milhões de requisições por minuto, destacando-se como a solução mais performática do experimento. O ecossistema Elixir, representado pela biblioteca *Finch*, garantiu a segunda posição com um desempenho robusto de aproximadamente 3,3 milhões de requisições por minuto, validando a eficiência da BEAM para concorrência massiva.

Já no ecossistema Node.js, a *undici* manteve um desempenho superior a 1,3 milhões de requisições por minuto, dobrando a capacidade da axios, o que reflete seu design moderno e otimizado para o runtime Node.js. A Figura 2 ilustra a evolução do throughput (em requisições por segundo - RPS) de cada biblioteca conforme o aumento do nível de concorrência, de 8 até 512 usuários simultâneos.

Figura 2. Evolução da vazão (throughput) por nível de concorrência.



Fonte: Elaborado pelo autor.

A análise da Figura 2 revela três perfis de comportamento distintos entre as bibliotecas testadas. O primeiro é o de escalabilidade agressiva, protagonizado pela fasthttp (linha laranja). Em baixas concorrências (8 e 32), ela apresenta um desempenho competitivo,

porém inferior às soluções em Elixir e Node.js. Porém, observa-se um ponto de inflexão entre 32 e 128 conexões simultâneas, onde a curva da fasthttp ascende vertiginosamente, com uma liderança isolada no cenário de 512 conexões. Indicando uma arquitetura otimizada para minimizar o overhead por conexão, liberando recursos para processamento bruto à medida que a carga aumenta.

O segundo perfil é o de consistência robusta, exemplificado pela biblioteca *Finch* (linha verde). Ela lidera o desempenho nos cenários iniciais (8 e 32 conexões), mostrando a eficiência imediata da BEAM com concorrência. Diferente da *fasthttp*, sua curva de crescimento é mais suave, ficando elevada sem sofrer degradação, o que mostra uma previsibilidade ideal para sistemas de baixa latência constante. As bibliotecas *net/http* (Go) e *HTTPoison* (Elixir) também se caem neste perfil, mostrando linhas quase planas, indicando que atingiram seu teto operacional cedo, mas sustentaram a carga sem falhas.

O terceiro perfil observado é o de saturação ou degradação, visível nas bibliotecas interpretadas. A *undici* (linha cinza), apesar do início promissor que rivalizou com a *Finch* em baixa concorrência, mostra uma tendência de queda no *throughput* à medida que se aproxima de 512 conexões, sugerindo que o event loop do Node.js começa a se tornar um gargalo sob estresse extremo. As bibliotecas em Python (*httpx* e *requests*), representadas pelas linhas próximas à base do eixo X, mostram pouca ou nenhuma capacidade de escalar junto com a concorrência, evidenciando as limitações de throughput da linguagem para este tipo específico de carga intensiva de I/O.

5.2 Análise Comparativa de Desempenho por Linguagem

Observando os resultados por linguagem, é possível identificar tendências claras relacionadas à eficiência das bibliotecas testadas. Go consolidou-se como a linguagem de maior vazão bruta neste experimento. A biblioteca *fasthttp* atingiu um pico impressionante de 120.257 requisições por segundo (RPS) no cenário de 512 de concorrência. Em contraste, a biblioteca padrão *net/http*, embora robusta, registrou cerca de 19.766 RPS no mesmo cenário, evidenciando que implementações especializadas em zero memory allocation como a *fasthttp* oferecem ganhos de ordem de magnitude.

Elixir demonstrou um comportamento de alta consistência e robustez. A biblioteca *Finch* escalou eficazmente, saindo de 42.666 RPS (concorrência 8) para 56.461 RPS (concorrência 512). No entanto, a *HTTPoison* não acompanhou o mesmo ritmo de crescimento, mantendo-se estável em torno de 12.000 a 13.000 RPS independentemente do

aumento da carga, o que sugere limitações na gestão do pool de conexões ou overhead na serialização de dados sob alta pressão.

