Performance Evaluation of Back-End Frameworks: A Comparative Study

Klayver Ximenes Carmo
klayverx@gmail.com
Universidade Federal do Ceará
Sobral, Ceará, Brasil

Fischer Jonatas Ferreira
fischer.ferreira@sobral.ufc.br
Universidade Federal do Ceará
Sobral, Ceará, Brasil

Eduardo Figueiredo
figueiredo@dcc.ufmg.br
Universidade Federal de Minas Gerais
Belo Horizonte, MG, Brasil

Resumo

Context: In today's technological landscape, the development of Web information systems, especially in the back-end, is an essential and constantly evolving activity. The choice of technology for this layer is crucial to ensuring the performance, security and efficiency of the system as a whole. **Problem**: To date, recent literature lacks studies that address the identification of specific requirements and characteristics to guide the selection of a back-end technology for Web systems. **Solution**: This paper carries out a comparative analysis of the most widely used back-end technologies according to the most recent survey by Stack Overflow: Node.js, Django REST Framework (DRF) and ASP.NET Core. **IS Theory**: This study was developed under the guidance of Performance Evaluation Frameworks, aimed at guiding the choice of back-end technologies in the architecture of Web information systems. **Method**: To conduct this study, an API server was implemented in each of the technologies to carry out tests and collect data and metrics, such as performance in different scenarios. **Summarization of results**: The results highlighted that .NET had a high consumption of resources, while Node.js and DRF showed low consumption and similar performance. In terms of response times, .NET was most effective, followed by Node.js and DRF. **Contributions and impact on IS**: The results of this study benefit developers, companies and researchers by providing information for choosing back-end technologies, as well as serving as a reference point for researchers working in the field.

CCS Concepts: • **Do Not Use This Code → Generate the Correct Terms for Your Paper**; *Generate the Correct Terms for Your Paper*; Generate the Correct Terms for Your Paper; Generate the Correct Terms for Your Paper.

SBSI 2024, May, 2024, Juiz de Fora, Brazil

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Keywords: Back-end, Node.js, Django REST Framework, ASP.NET Core

ACM Reference Format:
Klayver Ximenes Carmo, Fischer Jonatas Ferreira, and Eduardo Figueiredo. 2024. Performance Evaluation of Back-End Frameworks: A Comparative Study. In *Proceedings of (SBSI 2024).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introdução

O desenvolvimento de sistemas de informação para Web teve seu início na década de 90 com a criação dos primeiros sites, compostos principalmente por páginas estáticas [8, 17]. Com o passar do tempo, as tecnologias evoluíram permitindo um desenvolvimento de sistemas Web dinâmicos e interativos, até o surgimento dos termos *front-end* e *back-end*. O *back-end* surgiu como uma nova camada do desenvolvimento de sistemas de informações para Web para garantir um bom funcionamento de um site ou aplicação com gerenciamento de conteúdo e infraestrutura relacionadas ao servidor. Com isso, é possível agora gerenciar servidores, bancos de dados, processar e armazenar dados, bem como outros recursos necessários para a execução adequada de um sistemas de informação para Web.

A crescente e contínua demanda por desenvolvedores tem destacado a real importância desta função no cenário de desenvolvimento atual. Simultaneamente, o cenário em constante mudança da tecnologia trouxe muitas opções para atender a necessidade de sistemas Web escaláveis, eficientes e seguros. Cada uma dessas tecnologias, fornecendo recursos únicos, têm o propósito de tornar o desenvolvimento *back-end* mais robusto. A escolha do *framework* adequada para um projeto específico busca estar de acordo com as tendências do mercado, seguindo uma abordagem estratégica e planejada.

Em 2022, o *Stack Overflow* conduziu uma pesquisa com mais de 70 mil desenvolvedores com o objetivo de determinar as tecnologias mais desejadas, utilizadas e populares, bem como outras informações relevantes [16]. De acordo com os resultados, pelo décimo ano consecutivo, *JavaScript* foi a linguagem de programação mais utilizada pelos desenvolvedores, juntamente com *Python* e *C#* que estão entre as 10 primeiras posições. Os resultados da pesquisa também mostraram que os *frameworks* `Django` (Python), `ASP.NET Core` (C#) e `Node.js` (JavaScripst) estão entre os 10 mais utilizados, sendo este último o *framework back-end* mais utilizado pelos profissionais.

