Análise de Desempenho de Frameworks para Aplicações Distribuídas

André Luiz Gomes dos Santos
Universidade Tecnológica Federal do Paraná (UTFPR) - Curitiba
andreluiz@alunos.utfpr.edu.br

Christian Antunes Carneiro
Universidade Tecnológica Federal do Paraná (UTFPR) - Curitiba
christiancarneiro@alunos.utfpr.edu.br

José Henrique Ivanchechen
Universidade Tecnológica Federal do Paraná (UTFPR) - Curitiba
joseivanchechen@alunos.utfpr.edu.br

Ana Cristina Barreiras Kochem Vendramin
Universidade Tecnológica Federal do Paraná (UTFPR) - Curitiba
criskochem@utfpr.edu.br

Daniel Fernando Pigatto
Universidade Tecnológica Federal do Paraná (UTFPR) - Curitiba
pigatto@utfpr.edu.br

Abstract

As distributed systems become increasingly pervasive in everyday life, performance analyses that provide quantifiable insights into the efficiency of these systems are essential. This study proposes an analysis of the performance of backend frameworks that enable the development of such distributed systems. It is expected that the findings of this project will guide better technology choices for the development of future distributed applications. The results indicate that the choice of the ideal framework depends on the specific needs of the application. The Flask framework stands out as a robust and efficient option for scenarios requiring a high number of requests and handling of lightweight data. Actix is an interesting choice for applications that prioritize low resource consumption. ASP.NET can be a good option for handling large volumes of data, despite its tendency for high CPU usage.

Keywords

Backend Frameworks, Web Development, Distributed Systems, Performance Analysis

1 Introdução

Em um mundo cada vez mais conectado, onde a demanda por serviços digitais e aplicações complexas cresce exponencialmente, sistemas distribuídos e APIs (*Application Programming Interfaces*) tornaram-se elementos essenciais de infraestrutura tecnológica. A crescente necessidade de processar grandes volumes de dados e garantir a alta disponibilidade de serviços exige uma análise cuidadosa do desempenho de sistemas, especialmente na escolha de tecnologias e *frameworks* utilizados em seu desenvolvimento.

Atualmente, existem diversas linguagens de programação voltadas para a criação de APIs, cada uma funcionando de maneira específica no que diz respeito às estruturas de dados e ao seu processamento interno. Essas particularidades podem influenciar tanto a escrita quanto a execução dos programas, além de impactar o desempenho geral do sistema. Um *framework* é definido como uma estrutura de *software* projetada para ser reutilizável, contendo classes e ferramentas destinadas a resolver problemas em um domínio específico [1]. Ele também oferece recursos que agilizam o desenvolvimento e a melhoria de aplicações. Já os *frameworks* de *backend* têm como principal função lidar com a lógica de negócio, gerenciando o processamento e o armazenamento de dados [2].

Um fator crucial para garantir uma boa qualidade de serviço é o tempo de resposta, que influencia diretamente como os usuários percebem o tempo de carregamento de páginas ou o tempo de busca em um sistema [3]. Considerando a experiência do usuário final, torna-se evidente que a análise de desempenho dos *frameworks backend*, com foco na métrica de tempo de resposta, é fundamental. Além disso, a avaliação do comportamento do serviço em relação ao consumo de recursos computacionais é outro aspecto fundamental ao se analisar a escalabilidade de uma aplicação [4].

Este artigo destaca a importância de escolher um *framework backend* no desenvolvimento de aplicações web, com foco na análise de desempenho de cinco *frameworks* amplamente utilizados: Spring Boot (Java), ASP.NET Core (C#), Flask (Python), Express.js (JavaScript) e Actix (Rust). A seleção desses *frameworks* se baseia em sua relevância no mercado e ampla adoção por desenvolvedores, como mostra a pesquisa realizada pelo StackOverflow em 2021 [5].

O objetivo principal deste artigo é oferecer uma compreensão mais aprofundada sobre qual *framework* pode ser mais adequado para diferentes cenários de desenvolvimento web, levando em consideração métricas de desempenho, como o número de requisições atendidas por segundo, o consumo de CPU (*Central Processing Unit*) e o uso de memória RAM (*Random Access Memory*).

O restante deste artigo está organizado da seguinte forma: na Seção 2 são apresentados os conceitos fundamentais que sustentam e embasam este trabalho, fornecendo a base teórica necessária para sua compreensão, seguido pela Seção 3 que trata de alguns trabalhos relacionados com o atual documento. A Seção 4 descreve a metodologia adotada, detalhando o projeto do sistema *backend* e o fluxo de testes. Na Seção 5 são discutidos os resultados obtidos durante a execução do experimento. Por fim, a Seção 6 reúne as principais conclusões derivadas dos resultados apresentados e sugere direções para trabalhos futuros.

2 Fundamentação teórica

Um sistema distribuído é um sistema de computadores em rede no qual os processos e os recursos estão distribuídos entre múltiplos computadores com o objetivo de melhorar a confiabilidade, escalabilidade e eficiência [6].

A principal motivação para construir e usar sistemas distribuídos é o compartilhamento de recursos. Esses recursos englobam desde componentes de hardware, como discos e impressoras, até entidades definidas por software, como arquivos, bancos de dados e objetos de dados de todos os tipos [7]. Porém, desenvolver esses sistemas apresenta inúmeros desafios como lidar com a heterogeneidade de componentes, segurança, gerenciamento de falhas, escalabilidade e controle de concorrência.

A escalabilidade é um dos maiores desafios a serem enfrentados quando se desenvolve um sistema, referindo-se à capacidade deste de se adaptar ao crescimento, seja em termos de acessos, dados ou funcionalidades, sem comprometer o seu desempenho. Os sistemas distribuídos frequentemente utilizam uma arquitetura cliente-servidor, onde os servidores fornecem serviços específicos em resposta a solicitações de clientes. Nesse contexto, a arquitetura REST (do inglês, *Representation State Transfer*), por exemplo, é uma *middleware* que visa aumentar a independência e escalabilidade de um sistema distribuído [8].