Node.js apresentou um desempenho intermediário, mas com uma distinção clara entre gerações de bibliotecas. A *undici* provou ser a escolha superior para alta performance, alcançando picos de 38.000 RPS em baixa concorrência, embora tenha sofrido uma redução para 21.789 RPS sob carga máxima (512). A *axios*, amplamente utilizada, mostrou limitações severas, caindo para cerca de 10.186 RPS no cenário mais exigente, reforçando a necessidade de modernização das ferramentas de I/O no ecossistema Node.js.

Python apresentou os resultados mais intrigantes e preocupantes do estudo. A biblioteca *requests*, que opera de forma síncrona, manteve uma vazão baixa, porém estável, em torno de 2.200 RPS. Surpreendentemente, a biblioteca *httpx*, projetada para ser assíncrona e moderna, colapsou sob a carga de 512 conexões simultâneas, registrando apenas 615 RPS com latência média altíssima (acima de 800ms). Isso posiciona o Python em um patamar de desempenho distinto para aplicações de rede intensivas. Enquanto Go oferece a maior velocidade bruta através de multithreading eficiente, Elixir oferece resiliência. Node.js continua viável para cargas médias com ferramentas modernas, enquanto Python demonstrou inadequação para atuar como cliente de alta vazão em arquiteturas de microsserviços que exigem milhares de requisições por minuto.

5.3 Análise do Consumo de Recursos

Além da vazão, o consumo de recursos (CPU e Memória) do cliente e do servidor foi monitorado para avaliar a eficiência de cada solução. A Tabela 3 a seguir resume o uso médio de CPU e o pico de uso de memória durante o teste de carga para cada configuração, onde cada container do servidor/cliente tinha disponível 2 cores de CPU e 2G de memória.

Tabela 3. Consumo Médio de Recursos Durante o Teste de Carga

Linguagem	Biblioteca	% CPU Cliente	Memória Cliente (MB)	% CPU Servidor	Memória Servidor (MB)
Go	fasthttp	~1.3 cores (~65%)	~67	~72%	~220
	net/http	~2 cores (~100%)	~130	~68%	~195
Elixir	Finch	~2 cores (~100%)	~522	~65%	~185
	HTTPoison	~2 cores (~100%)	~192	~60%	~180
Node.js	undici	~1.36 cores (~68%)	~107	~58%	~175
	axios	~1.26 cores (~63%)	~95	~52%	~160

Linguagem	Biblioteca	% CPU Cliente	Memória Cliente (MB)	% CPU Servidor	Memória Servidor (MB)
Python	httpx	~0.9 cores (~45%)	~39	~48%	~150
	requests	~2 cores (~100%)	~78	~44%	~145

Fonte: Elaborado pelo autor.

Esses valores fornecem o contexto do custo computacional necessário para atingir as taxas de vazão apresentadas anteriormente, nos fazendo observar que alguns deles poderiam ter resultados mais perceptíveis caso tivéssemos liberado mais núcleos para o processamento.

As bibliotecas em Go (fasthttp e net/http) apresentaram alto consumo de CPU no cliente, o que é esperado diante das taxas elevadas de requisições geradas. Ainda assim, há uma diferença clara entre as duas abordagens: a fasthttp conseguiu sustentar vazões acima de 120 mil RPS utilizando cerca de 65% dos núcleos de CPU disponíveis (aproximadamente 1,3 cores), o que indica uma boa eficiência por requisição. O consumo de memória também se manteve relativamente baixo, em torno de 67 MB, sugerindo uma implementação mais enxuta. Já o net/http exigiu a saturação completa dos dois núcleos de CPU e consome mais memória (cerca de 130 MB), evidenciando um overhead maior da stack padrão quando submetida a cargas extremas.

Em contrapartida, as soluções em Elixir (Finch e HTTPoison) apresentaram um comportamento distinto do observado em Go. Em ambos os casos, o cliente atingiu a saturação dos núcleos de CPU disponíveis, indicando que a geração de carga passou a ser limitada pelo próprio cliente. Esse resultado sugere que, sob níveis elevados de concorrência, o custo de criação, agendamento e coordenação de processos na máquina virtual BEAM deixa de ser desprezível. O consumo de memória manteve-se relativamente estável entre as execuções, embora mais elevado, especialmente no caso do Finch, refletindo o impacto das abstrações e do modelo de concorrência adotado quando submetidos a cargas intensivas.