Diante da grande quantidade de tecnologias existentes, este trabalho tem como objetivo fazer uma comparação, analisando características e elencando vantagens e desvantagens dos *frameworks* de *back-end* mais utilizados segundo a pesquisa do *Stack Overflow*. Os *frameworks* escolhidos foram `Node.js`, `Django REST Framework` (DRF) e `ASP.NET Core`. Este estudo visa oferecer uma fundamentação prática para auxiliar empresas e desenvolvedores de software na escolha dos *frameworks* de *back-end*, visando aprimorar a qualidade de seus sistemas de informação. O objetivo é fornecer recursos que permitam tomar decisões embasadas na escolha da tecnologia de *back-end* mais apropriada, fornecendo uma compreensão das vantagens e desafios de cada *framework*.

Ainda, esse estudo apresenta uma comparação prática que explora aspectos como análise de desempenho e performance, realizando testes e medições em cenários específicos. Estes testes permitiram a compreensão de como os *frameworks* se comportam com respeito ao consumo de recursos e tempo de resposta em diferentes cenários.

Os resultados deste estudo revelaram padrões distintos no consumo de recursos pelos diferentes *frameworks* analisados. Notoriamente, o consumo de recursos pelo .NET mostrou-se significativamente elevado, contrastando com o baixo consumo e desempenho semelhante observados tanto no Node.js quanto no DRF. No que diz respeito aos tempos de resposta, ficou aparente que o .NET apresentou o desempenho mais eficaz, seguido de perto pelo Node.js e pelo DRF.

Em resumo, os resultados deste estudo oferecem benefícios práticos para desenvolvedores, empresas e pesquisadores interessados em *frameworks back-end* de sistemas Web. Ao destacar as vantagens e desvantagens específicas de cada *framework*, o estudo possibilita que os desenvolvedores utilizem esse conhecimento para a escolha de *frameworks* de *back-end* para a implementação de seus sistemas de informação. Além disso, serve como uma fonte para pesquisadores, contribuindo para o avanço do desenvolvimento de módulos *back-end* em sistemas de informação.

Os dados dos cenários de testes utilizados neste estudo estão disponíveis em uma tabela de artefatos pública [13]. Nela estão presentes os dados obtidos em todas as execuções e as médias dos mesmos. Além disso, os códigos utilizados com as implementações das APIs (Interface de Programação de Aplicação) nos *frameworks* em estudo estão disponíveis publicamente em um repositório no GitHub [12].

Todos os dados dos cenários de testes utilizados neste estudo (tabelas e gráficos), juntamente com os códigos utilizados nas implementações das APIs nos *frameworks* em análise [12].

Este trabalho está organizado da seguinte forma: Seção 2 apresenta os fundamentos teóricos deste estudo, discutindo conceitos e tópicos relevantes no desenvolvimento do *back-end* de sistemas. A Seção 3 apresenta os objetivos, questões de pesquisa e configurações do estudo, descrevendo a metodologia empregada na condução da pesquisa. Os resultados obtidos por meio dos testes são expostos na Seção 4, seguidos por discussões na Seção 5. A apresentação de trabalhos relacionados a este estudo é realizada na Seção 6. Na Seção 7, são discutidas as ameaças e as estratégias adotadas para contornar os problemas que poderiam comprometer a validade do estudo. Por fim, na Seção 8, são apresentadas as considerações finais.

2 Frameworks Back-end

O `Node.js`[1] é uma plataforma para desenvolvimento de *software back-end* criado em 2009 por Ryan Dahl. Sua criação teve como intuito ser uma alternativa para as tecnologias da época, já que as existentes até então, possuíam deficiência em aplicações Web em tempo real. Além disso, o `Node.js` é baseado em *JavaScript*, sendo construído sobre a *engine JavaScript* criado pelo Google para o Chrome, chamado de V8[2]. Como um ambiente de execução *JavaScript* assíncrono orientado a eventos, o `Node.js` é projetado para desenvolvimento de aplicações escaláveis de rede [11]. Sua versatilidade permite que seja utilizado em vários contextos de desenvolvimento, como em aplicações em tempo real, na construção de jogos, plataformas de *streaming* e ambientes escaláveis, como serviços de *back-end* e servidores.