Linguagens de programação são ferramentas utilizadas para desenvolver sistemas computacionais, permitindo a criação de programas que podem ser compreendidos e mantidos por outras pessoas, facilitando a colaboração e manutenção do código [9].

As linguagens de programação possuem características distintas: algumas são projetadas para facilitar uma escrita rápida e intuitiva, mas podem apresentar desafios na execução ou na definição de métodos eficazes para testar a aplicação [9].

Java é uma linguagem orientada a objetos baseada em classes, com variáveis chamadas "campos" para representar seu estado e métodos para manipulação interna e interação externa. Sua execução ocorre em uma máquina virtual Java, garantindo portabilidade entre diferentes plataformas como Windows, Linux e macOS [10]. Similarmente, C# também é uma linguagem tipada e orientada a objetos, com suporte a recursos como coleta de lixo, tipos anuláveis e operações assíncronas [11]. Python, por sua vez, caracteriza-se por ter um sistema de tipagem fraca e dinâmica, além de ser uma linguagem interpretada permitindo execução imediata do código [12]. Já o JavaScript, uma linguagem dinâmica e multi-paradigma, é amplamente empregada no desenvolvimento web e de APIs, com tipagem inferida em tempo de execução [13]. Por fim, Rust, inicialmente desenvolvida pela Mozilla, equilibra segurança com controle detalhado sobre o gerenciamento de recursos [14].

Por meio dessas linguagens de programação, pode-se criar abstrações e outros tipos de *middlewares* para realizar diversas operações, como por exemplo interagir com sistemas distribuídos. *Frameworks* são estruturas de *software* que fornecem bibliotecas, utilitários e modelos para simplificar o desenvolvimento de aplicações, tornando o processo mais eficiente ao reduzir tarefas repetitivas e complexas [1]. Esses *frameworks* são reutilizáveis e podem ser classificados em dois tipos principais: *frontend*, focado na interação com o usuário, e *backend*, responsável por implementar a lógica de negócios [2]. No *backend*, destacam-se alguns *frameworks* amplamente conhecidos, como: Spring Boot (Java) [15, 16], ASP.NET Core (C#) [17], Flask (Python) [18, 19], Express.js (JavaScript) [20] e Actix (Rust) [21, 22]. Esses *frameworks* são utilizados no desenvolvimento de sistemas web e microsserviços devido à sua eficiência e popularidade.

A escolha dessas cinco linguagens de programação e *frameworks* foi motivada pela sua independência de ambientes específicos de execução, o que garante maior flexibilidade e portabilidade, sendo amplamente suportadas em diferentes sistemas operacionais e ambientes de desenvolvimento, sem a necessidade de adaptações significativas [10, 11, 23–25].

Uma maneira de disponibilizar serviços utilizando esses *frameworks* em uma rede é por meio do uso de contêineres. Essa tecnologia de virtualização em nível de sistema operacional permite a execução de aplicativos e serviços de forma isolada, proporcionando um ambiente consistente e portátil [26]. Um contêiner encapsula o código do aplicativo, suas dependências e configurações em uma unidade autônoma, que pode ser implementada e executada de maneira consistente em diferentes ambientes [26]. Neste projeto, o *Docker* foi adotado como serviço de conteinerização das aplicações, devido ao fato de ser um ecossistema amplamente consolidado [27].

Para a troca de dados, utilizam-se estruturas MIME (*Multipurpose Internet Mail Extensions*), definidas pela RFC (*Request for Comments*) 2045. Essas estruturas padronizam o envio de diversos tipos de dados (texto, imagens, áudio) no corpo da mensagem, acompanhados do cabeçalho *Content-Type* que informa o tipo MIME para que o receptor possa interpretar [7]. Além disso, também existem diferentes formatos de serialização de dados, como JSON e Protobuf. JSON é um formato textual, independente de linguagem e de fácil compreensão [28]. Protobuf, desenvolvido pela Google, é conhecido por ser leve e eficiente, ideal para cenários de alto desempenho [29, 30].

O protocolo HTTP (*Hypertext Transfer Protocol*) define métodos para realizar requisições. O método GET acessa informações, POST envia dados, PUT altera dados e DELETE remove dados do servidor.

O HTTP normalmente utiliza o TCP (*Transmission Control Protocol*) como protocolo de transporte. Sendo um protocolo sem estado, o HTTP trata cada requisição de forma independente. Isso implica que o HTTP 1.0 emprega conexões não persistentes sobre o TCP, estabelecendo uma nova conexão TCP para cada requisição feita. No entanto, no HTTP 1.1, a persistência das conexões foi introduzida, permitindo que uma mesma conexão TCP seja mantida aberta para múltiplas requisições [31].

Softwares de *benchmark* são ferramentas utilizadas para avaliar o desempenho de sistemas, através de métricas como tempo de resposta e requisições atendidas por segundo. Ferramentas como *Apache JMeter* [32] e *ApacheBench* [33] simulam cargas de trabalho para testar servidores e identificar gargalos de desempenho. O Docker SDK é outra ferramenta relevante, usada para mensurar o uso de recursos, como CPU e RAM, em aplicações conteinerizadas [34]. A biblioteca do Docker SDK foi escolhida neste trabalho.

3 Trabalhos Relacionados

Existem poucos trabalhos atualmente que realizam uma análise comparativa entre diversos *frameworks* de forma aprofundada.

Lei et al. [35] realizaram uma comparação sobre concorrência ou, mais especificamente, o número de requisições que sistemas construídos em PHP, Python-Web e Node.js conseguiriam suportar em média a cada segundo, bem como o tempo de resposta das requisições. Os testes simularam cenários nos quais foram realizados o *login* de usuários, criptografia de senhas, retorno de "Hello World" e cálculo da sequência de Fibonacci. Variando o número de clientes, observou-se a variação na média de respostas por segundo e no tempo de resposta para cada uma das implementações do sistema.

Os softwares de teste utilizados para as requisições e medições foram o *ApacheBench* [33] e *Loadrunner* [36]. Os resultados dos testes demonstraram que o PHP apresentou uma capacidade bem menor de processar requisições juntamente com um maior tempo de resposta em relação às outras tecnologias. No estudo também foi observado que o desempenho do Node.js foi, em média, superior ao Python-Web nas duas métricas.