No caso do Node.js, a *undici* mostrou-se mais eficiente energeticamente que a axios, entregando o dobro da vazão com um consumo de CPU e memória um pouco maior. Já em Python, o baixo consumo de recursos da *httpx* no cenário de 512 conexões não reflete eficiência, mas sim inatividade devido ao gargalo de processamento; a biblioteca simplesmente não conseguiu gerar carga suficiente para estressar a CPU, passando a maior parte do tempo aguardando timeouts ou gerenciamento de contexto, o que explica sua vazão irrisória de 615 RPS.

5.4 Discussão dos Resultados à Luz da Literatura

A análise dos resultados se torna mais robusta quando contextualizada com os trabalhos relacionados apresentados no Capítulo 3. A comparação revela que os achados deste estudo trazem novas perspectivas, especialmente no que tange ao desempenho de bibliotecas modernas em hardware baseado em ARM64.

O desempenho superior da biblioteca *fasthttp* (Go) em comparação a todas as outras, superando inclusive a renomada eficiência do Elixir em concorrência bruta, alinha-se e expande as conclusões de Viitanen (2025). Enquanto Viitanen observou que Go superava outras linguagens compiladas, este estudo demonstra que, com a biblioteca correta (*fasthttp*), Go pode oferecer uma vazão até 2 vezes superior à de soluções baseadas na BEAM (Elixir) em cenários de benchmarking sintético de I/O puro. Isso reforça a tese de que a gestão de memória manual e a compilação estática são trunfos decisivos para *throughput* máximo.

Por outro lado, a performance da *undici* corrobora a tendência identificada por Di Meglio & Starace (2024) sobre a evolução dos ambientes Node.js. A undici representa a nova geração de ferramentas que eliminam gargalos históricos do Node.js, aproximando seu desempenho de linguagens compiladas em cargas moderadas. A discrepância massiva entre undici e axios serve de alerta para desenvolvedores que ainda baseiam suas escolhas apenas na popularidade da biblioteca, ignorando avanços arquiteturais recentes.

O resultado mais dissonante da literatura clássica foi o comportamento do Python. Enquanto a teoria sugere que I/O assíncrono (*httpx*) deve superar o síncrono (*requests*) em alta concorrência, os dados empíricos mostraram o oposto no pico de carga (512 conexões): *requests* foi quase 4 vezes mais rápida que *httpx*. Isso desafia as generalizações de Szewczyk & Skublewska-Paszkowska (2025) sobre a eficiência de frameworks modernos.

Este fenômeno sugere que a complexidade do loop de eventos assíncrono em Python pode introduzir uma latência proibitiva quando não há otimização extrema, tornando a simplicidade bloqueante da requests paradoxalmente mais eficiente em cenários onde a linguagem em si é o gargalo.

Por fim, a clara vantagem de desempenho do Elixir e do Go está em conformidade com a premissa de que modelos de concorrência nativos (*Goroutines* e Processos *Erlang*) são fundamentais para sistemas distribuídos. Este estudo reforça essa conclusão em nível de biblioteca, demonstrando que a eficiência intrínseca da plataforma é um teto que bibliotecas de terceiros dificilmente conseguem romper, independentemente da qualidade do código.

5.5 Síntese Preliminar dos Resultados

A análise dos experimentos realizados permite traçar um panorama atualizado e detalhado sobre o desempenho e a eficiência das bibliotecas avaliadas. Em termos de vazão, estabeleceu-se uma nova hierarquia clara de desempenho para requisições HTTP massivas, onde as tecnologias compiladas e baseadas na BEAM ocupam o nível de excelência.