O `Django`[3] é um *framework* Web *Python* criado em 2003 por Adrian Holovaty e Simon Willison que incentiva o desenvolvimento rápido e um *design* limpo e pragmático, com foco em escalabilidade rápida e flexível, velocidade de desenvolvimento e segurança. O `Django REST Framework` (DRF) surgiu como uma extensão do `Django`, projetado especificamente para o desenvolvimento de APIs Web. O mesmo herda a simplicidade e flexibilidade do `Django` enquanto adiciona funcionalidades essenciais para a construção de APIs. Características marcantes do `DRF` incluem serialização que suporta ORM (mapeamento objeto-relacional) e tipos de dados não-ORM, autenticação personalizável e políticas de permissões, além da navegação API com interface. O *framework* também se destaca pela sua capacidade em lidar com requisições simultâneas eficientemente, graças à sua construção assíncrona [2].

O ecossistema `.NET` da Microsoft, lançado em 2002 com o `.NET Framework` para aplicações Windows, expandiu-se com o `.NET Core` de 2014, uma versão multiplataforma e *open-source*. O ambiente do `.NET` suporta Linux, MacOS e Windows em várias arquiteturas como Arm64 e x64. Programas em C# executam no CLR (*Common Language Runtime*) do `.NET`, um *runtime* que oferece coleta de lixo e segurança de memória. Compatível com múltiplas linguagens da Microsoft (C#, F# e Visual Basic), ele permite execução sem recompilação através da instalação do *runtime* apropriado [9]. A compilação no `.NET` usa IL (linguagem intermediária) para

[1] https://nodejs.org/
[2] https://v8.dev/
[3] https://www.djangoproject.com/

compatibilidade entre SOs e arquiteturas. Suporta JIT (*Just-In-Time*) para otimização específica do ambiente ou AOT (*Ahead-Of-Time*) para inicialização rápida mas requerendo compilações distintas por plataforma [9].

3 Configurações do estudo

Esta seção descreve o procedimento experimental adotado deste estudo. Inicialmente, na Seção 3.1, são apresentados os objetivos e as questões de pesquisa. Na Seção 3.2 são detalhadas as métricas comparativas. Posteriormente, na Seção 3.3, são descritos os detalhes da implementação das APIs em cada um dos *frameworks* selecionados. A Seção 3.4 aborda a modelagem e implementação dos testes utilizados para a comparação entre os *frameworks*.

3.1 Objetivo e questões de pesquisa

A definição do objetivo foi feita com base no modelo objetivo-questão-métrica GQM [1]. **Comparar** *frameworks back-end* de maneira prática; **com a finalidade** de analisar sua performance e desempenho; **no que diz respeito** a auxiliar profissionais e pesquisadores na escolha do *framework* para *back-end* **sob a perspectiva** de desenvolvedores e pesquisadores na área de desenvolvimento *back-end* no contexto de construção e estudo de sistemas de informação para Web.

Considerando estudos anteriores [3, 6, 10], foram modelados três tipos de cenários para a coleta das métricas de performance e rendimento de cada API, são eles:

- Teste de pico: a aplicação será submetida à uma grande quantidade de requisições durante um curto intervalo de tempo e será observado o comportamento da mesma com relação ao tempo de resposta e consumo de recurso.
- Teste de carga crescente: a aplicação será submetida à uma carga crescente de requisições e monitorada para analisar o comportamento sob esta situação.
- Teste de resistência: como uma variação de *soak test*[4], neste teste será feito um cenário com requisições a uma carga constante durante um longo período de tempo, buscando identificar o comportamento geral do sistema.

Com base em pesquisas bibliográficas, até o momento, não foram encontrados estudos comparativos focados à avaliação prática dos três *frameworks* alvo. Motivado por isto, foram formuladas as seguintes questões de pesquisa:

Cenário 1 - Teste de pico

QP₁: *Qual é o comportamento dos frameworks com relação ao consumo de recursos no cenário de teste de pico?*

QP₂: *Qual é o comportamento dos frameworks com relação ao tempo de resposta no cenário de teste de pico?*

Cenário 2 - Teste de carga crescente

QP₃: *Qual é o comportamento dos frameworks com relação ao consumo de recursos no cenário de teste de carga crescente?*

QP₄: *Qual é o comportamento dos frameworks com relação ao tempo de resposta no cenário de teste de carga crescente?*