Sharma et al. [37] realizaram uma comparação entre os *frameworks backend* Django e Node.js. O artigo analisou o comportamento dos *frameworks* em relação ao número de transações por segundo e o tempo de resposta variando o número de clientes simultâneos. Foram desenvolvidas três implementações de diferentes algoritmos para a realização dos testes. Na primeira implementação foi programado o retorno de uma mensagem "Hello World" como resposta, já na segunda foi feito o carregamento de uma imagem e na terceira foi calculada a soma de uma progressão harmônica. Utilizou-se o programa *ApacheBench* [33] e *Siege* [38] para medição dos dados para o tempo de processamento e quantidade de requisições nos testes de *benchmark*. O resultado dos testes mostrou nos três cenários uma similaridade entre os *frameworks* quanto à quantidade de transações média por segundo. Contudo, em relação ao tempo de resposta, o Node.js teve um desempenho significativamente superior com um tempo 30 vezes menor em alguns casos.

Dhalla [4] realizou a comparação entre aplicações implementadas com os *frameworks* Spring em Java e ASP.NET Core em C# no qual foi avaliada a média do tempo de resposta para o cliente e consumo de recursos computacionais como CPU e memória RAM. No estudo simulou-se de 1.000 até 64.000 clientes ativos variando o número de requisições feitas nos dois sistemas. O software de teste utilizado para realizar as requisições e medições foi o *Apache JMeter* [32]. Os resultados dos testes demonstraram que a capacidade dos dois *frameworks* quanto à capacidade de processamento do número de requisições simultâneas é bem próxima, contudo o ASP.NET apresentou um melhor desempenho já que apresentou um menor consumo de recursos junto a um menor tempo de resposta nos mesmos cenários do Spring.

Por fim, o estudo de Challapalli et al. [39] realizou a comparação de desempenho entre servidores desenvolvidos em Python e Node.js. Para a realização dos testes foi utilizado um dispositivo no qual os dois servidores foram executados junto do software de *benchmark*. Nesse experimento foram utilizados dois *softwares* de *benchmark*, *Locust* [40] e *Autocannon* [41], desenvolvidos em Python e Node.js, respectivamente. O ambiente desses testes foi composto por 10 usuários simulados pelos *softwares*, verificando-se, similarmente aos estudos anteriores, a média de requisições por segundo e também o tempo de resposta. Os resultados, em um intervalo de 30 segundos, mostraram que o Node.js teve um desempenho muito superior ao Python em ambos os parâmetros, uma vez que no estudo foi verificada a capacidade do Node.js tratar em média 250 vezes mais requisições do que o Python e também com o tempo de resposta médio quase 7 vezes inferior.

Diante desse levantamento sobre o estado da arte na análise de desempenho de diferentes tecnologias para *backend*, o presente trabalho propõe uma análise robusta de cinco diferentes *frameworks* para desenvolvimento web: Spring, ASP.NET, Express.js, Flask e Actix, verificando, dado cenários isonômicos de hardware, o desempenho de cada um em tarefas específicas, a partir do consumo de recursos computacionais ao longo do tempo e o número médio de requisições suportado por segundo.

4 Metodologia

Foram desenvolvidas aplicações análogas nos cinco *frameworks* propostos e estas foram executadas em um ambiente de virtualização para isolamento e controle dos recursos computacionais. A Tabela 1 resume as versões utilizadas para cada linguagem e *framework*. Também foi utilizada a versão mais atual do gerenciador de pacotes de cada linguagem.

Tabela 1: Versões utilizadas para o desenvolvimento

Versão do *framework*	Versão da linguagem
Actix 4	Rust 1.77
ASP.NET 6.0	C# 10
Express 4.18.2	Node.js 18
Flask 3.0	Python 3.10.12
Spring Boot 2.7.1	JDK 21

Para a análise de desempenho dos *frameworks*, foram desenvolvidos cinco sistemas *backend*, cada um utilizando um dos *frameworks* mencionados. Os sistemas foram projetados para simular operações comuns em aplicações web, como manipulação de imagens, serialização de dados e cálculos matemáticos. As seguintes funcionalidades foram implementadas como *endpoints*:

- /status/ok: serve como uma base para medir o desempenho dos *frameworks*, respondendo apenas com o código 200 para indicar que a requisição foi bem-sucedida;
- /simulation/json: recebe um corpo de mensagem em formato JSON, desserializa-o, e o envia de volta ao cliente após a serialização;
- /simulation/protobuf: similar ao anterior, esse *endpoint* é responsável por recebe um corpo de mensagem em formato JSON, desserializa-o, e o envia de volta ao cliente após a serialização;
- /simulation/harmonic-progression?n=*termos*: calcula uma progressão harmônica com base no número de *termos* especificado;
- /static/small-image.png: responsável por retornar uma imagem pequena, um arquivo estático, de 100 *kilobytes*;
- /static/big-image.png: responsável por retornar uma imagem grande, um arquivo estático, de 10 *megabytes*;
- /image/save-big-image: responsável por receber uma imagem de 10 *megabytes* e salvá-la em disco.

Os testes foram executados em um ambiente controlado, utilizando Docker para garantir isonomia de *hardware* e *software*, onde cada um dos contêineres foi limitado a 1 GB de memória RAM e 1 núcleo de CPU. Essa escolha permitiu simular condições de recursos semelhantes às encontradas em ambientes de produção típicos onde os recursos também são limitados.

Cada *framework* foi submetido a diferentes cargas de trabalho, simulando um número crescente de requisições simultâneas. As métricas de consumo de CPU e RAM foram coletadas utilizando a biblioteca Docker SDK para Python. Para cada carga de teste, isto é,

a execução de um número N de requisições ao *endpoint* selecionado, repete-se o mesmo teste 30 vezes juntamente com o processo de reinício do *container* entre as cargas de teste. Além disso, é realizada uma pausa de 30 segundos após cada reinicialização que hospeda o sistema *backend* a ser testado, com o objetivo de desconsiderar o tempo de inicialização do sistema.