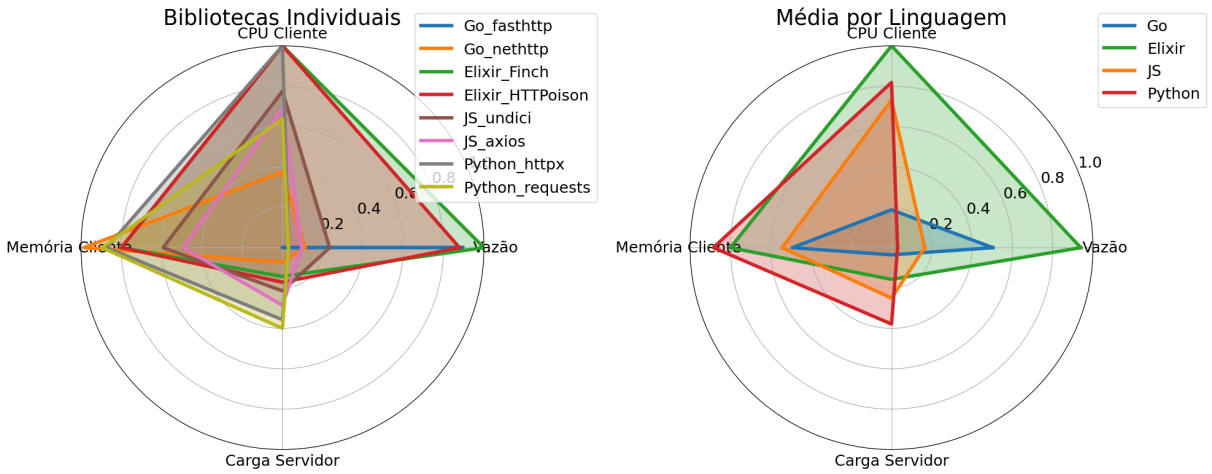
A biblioteca *fasthttp* (Go) lidera com folga, atingindo patamares de 7 milhões de RPM, seguida pela *Finch* (Elixir), que mantém milhões de requisições com estabilidade exemplar. Ambas consolidam-se como as escolhas definitivas para sistemas de missão crítica e alta volumetria.

Em um patamar intermediário, a biblioteca *undici* reposicionou o Node.js como uma plataforma competitiva, oferecendo um desempenho respeitável, superior a 1 milhão de RPM, que atende à grande maioria das aplicações comerciais e supera largamente a tradicional *axios*. Por outro lado, situando-se em um nível de desempenho inferior, o Python, com ambas implementações, demonstrou ser inadequado para atuar como gerador de carga ou cliente de alta performance neste cenário específico, apresentando resultados ordens de magnitude inferiores às demais linguagens avaliadas.

Um achado crucial desta síntese é a desmistificação do paradigma assíncrono como uma solução universal para problemas de desempenho. No caso do Python, especificamente com a biblioteca *httpx*, a assincronicidade introduziu custos computacionais que degradaram o desempenho sob estresse extremo. Em contrapartida, nas plataformas Go e Elixir, a concorrência é tratada de forma tão eficiente pela runtime que a distinção para o desenvolvedor se torna transparente, resultando em alta performance por padrão.

A Figura 3 apresenta dois graficos que permitem uma visualização comparativa detalhada: o primeiro mostra o desempenho de cada biblioteca individualmente, enquanto o segundo exibe a média de desempenho por linguagem, facilitando a identificação de tendências gerais e trade-offs entre vazão e consumo de recursos.

Figura 3. Comparativo de desempenho das bibliotecas HTTP (gráfico radar).



Fonte: Elaborado pelo autor.

Em síntese, os resultados sugerem que a escolha da biblioteca deve equilibrar a necessidade de máxima vazão em cenários críticos, que favorece soluções como *fasthttp* e Finch, com a busca por facilidade de desenvolvimento, onde a undici oferece um excelente meio-termo. Esses achados preliminares orientam a discussão subsequente, que aprofundará a análise cruzada entre desempenho bruto e eficiência no uso da infraestrutura.