Cenário 3 - Teste de resistência

QP₅: *Qual é o comportamento dos frameworks com relação ao consumo de recursos no cenário de teste de resistência?*

QP₆: *Qual é o comportamento dos frameworks com relação ao tempo de resposta no cenário de teste de resistência?*

3.2 Definição das métricas comparativas

Após o desenvolvimento das APIs em cada uma dos *frameworks* em estudo, `Node.js`, `Django` e `.NET`, uma etapa fundamental nesta pesquisa envolveu a condução de testes práticos. O objetivo principal foi avaliar o desempenho e a eficiência de cada tecnologia em diferentes contextos. Para isso, foram implementados testes de carga nos servidores desenvolvidos. Esses testes simulam situações do mundo real, submetendo as APIs a diferentes níveis de tráfego e demanda, permitindo avaliar como cada tecnologia se comporta em determinados cenário.

Nos testes de carga, destacam-se duas categorias de métricas essenciais que podem ser monitoradas e coletadas: métricas de rendimento e métricas de desempenho. Essas métricas são fundamentais para entender como os *frameworks* lidam quando são submetidas a diferentes cenários de cargas e fornecer informações sobre sua capacidade de resposta, eficiência e escalabilidade em ambientes variados [6].

Para este trabalho, com base no estudo apresentado anteriormente, foram escolhidas as seguintes métricas:

- Taxa de erros de requisição: Como uma métrica de rendimento, é essencial para analisar a proporção de solicitações que resultam em erros em relação ao número total de solicitações feitas durante os testes, incluindo falhas de requisição ou erros internos do servidor.
- Tempo de requisição: Dentro do contexto de performance, esta métrica é importante para medir o tempo que uma API leva para responder ou completar uma solicitação de um cliente. Um menor tempo de resposta geralmente indica um melhor desempenho da API.
- Consumo de RAM: Dentro do contexto de gerenciamento de recursos, a avaliação do consumo de memória permite entender o quanto de memória a tecnologia consome durante a execução das solicitações. Isso é essencial para dimensionar e otimizar o uso de recursos do servidor.
- Consumo de CPU: Similar ao consumo de memória, a análise do consumo de CPU também é importante para o gerenciamento de recursos de uma aplicação. A alta utilização da CPU pode levar a custos operacionais elevados e, em última instância, impactar o desempenho e escalabilidade.

[4] Teste utilizado para identificar problemas de um sistema quando exposto a uma carga constante por um grande período de tempo.

3.3 Modelagem e implementação das APIs

Foram desenvolvidas três APIs, uma para cada *framework*. Cada API foi implementada de forma independente, utilizando a mesma tecnologia de banco de dados (`MySQL`), mas com estruturas diferentes devido às peculiaridades de cada *framework*. Os CRUDs implementados foram feitos seguindo os padrões de solicitação de uma API REST [4]. Cada método HTTP foi responsável por fazer uma operação no sistema, sendo eles:

- **GET**: Buscar informações dos dados disponíveis na base de dados.
- **POST**: Inserir um dado na base de dados.
- **PUT**: Alterar informações de um dado já existente.
- **DELETE**: Remover um dado da base de dados.

3.4 Modelagem e implementação dos testes

Após o desenvolvimento das APIs utilizando os *frameworks* em estudo, foram realizados testes de carga para a obtenção das métricas detalhadas na Seção 3.2. O processo de testes envolve alguns passos importantes para a melhor otimização dos processos e validação dos resultados. A configuração dos mesmos foram feitos com base em estudos anteriores [3, 6].

Para que os servidores desenvolvidos localmente fossem monitorados e comparados em sua melhor capacidade, foram feitos alguns procedimentos específicos. A quantidade de conexões do banco de dados foi alterada para que não interferisse nos resultados. O banco de dados utilizados nas APIs foi o `MySQL`, que tem capacidade máxima padrão de 151 conexões. Esta capacidade foi aumentada para o valor máximo de 10.000 conexões[5]. A cada cenário de teste os servidores foram reiniciados para não ter influências de resultados passados ou requisições pendentes. A máquina utilizada contém as seguintes especificações:

- Processador: Ryzen 5 5600x;
- Memória RAM: 16GB, 3200MHz;
- Placa de vídeo: GTX 1660;

3.5 Obtenção dos dados comparativos

A modelagem e execução dos testes foram feitos com a ferramenta de testes `JMeter`[6]. O `JMeter` é uma ferramenta de código aberto em Java utilizada para testes estáticos e dinâmicos em sistemas Web, servidores e outros tipos de sistema. Ele oferece recursos para modelagem, execução, geração de relatórios e análise de testes. Sua instalação é de fácil entendimento, podendo ser feita pelo site oficial[7].