Após a realização de cada teste com a carga especificada, os resultados são armazenados em um arquivo para posterior análise por um *script* também desenvolvido em Python 3.10. Esse *script* é responsável por gerar gráficos com base nos dados obtidos (requisições atendidas por segundo, utilização de CPU e memória RAM) para cada *endpoint* chamado em cada *framework*.

Foi utilizada uma máquina para hospedar tanto o Docker que executou os contêineres de aplicação quanto a execução do *script* de testes. O sistema operacional escolhido foi o Debian GNU/Linux 12, juntamente com um processador AMD Ryzen 7 PRO 1700 @ 3.0GHz (8 núcleos/16 *threads*), com 16 GB de memória RAM e um SSD Xray de 1TB para armazenamento.

5 Resultados

Esta seção tem como objetivo apresentar os resultados dos experimentos realizados com as aplicações desenvolvidas utilizando os *frameworks* Spring, ASP.NET, Express.js, Flask e Actix, detalhando o desempenho de cada um nos diferentes *endpoints* testados.

Conforme ilustrado pela Figura 1, o *framework* ASP.NET obteve o melhor desempenho inicial, atingindo cerca de 650 requisições por segundo com cargas entre 15 mil e 25 mil requisições. Os *frameworks* Express.js e Actix apresentaram desempenho similar e crescente, acompanhando o aumento no número de requisições. O Flask manteve desempenho estável acima de 550 requisições por segundo. O Spring obteve o pior desempenho, iniciando com 350 requisições por segundo e atingindo 525 requisições ao final.

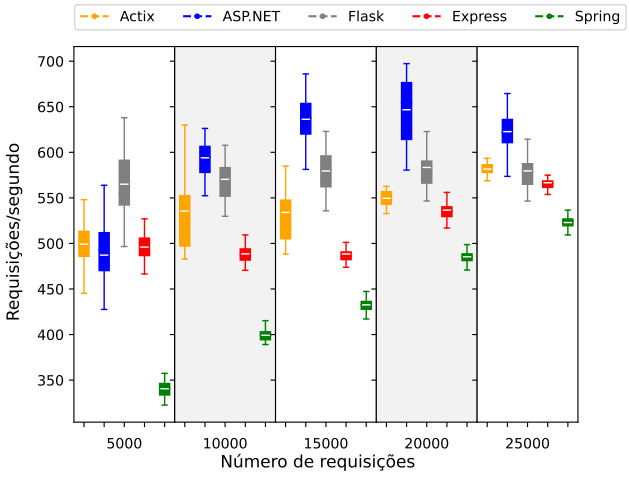

Figura 1: Requisições atendidas por segundo para o *endpoint* /status/ok.

Na Figura 2 é possível observar que o Flask apresentou o melhor desempenho em todas as cargas, com médias consistentes. Os *frameworks* Express.js e Actix mantiveram um desempenho semelhante, com aumento gradual. O ASP.NET mostrou grande variação entre os valores máximos e mínimos, embora a média tenha se mantido acima de 400 requisições por segundo. O Spring teve um desempenho inferior, mas próximo ao ASP.NET na carga máxima.

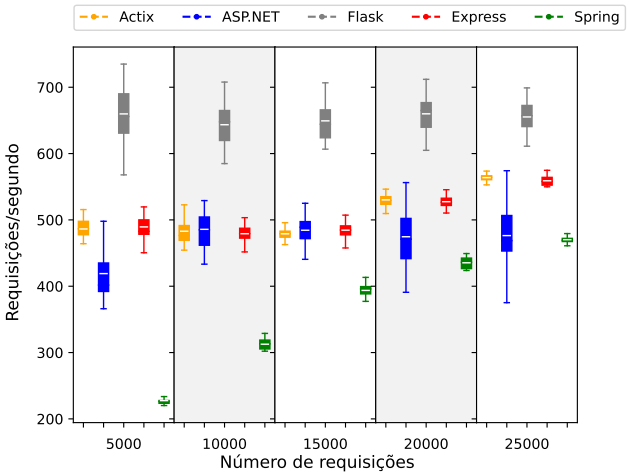

Figura 2: Requisições atendidas por segundo para o *endpoint* /simulation/json.

A Figura 3 retrata resultados ligeiramente similares ao *endpoint* da Figura 2, com o Flask possuindo melhores médias, seguido pelo Express.js e o Actix. Os *frameworks* ASP.NET e Spring apresentaram médias inferiores aos outros sistemas testados.

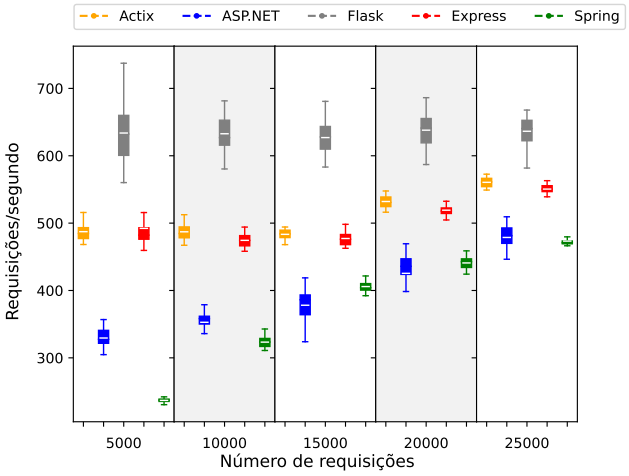

Figura 3: Requisições atendidas por segundo para o *endpoint* /simulation/protobuf.

Novamente, assim como visto anteriormente na Figura 3, o Flask lidera com as melhores médias na Figura 4, com uma média estável próxima de 600 requisições por segundo. O Express.js, o Actix e o ASP.NET apresentaram suas médias próximas de 450 e 500 requisições atendidas por segundo. O Spring novamente apresentou o pior desempenho.