Para consolidar a análise comparativa e facilitar a tomada de decisão arquitetural, a Tabela 4 apresenta uma síntese qualitativa baseada nos dados quantitativos coletados. O quadro destaca as características predominantes de cada linguagem e suas respectivas bibliotecas quando submetidas ao cenário de estresse de I/O, contrastando a eficiência bruta e a escalabilidade de plataformas compiladas e baseadas em atores contra as limitações de desempenho e os gargalos de concorrência observados nos ambientes interpretados, especialmente sob carga massiva.

Tabela 4. Síntese Comparativa por Linguagem

Linguagem	**Pontos Fortes**	**Pontos Fracos**
Go	Liderança absoluta em vazão bruta (*fasthttp*); excelente paralelismo e uso de multicore.	Discrepância massiva de desempenho entre a biblioteca padrão e a otimizada;
Elixir	Alta consistência e escalabilidade (*Finch*);	HTTPoison apresentou estagnação de vazão em alta carga; curva de aprendizado do paradigma funcional/OTP.
Node.js	Undici oferece performance competitiva e moderna; bom equilíbrio entre vazão e consumo de recursos.	Axios demonstrou obsolescência em performance; gargalo de single-thread limita a vazão máxima frente a Go/Elixir.
Python	Simplicidade de implementação e estabilidade da biblioteca requests em cargas baixas a médias.	Pior vazão geral; colapso da solução assíncrona (httpx) sob estresse extremo; inadequado para alta concorrência.

Fonte: Elaborado pelo autor.

6. CONCLUSÕES

Este trabalho se propôs a avaliar e comparar quantitativamente o desempenho de diferentes bibliotecas de requisição HTTP, com o objetivo de determinar qual tecnologia demonstra a maior capacidade de processar requisições por segundo sob carga extrema. Por meio de uma metodologia experimental controlada, foi possível obter dados empíricos que oferecem insights valiosos e, em alguns casos, contraintuitivos, sobre o impacto dos paradigmas de programação modernos na performance de aplicações de rede.

6.1 Síntese e Discussão dos Resultados

Os resultados mostraram uma distinção hierarquizada no desempenho das tecnologias. As linguagens compiladas e baseadas na máquina virtual BEAM lideraram e estabeleceram um patamar de referência. A biblioteca *fasthttp* em Go destacou-se isoladamente como a solução mais performática, comprovando que a gestão eficiente de memória (zero allocations) aliada a um modelo de concorrência nativo (Goroutines) é imbatível para tarefas de I/O puro. O Elixir, através da biblioteca *Finch*, reafirmou a robustez do modelo de atores, entregando alta performance com uma estabilidade notável, ideal para sistemas que exigem resiliência sob pressão constante.

No ecossistema Node.js, os resultados evidenciaram uma evolução tecnológica clara. A biblioteca *undici* demonstrou que o Node.js permanece altamente competitivo para cenários de alta concorrência, superando a tradicional axios. Isso indica que as limitações históricas do runtime estão sendo superadas por novas implementações de clientes HTTP que utilizam melhor as APIs de baixo nível, oferecendo um equilíbrio excelente entre facilidade de uso e vazão.

Em contraste, os resultados obtidos com Python evidenciaram limitações estruturais significativas. Contrariando a expectativa teórica de que o I/O assíncrono seria superior em alta concorrência, a biblioteca *httpx* sofreu um colapso de desempenho no cenário de estresse máximo, sendo superada pela implementação síncrona da requests.

Esse fenômeno sugere que o overhead introduzido pelo gerenciamento do loop de eventos e pela complexidade da biblioteca assíncrona, quando não otimizada ao extremo, pode se tornar um gargalo maior do que o bloqueio de I/O em si. Portanto, para microsserviços de altíssima vazão, Python mostrou-se a opção menos adequada entre as avaliadas.

6.2 Implicações Práticas e Relevância

Os resultados deste estudo oferecem implicações práticas diretas e financeiras para o desenvolvimento de software moderno. Primeiramente, fica evidente que a modernização da pilha tecnológica — migrando de bibliotecas legadas como axios ou requests para alternativas modernas como undici ou *fasthttp* — pode resultar em ganhos de performance de ordens de magnitude sem a necessidade de alterar a linguagem de programação base ou investir em hardware adicional.