Os testes feitos no `JMeter` consistem em requisições que simulam usuários, chamados de VUs, usuários virtuais. Estes VUs simulam o comportamento de um usuário real utilizando a aplicação a ser testada, representados por *threads*. Em todos os cenários, o fluxo de requisições escolhido foi:

1. Criar um componente no banco de dados;
2. Atualizar os dados do componente criado no passo anterior;
3. Fazer a leitura do componente criado;
4. Apagar o componente criado no Passo 1;

A modelagem dos testes foram feitas utilizando o *plugin Stepping Thread Group* e *Ultimate Thread Group*[8]. Com ele é possível definir vários parâmetros de configuração para testes de carga. Os principais parâmetros utilizados foram a quantidade de grupos de *threads*, atraso de iniciação de cada grupo e o tempo de duração dos mesmos. A seguir são apresentados as descrições de como foram feitas as modelagem dos testes com relação à quantidade de cargas utilizadas durante a execução de cada cenário.

- **Teste de pico**: Neste teste, o servidor foi inicializado com uma carga de 25 a 50 *threads* e repentinamente elevada para um pico de 500 *threads* durante 30 segundos. Após isto a quantidade de *threads* decaiu para os mesmos valores do início.
- **Teste de carga crescente**: Neste cenário o servidor iniciou com 50 usuários tendo um aumento de 25 usuários gradualmente a cada 20 segundos, chegando ao limite de 200 *threads*.
- **Teste de resistência**: Diferente dos outros dois cenários, no teste de resistência não tem variação de carga ao longo do tempo. Os servidores foram submetidos à 50 *threads* durante um período de 3 horas.

Na execução de cada teste, o `JMeter` faz a leitura e monitoramento dos dados e recursos. Ao final da execução é gerado um arquivo com os dados do teste executado. Com estes dados, foram feitos relatórios onde foi possível fazer a verificação do tempo de execução, sucesso das requisições, quantidade de dados trafegados e outros. Os testes foram feitos no sistema operacional Linux, auxiliando no monitoramento do consumo de recursos das APIs. Primeiramente foi utilizado o comando "lsof"[9]. Com ele é possível identificar o processo de execução do servidor através da porta aberta na execução da API.

Após a identificação do PID do processo, é feito o monitoramento do mesmo. Para isto, foi utilizado o comando "top"[10]. Com este comando, foi possível gerar um arquivo de logs apresentando os dados de consumo de recursos em relação ao tempo.

Para otimização dos processos de testes, foram utilizados recursos como variáveis de ambientes do `JMeter` para facilitar a transição de cenários de testes e os *frameworks* a serem testados. Além disso, também foi feito um *script* para a execução do arquivo de teste do `JMeter` para execução via linha de comando, assim como para os comandos nativos do

[5] https://dev.mysql.com/doc/refman/8.0/en/too-many-connections.html
[6] https://jmeter.apache.org/
[7] https://jmeter.apache.org/download_jmeter.cgi
[8] http://jmeterplugins.com/wiki/Start/index.html
[9] https://www.geeksforgeeks.org/lsof-command-in-linux-with-examples/
[10] https://www.geeksforgeeks.org/top-command-in-linux-with-examples/

Linux para identificação do PID dos processos e geração de arquivos de logs.

4 Resultados

Os resultados da análise das métricas práticas dos *frameworks* `Node.js`, `Django` e `.NET` são apresentados nos tópicos a seguir. Tais informações foram obtidas por meio de testes realizados nos servidores construídos em cada uma dos *frameworks*, conforme descrito na Seção 3. A discussão das comparações abordam os atributos citados na Seção 3.2. A Seção 4.1 apresenta os resultados obtidos no teste de pico, enquanto a Seção 4.2 mostra os dados resultantes do teste de carga crescente. A Seção 4.3 demonstra os resultados do teste de resistência. Por fim, a Seção 4.4 descreve algumas outras observações notadas durante a realização dos testes.