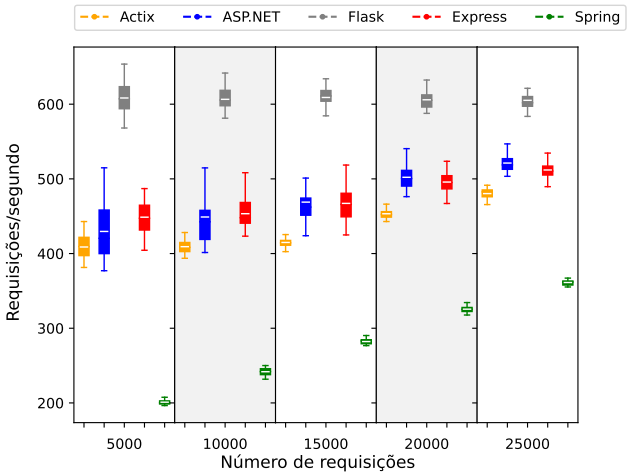

Figura 4: Requisições atendidas por segundo para o *endpoint* /static/small-image.png.

Para os resultados obtidos no *endpoint* da Figura 5 observamos que os *frameworks* Actix, ASP.NET e Flask apresentaram resultados similares, com médias entre 30 e 35 requisições atendidas por segundo. O Express.js teve suas médias levemente inferiores. O Spring teve desempenho significativamente menor, com média de 5 requisições por segundo. A carga máxima para os testes foi reduzida devido ao alto processamento exigido.

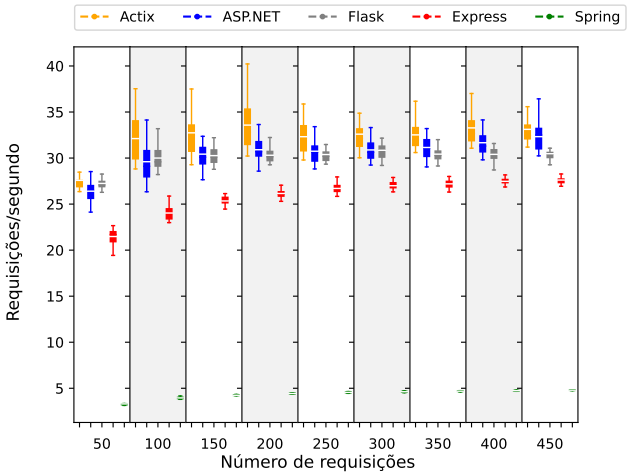

Figura 5: Requisições atendidas por segundo para o *endpoint* /static/big-image.png.

Os *frameworks* ASP.NET e Express.js tiveram desempenho próximo a 20 requisições por segundo, como mostrado na Figura 6. O *framework* Actix teve desempenho inicial superior, mas reduziu nas cargas maiores, com grande variação entre a máxima e a mínima. O Flask apresentou comportamento similar ao Actix, com uma queda acentuada no desempenho após 300 requisições simultâneas. O Spring teve o menor desempenho, com média abaixo de 5 requisições por segundo após 100 requisições simultâneas.

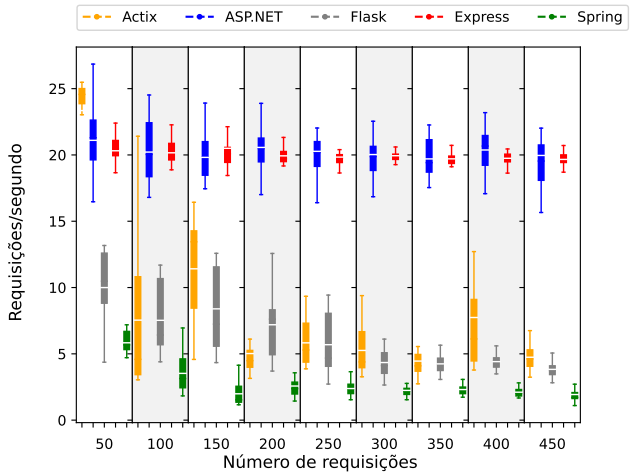

Figura 6: Requisições atendidas por segundo para o *endpoint* /image/save-big-image.

Os *frameworks* Actix e Express.js obtiveram as melhores médias, iniciando em 500 e chegando a quase 600 requisições atendidas por segundo conforme ilustrado na Figura 7. Já os *frameworks* ASP.NET e Spring obtiveram resultados inferiores, porém com aumento gradual. O Flask obteve o pior desempenho, com média próxima de 100 requisições atendidas por segundo.

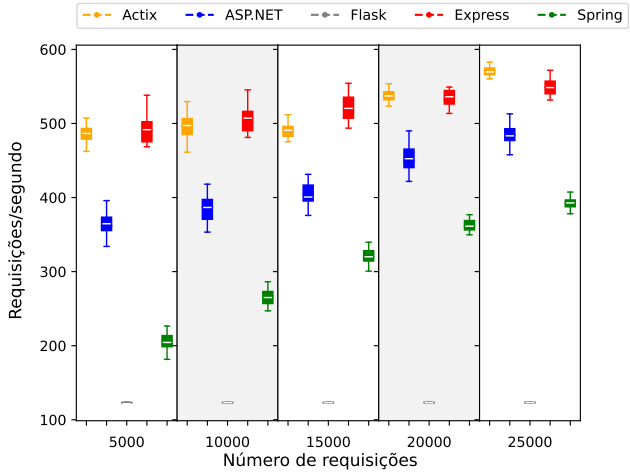

Figura 7: Requisições atendidas por segundo para o *endpoint* /simulation/harmonic-progression?n=100000.

A análise de consumo de CPU e RAM em diferentes *frameworks* evidenciou variações significativas de desempenho. O *framework* Actix por exemplo demonstrou um consumo de CPU relativamente baixo, com uma média entre 10% e 30% na maioria dos testes, exceto em *endpoints* mais exigentes, como o retorno de imagens grandes (Figura 12), onde o uso subiu para uma faixa de 40% a 60%. Seu consumo de RAM também foi baixo por todos os *endpoints* testados, frequentemente próximo de 1%, com exceção de operações mais

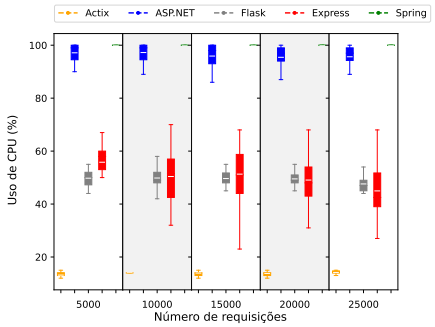

Figura 8: Consumo de CPU para o *endpoint* /status/ok.