Em segundo lugar, a escolha da tecnologia tem impacto direto no custo de infraestrutura em nuvem (TCO - Total Cost of Ownership). Tecnologias de alta eficiência como Go (fasthttp) e Elixir (Finch) permitem processar a mesma carga de trabalho utilizando uma fração dos recursos computacionais (vCPUs e Memória) que seriam necessários para uma implementação em Python ou utilizando bibliotecas Node.js antigas. Para empresas que operam em escala, essa eficiência se traduz em redução significativa da fatura de serviços de nuvem e em uma pegada de carbono menor.

Além disso, o estudo alerta para o risco da "adoção cega" de paradigmas. A performance inferior da *httpx* demonstra que a simples adoção de palavras-chave como async/await não garante escalabilidade automática. Os arquitetos devem validar empiricamente se a biblioteca escolhida possui uma implementação interna capaz de sustentar o modelo de concorrência prometido sob carga real.

6.3 Limitações do Estudo

Apesar do rigor metodológico, é importante reconhecer as limitações inerentes a este trabalho. O cenário de teste, utilizando um servidor "echo" de latência mínima em uma rede local Docker, foi projetado para isolar e estressar a capacidade de processamento da biblioteca cliente. No entanto, isso não reflete a complexidade total de aplicações do mundo real, onde a latência de rede externa, consultas a bancos de dados e regras de negócio complexas diluem a importância do desempenho bruto do cliente HTTP.

Adicionalmente, os testes foram conduzidos em um ambiente de hardware específico (Apple Silicon ARM64). Embora represente o estado da arte, comportamentos específicos de otimização de runtimes (como o V8 do Node.js ou o compilador Go) podem variar ligeiramente em arquiteturas x86_64 ou em ambientes de nuvem virtualizados com recursos restritos (como AWS Lambda ou instâncias EC2 pequenas).

Outra limitação reside na métrica focada exclusivamente em "sucesso sob estresse". O estudo priorizou a vazão máxima, mas não aprofundou a análise de jitter (variação da latência) ou a recuperação de falhas intermitentes, fatores que são cruciais para a experiência do usuário final em sistemas distribuídos instáveis.

6.4 Trabalhos Futuros

Com base nas conclusões e limitações identificadas, diversas avenidas para trabalhos futuros podem ser exploradas para expandir o conhecimento nesta área. Uma extensão natural seria a complexificação do cenário de teste do lado do servidor. Futuros benchmarks poderiam incluir operações de banco de dados (CRUD), como nos estudos de Szewczyk e Skublewska-Paszkowska (2025) e Pankiv (2019), ou tarefas computacionalmente intensivas, como a serialização de grandes objetos JSON.

Essa abordagem seria fundamental para investigar se o gargalo observado na biblioteca *httpx* (Python) persiste em cenários mistos ou se a latência do banco de dados equalizaria o desempenho entre as linguagens, permitindo avaliar melhor o comportamento sob diferentes tipos de carga (CPU-bound vs. I/O-bound).

Outra direção importante seria a avaliação do desempenho em ambientes de rede não ideais. A introdução de latência, jitter e perda de pacotes simulados poderia revelar como a robustez e os mecanismos de tratamento de erro de cada biblioteca impactam o desempenho em condições mais realistas, testando a eficiência dos pools de conexão que se mostraram críticos para o desempenho do Elixir e do Go neste estudo.

Por fim, a pesquisa também poderia ser expandida para incluir uma análise mais holística, incorporando métricas de eficiência de recursos mais detalhadas, como o consumo de energia, seguindo a abordagem de Di Meglio e Starace (2024). Além disso, a inclusão de métricas qualitativas, como a complexidade do código e a facilidade de desenvolvimento, forneceria uma recomendação mais completa e contextualizada, ajudando a determinar se o ganho de vazão extrema justifica eventuais custos adicionais de implementação e manutenção.