A magnitude do consumo de CPU é avaliada em relação ao total de núcleos do processador, assim como para os valores de consumo de RAM. Os gráficos foram ajustados de acordo com os valores de pico obtidos em cada medição.

4.1 Teste de pico

QP$_1$: *"Qual é o comportamento dos frameworks com relação ao consumo de recursos no cenário de teste de pico?"*. A Figura 1 apresenta os resultados médios do consumo de CPU e memória RAM em porcentagem dos *frameworks* durante os testes de pico, respectivamente. Os *frameworks* `Node.js` e `Django` tiveram os consumos relativamente próximos e baixos, comparados ao `.NET`, que, por sua vez, apresentou um consumo mais elevado.

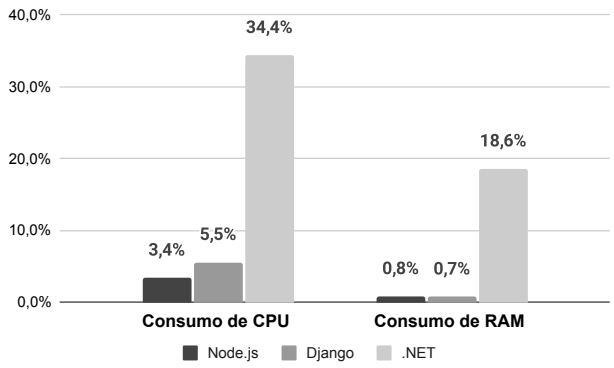


Figura 1. Consumo recursos - teste de pico

QP$_1$. No cenário de teste de pico, o `.NET` demonstrou os consumos mais elevados de recursos, não se mostrando como a opção mais eficiente nesta característica para este cenário. O `Node.js` e o `Django` apresentaram resultados próximos, destacando-se como escolhas mais favoráveis.

QP$_2$. *"Qual é o comportamento dos frameworks com relação ao tempo de resposta no cenário de teste de pico?"*. A Figura 2 apresenta os resultados médios do tempo de resposta agrupados por métodos HTTP dos *frameworks* durante os testes de pico. Como dado discrepante, foi registrado o método POST do `Django` neste cenário. Os demais métodos não tiveram o mesmo comportamento, tendo uma maior proximidade dos tempos. Dentre eles se destacam os tempos do `.NET`, sendo inferior a todos neste caso de teste.

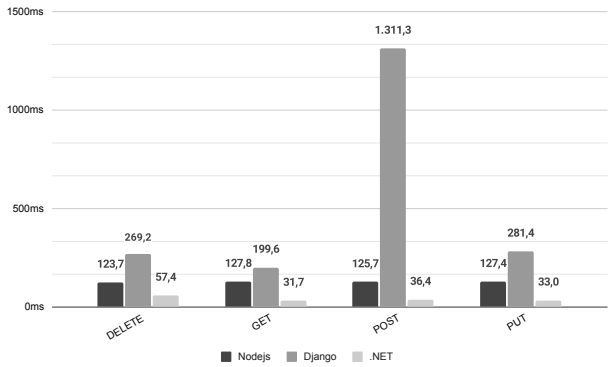


Figura 2. Tempo de resposta médio dos métodos HTTP - teste de pico

QP$_2$. No cenário de teste de pico, o `.NET` sobressaiu ao apresentar os melhores resultados em tempo médio de resposta para todos os métodos, seguido pelo `Node.js` e `Django`.

4.2 Teste de carga crescente

QP$_3$: *"Qual é o comportamento dos frameworks com relação ao consumo de recursos no cenário de teste de carga crescente?"*. A Figura 3 apresenta os resultados médios do consumo de CPU e memória RAM em porcentagem dos *frameworks* durante os testes de carga crescente, respectivamente. Os *frameworks* `Node.js` e `Django` tiveram o consumo relativamente próximos e baixo, comparados ao `.NET`, sendo o `Node.js` com os menores valores. O `.NET`, por sua vez, teve um consumo mais elevado.