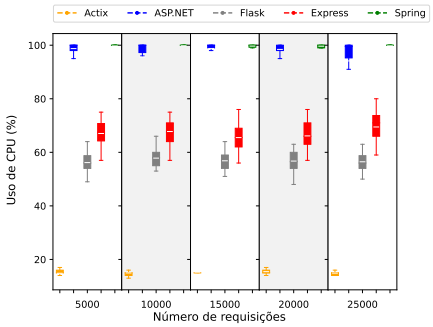

Figura 9: Consumo de CPU para o *endpoint* /simulation/json.

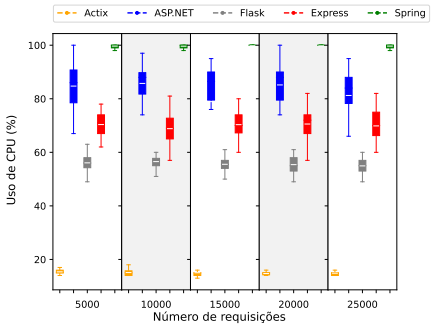

Figura 10: Consumo de CPU para o *endpoint* /simulation/protobuf.

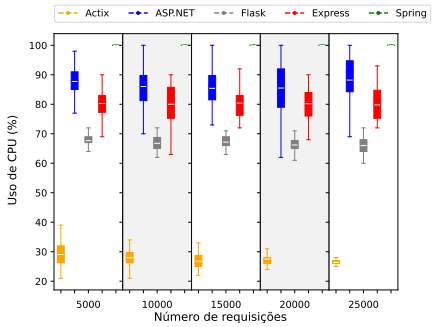

Figura 11: Consumo de CPU para o *endpoint* /image/small-image.png.

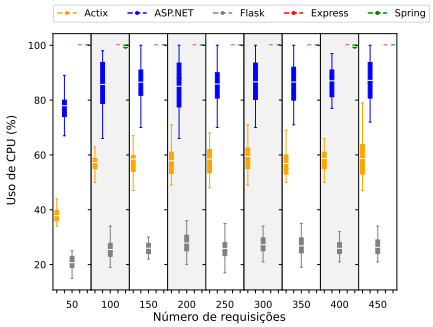

Figura 12: Consumo de CPU para o *endpoint* /image/big-image.png.

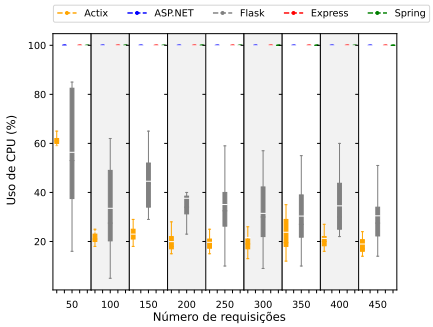

Figura 13: Consumo de CPU para o *endpoint* /image/save-big-image.

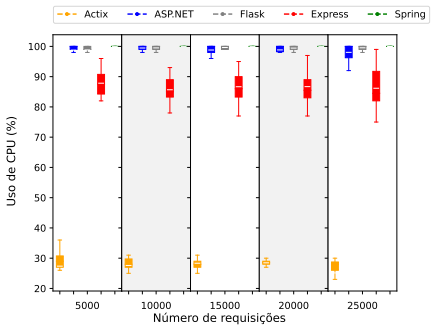

Figura 14: Consumo de CPU para o *endpoint* /simulation/harmonic-progression?n=100000.

pesadas como a de escrita de imagem grande (Figura 20), onde chegou a superar 10%.

O ASP.NET mostrou desempenho com alto consumo de CPU, frequentemente próximo de 100%, em diversos *endpoints* como /status/ok (Figura 8) e /image/big-image.png (Figura 12), evidenciando que o *framework* tende a sobrecarregar o processador em tarefas intensivas. Em termos de uso de consumo de memória RAM, o ASP.NET apresentou variações amplas, com picos de acima 60% quando envolvia escrita em disco (Figura 20), mas com médias em torno de 2% a 4% nos demais *endpoints*.

O *framework* Spring também apresentou um consumo constantemente elevado de CPU, com uma média próxima de 100% em todos os testes realizados, indicando uma demanda alta de processamento. A utilização de RAM variou entre 12% e 15% na maioria dos casos, mas caiu para 7% a 10% em *endpoints* relacionados a serialização e desserialização de dados (Figura 16 e Figura 17) e leitura de imagens grandes (Figura 19). Ainda assim, o *framework* se destacou por manter um consumo consistente em operações de maior carga.

O Express.js obteve um comportamento onde a variação de consumo de CPU aumentava à medida que a carga de teste aumentava, com um início próximo de 60% e finalizando em torno de 30% e 70% (Figura 8). No entanto, em operações mais intensivas, como o retorno de imagens grandes e progressão harmônica como demonstrados nas Figura 12 e Figura 14, o uso de CPU chegou a 100%. O consumo de RAM se manteve entre 3% e 6% e próximo de 25% ao lidar com operação de escrita em disco (Figura 13).

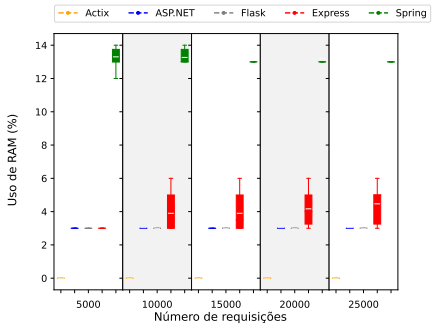

Figura 15: Consumo de memória RAM para o *endpoint* /status/ok.

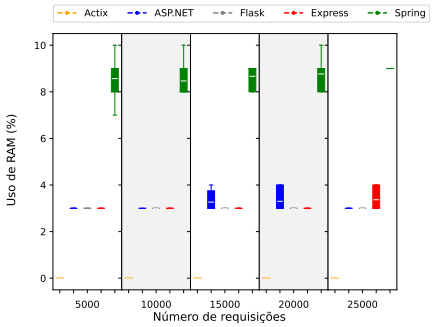

Figura 16: Consumo de memória RAM para o *endpoint* /simulation/json.