6.5 Considerações Finais

Este trabalho demonstrou empiricamente que a escolha da biblioteca de requisição HTTP é uma decisão arquitetural crítica. A era das aplicações web de alta concorrência exige que o desenvolvimento de software vá além da funcionalidade e abrace a eficiência como

requisito não funcional primário. Os dados comprovam que plataformas com suporte nativo e maduro à concorrência, como Go e Elixir, oferecem uma vantagem competitiva substancial.

Contudo, a maior contribuição deste estudo talvez seja a evidência de que a evolução dentro dos ecossistemas é constante. O salto de desempenho da undici sobre a axios prova que há espaço para otimização mesmo em linguagens interpretadas. Para desenvolvedores e arquitetos, a mensagem final é clara: não assumam o desempenho baseado apenas na reputação da linguagem; testem, meçam e escolham as ferramentas que demonstram, com dados, serem capazes de suportar o futuro de suas aplicações.

REFERÊNCIAS

ABBADE, L. R. et al. Performance comparison of programming languages for Internet of Things middleware. Transactions on Emerging Telecommunications Technologies, v. 31, n. 12, p. e3891, 2020.

ABDULLAH, Mahmood F. Evaluating and scaling web programming languages: A comparison between PHP, Javascript. Rafidain Journal of Computer Sciences and Mathematics, v. 7, n. 3, p. 169–175, 2010.

ABDULLAH, A. A. Performance evaluation of web development technologies in php, and javascript. In: 2010 International Conference on Computer and Communication Engineering (ICCCE). Anais...2010.

APTE, V.; HANSEN, T.; REESER, P. Performance comparison of dynamic web platforms. Computer Communications, v. 26, n. 8, p. 888–898, 2003.

DI MEGLIO, S.; STARACE, L. L. L. Evaluating Performance and Resource Consumption of REST Frameworks and Execution Environments: Insights and Guidelines for Developers and Companies. IEEE Access, v. 12, p. 161649–161670, 2024.

DÍAZ CLAVIJO, D. A Practical Comparison of Agile Web Frameworks. Tese (Graduação) – Universidad de Las Palmas de Gran Canaria, 2014.

FERREIRA, J. A Javascript framework comparison based on benchmarking software metrics and environment configuration. Dissertação (Mestrado em Computação) – Dublin Institute of Technology, Dublin, 2018.

LEI, Kai; MA, Yining; TAN, Zhi. Performance Comparison and Evaluation of Web Development Technologies in PHP, Python and Node.js. In: 2014 IEEE 17th International Conference on Computational Science and Engineering. Chengdu, China: IEEE, 2014. p. 661-668.

NATH, K.; DHAR, S.; BASISTHA, S. Web 1.0 to Web 3.0 – Evolution of the Web and its various challenges. In: 2014 International Conference on Reliability, Optimization and Information Technology (ICROIT). Anais...Faridabad, India: IEEE, 2014. p. 87-90.

PANKIV, A. Concurrent benchmark system for web-frameworks on Python. Tese (Bacharelado) – Ukrainian Catholic University, Lviv, 2019.

RANJAN, A. et al. A comparative study between dynamic web scripting languages. International Journal of Computer Applications, v. 47, n. 20, p. 11-14, 2012.

RATANAWORABHAN, P.; LIVSHITS, B.; ZORN, B. G. JSMeter: Comparing the Behavior of Javascript Benchmarks with Real Web Applications. In: Proceedings of the 2010 USENIX

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.

STACK OVERFLOW. 2024 Developer Survey. Stack Overflow, 2024. Disponível em: https://survey.stackoverflow.co/2024/. Acesso em: 8 jul. 2025.

UZUN, E. et al. A comparative study of python libraries for data extraction from the web. In: 2018 2nd International Conference on Natural Language and Speech Processing (ICNLSP). Anais...2018.

VERMA, D. A Comparison of Web Framework Efficiency– Performance and network analysis of modern web frameworks. Tese (Bacharelado em Engenharia de Tecnologia da Informação e Comunicação) – Turku University of Applied Sciences, 2022.

VIITANEN, T. A Comparative Analysis of Modern Programming Languages in REST API Development. Tese (Bacharelado em Engenharia de TIC) – Tampere University of Applied Sciences, 2025.