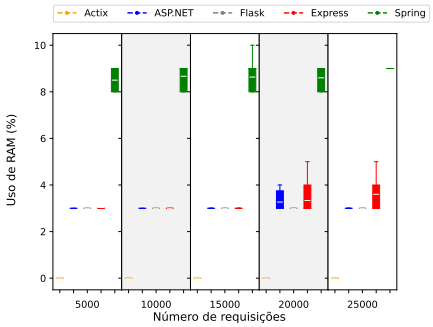

Figura 17: Consumo de memória RAM para o *endpoint* /simulation/protobuf.

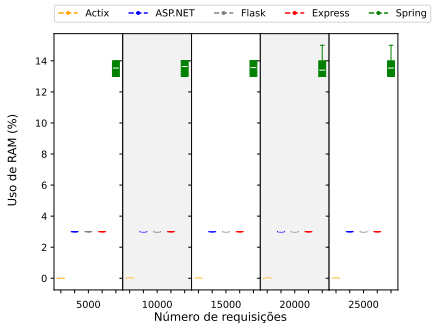

Figura 18: Consumo de memória RAM para o *endpoint* /image/small-image.png.

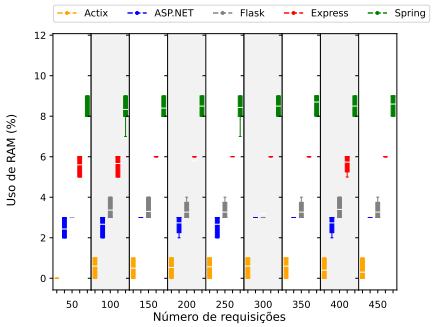

Figura 19: Consumo de memória RAM para o *endpoint* /image/big-image.png.

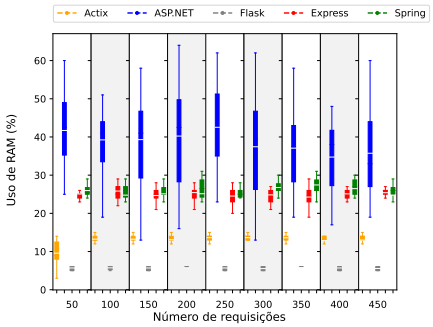

Figura 20: Consumo de memória RAM para o *endpoint* /image/save-big-image.

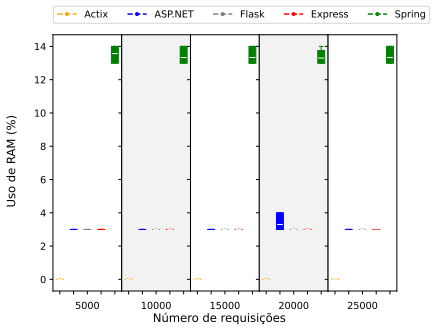

Figura 21: Consumo de memória RAM para o *endpoint* /simulation/harmonic-progression?n=100000.

O Flask apresentou um perfil de consumo de CPU que, em muitos casos, se assemelhou ao do Express.js, com médias estáveis em torno de 50% (Figura 8) e até 70% para a leitura de uma imagem pequena (Figura 11). Em tarefas mais complexas, como a progressão harmônica, o uso de CPU alcançou 100% (Figura 14). Quanto à utilização de memória RAM, Flask foi relativamente eficiente, com consumo médio de 3% na maioria dos testes (Figura 16), embora tenha mostrado uma variação mais significativa na escrita de imagens grandes, onde o uso subiu para cerca de 5% (Figura 19).

No geral, o Actix foi o mais eficiente em termos de uso de CPU e RAM, enquanto os *frameworks* ASP.NET e Spring se destacaram pelo alto consumo de CPU. O Express.js e o Flask mostraram comportamentos intermediários, com variações de consumo dependendo da carga de trabalho e do tipo de operação.

6 Conclusão

No presente trabalho foi realizado um estudo para analisar o desempenho dos *frameworks* Actix, ASP.NET, Express.js, Flask e Spring. Para isso, foram desenvolvidos diversos sistemas utilizando diferentes *endpoints* que ajudaram a analisar as métricas escolhidas para este estudo, sendo elas o número de requisições atendidas por segundo, percentual de utilização de CPU e o percentual de utilização de memória RAM.

A análise dos resultados realizados com diferentes *frameworks* e *endpoints* revela pontos importantes sobre o desempenho e a escalabilidade dessas tecnologias em cenários variados.

Por linhas gerais, o *framework* Flask se destacou, apresentando um alto número de requisições atendidas por segundo, especialmente em *endpoints* simples, processamento de dados (JSON e Protobuf) e manipulação de imagens pequenas. O Actix demonstrou grande eficiência no consumo de CPU e RAM na maioria dos testes, consolidando sua reputação como *framework* leve e de alto desempenho. No entanto, o Actix apresentou inconsistências no *endpoint* de escrita de imagem grande. O *framework* ASP.NET, apesar do alto consumo de CPU, produziu resultados com um bom desempenho em cenários com grande volume de dados e manipulação de imagens grandes. Já o Express.js obteve resultados satisfatórios na maioria dos testes, mas não se destacou em nenhuma métrica específica. Por fim, o *framework* Spring apresentou o menor número de requisições atendidas por segundo e alto consumo de CPU, tornando-o menos atrativo em termos de requisições atendidas por segundo.

É importante ressaltar que os resultados obtidos podem variar dependendo do *hardware*, da configuração do ambiente e da implementação específica da aplicação. Como trabalhos futuros, pretende-se avaliar o desempenho dos *frameworks* em ambientes de produção reais, analisar aspectos de escalabilidade e tolerância a falhas em cenários complexos, explorar a inclusão de *endpoints* com processamento mais intensivo e serviços em tempo real, realizar a avaliação de desempenho em diferentes provedores de nuvem, investigar a integração dos *frameworks* com arquiteturas de microsserviços, e analisar a experiência do desenvolvedor e a curva de aprendizado de cada *framework*.

Referências

[1] Michael Mattsson. *Evolution and composition of object-oriented frameworks*. PhD thesis, Blekinge Institute of Technology, 2000.
[2] IH Madurapperuma, MS Shafana, and MJA Sabani. State-of-art frameworks for front-end and back-end web development. 2022.
[3] Daniel A Menascé. Load testing of web sites. *IEEE internet computing*, 6(4):70–74, 2002.
[4] Hardeep Kaur Dhalla. A performance comparison of restful applications implemented in spring boot java and ms.net core. *Journal of Physics: Conference Series*, 1933(1), Jun 2021. doi: 10.1088/1742-6596/1933/1/012041.
[5] StackOverflow. Stack overflow developer survey 2021, 2021. URL https://insights.stackoverflow.com/survey/2021#section-most-popular-technologies-web-frameworks.
[6] Maarten van Steen and Andrew S. Tanenbaum. *Distributed Systems*. distributed-systems.net, 4th edition, 2023. URL https://www.distributed-systems.net/index.php/books/ds4/.
[7] George Coulouris, Jean Dollimore, Tim Kindberg, and Gordon Blair. *Distributed Systems: Concepts and Design*. Addison-Wesley Publishing Company, USA, 5th edition, 2011. ISBN 0132143011.
[8] Roy T. Fielding and Richard N. Taylor. Principled design of the modern web architecture. *ACM Trans. Internet Technol.*, 2(2):115–150, may 2002. ISSN 1533-5399. doi: 10.1145/514183.514185. URL https://doi.org/10.1145/514183.514185.
[9] John C Mitchell and Krzysztof Apt. *Concepts in programming languages*. Cambridge University Press, 2003.
[10] Ken Arnold, James Gosling, and David Holmes. *The Java programming language*. Addison Wesley Professional, 2005.
[11] Bill Wagner, Tom Dykstra, Rodrigo C. M. Santos, and Kent Sharkey. Um tour por c# – visão geral, Feb 2023. URL https://learn.microsoft.com/pt-br/dotnet/csharp/tour-of-csharp/.
[12] Guido van Rossum and Jelke de Boer. Interactively testing remote servers using the python programming language. *CWI Quarterly*, 4(4):283–304, December 1991.
[13] MozDevNet. Javascript, 2023. URL https://developer.mozilla.org/en-US/docs/Web/JavaScript.
[14] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. Rustbelt: Securing the foundations of the rust programming language. *Proceedings of the ACM on Programming Languages*, 2(POPL):1–34, 2017.
[15] Rahul Gunkar. Introduction to spring boot, 2024. URL https://www.geeksforgeeks.org/introduction-to-spring-boot/.
[16] Spring. Spring boot, 2024. URL https://docs.spring.io/spring-boot/.
[17] Microsoft. O que é asp.net?, 2024. URL https://dotnet.microsoft.com/pt-br/learn/aspnet/what-is-aspnet.
[18] Armin Ronacher. Opening the flask, 2011. URL http://mitsuhiko.pocoo.org/flask-pycon-2011.pdf.
[19] Flask. Welcome to flask, 2024. URL https://flask.palletsprojects.com.
[20] T. J. Holowaychuk. Express, 2024. URL https://github.com/expressjs/express.
[21] Nikolay Kim. Actix web releases, 2017. URL https://github.com/actix/actix-web/releases/tag/v0.2.0.
[22] Actix. Welcome | actix web, 2024. URL https://actix.rs/docs/.
[23] Mark Lutz. *Programming python*. O'Reilly, 2019.
[24] Douglas Crockford. *JavaScript: The Good Parts*. O'Reilly, 2008.
[25] Steve Klabnik and Carol Nichols. *The Rust programming language*. No Starch Press, 2023.
[26] RedHat. What is a container?, 2023. URL https://www.redhat.com/en/topics/containers.
[27] Docker. Why docker?, 2023. URL https://www.docker.com/why-docker.
[28] JSON. Introducing json, 2023. URL https://www.json.org/.
[29] Google. Protocol buffers, 2024. URL https://protobuf.dev/.
[30] K. Afsari, C. M. Eastman, and D. Shelden. Data transmission opportunities for collaborative cloud-based building information modeling. *Blucher Design Proceedings*, 2016. doi: 10.5151/despro-sigradi2016-448.
[31] James F. Kurose and Keith W. Ross. *Computer Networking: A Top-Down Approach*. Pearson, 8th edition, 2020.
[32] Apache. Apache jmeter, 2023. URL https://jmeter.apache.org/.
[33] Apache. Apache bench, 2023. URL https://httpd.apache.org/docs/2.4/programs/ab.html.
[34] Docker SDK. Docker sdk, 2024. URL https://docker-py.readthedocs.io/en/stable/.
[35] Kai Lei, Yining Ma, and Zhi Tan. Performance comparison and evaluation of web development technologies in php, python, and node.js. *2014 IEEE 17th International Conference on Computational Science and Engineering*, Dec 2014. doi: 10.1109/cse.2014.142.
[36] Micro Focus. Loadrunner, May 2023. URL https://www.microfocus.com/en-us/products/loadrunner-professional/overview.
[37] Dr Anupam Sharma, Archit Jain, Ayush Bahuguna, and Deeksha Dinkar. Comparison and evaluation of web development technologies in node.js and django. *International Journal of Science and Research (IJSR)*, 9(ue 12):1416–1420,, 12 2020. doi: 10.21275/SR201202223534.
[38] Jeff Fulmer. Siege, Sep 2014. URL https://github.com/JoeDog/siege.
[39] Sai Sri Challapalli, Prakarsh Kaushik, Shashikant Suman, Basu Dev Shivahare, Vimal Bibhu, and Amar Deep Gupta. Web development and performance comparison of web development technologies in node.js and python. *2021 International Conference on Technological Advancements and Innovations (ICTAI)*, Nov 2021. doi: 10.1109/ictai53825.2021.9673464.
[40] Jonatan Heyman. Locust.io, 2023. URL https://locust.io/.
[41] Matteo Collina. Autocannon, Mar 2016. URL https://github.com/mcollina/autocannon.