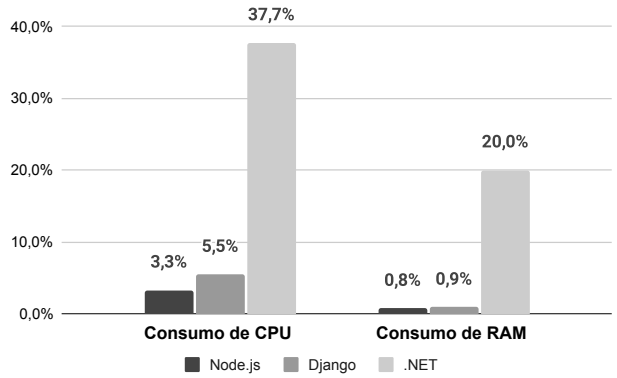


Figura 3. Consumo de recursos - teste de carga crescente

QP$_3$. No cenário de teste de carga crescente, o `.NET` demonstrou os maiores valores no quesito de consumos de recursos, não se mostrando como a opção mais eficiente nesta característica para este cenário. O `Node.js` e o `Django` apresentaram resultados próximos em relação ao consumo de CPU e RAM, destacando-se como escolhas mais favoráveis.

QP$_4$: *"Qual é o comportamento dos frameworks com relação ao tempo de resposta no cenário de teste de carga crescente?"*. A Figura 4 apresenta os resultados médios do tempo de resposta agrupados por métodos HTTP dos *frameworks* durante os testes de carga crescente. Como dado discrepante, foi registrado o método POST do `Django` neste cenário, tendo aproximadamente o dobro do tempo comparado com seus outros métodos (DELETE, GET e PUT). Os demais métodos não tiveram o mesmo comportamento, tendo uma maior proximidade nos tempos. Dentre eles, se destacam os tempos do `Node.js` e `.NET` com pouca variação de tempo entre os métodos, sendo o `.NET` com menores tempos neste caso de teste.

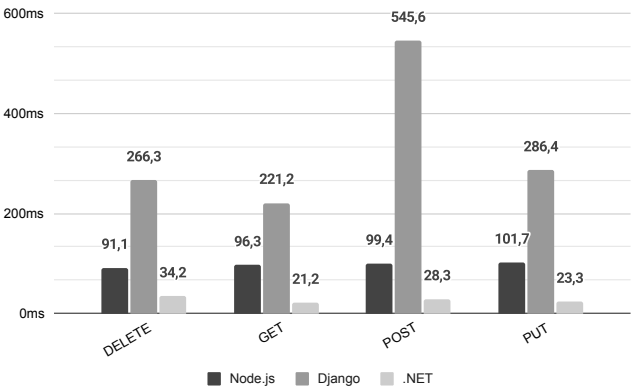


Figura 4. Tempo de resposta médio dos métodos HTTP - teste de carga crescente

QP$_4$. Na métrica de tempo de resposta, no cenário de teste de carga crescente, o `.NET` apresentou os melhores resultados em todos os métodos, seguido pelo `Node.js` e `Django`, que apresentaram os maiores tempos.

4.3 Teste de resistência

QP$_5$: *"Qual é o comportamento dos frameworks com relação ao consumo de recursos no cenário de teste de resistência?"*. A Figura 5 apresenta os resultados médios do consumo de CPU e memória RAM em porcentagem dos *frameworks* durante os testes de resistência, respectivamente. Neste cenário, os *frameworks* `Node.js` e `Django` tiveram baixos consumos de recursos quando comparados ao `.NET`. O `Node.js` obteve o menor consumo de CPU, enquanto o `Django` e o `Node.js` apresentaram valores iguais no consumo de memória, ambos demonstrando eficiência semelhante.

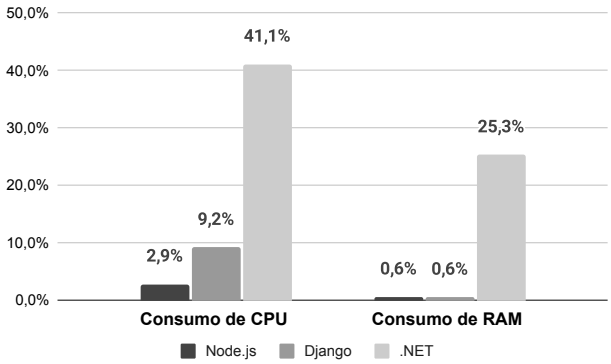


Figura 5. Consumo de recursos - teste de resistência

QP$_5$. No cenário de teste de resistência, o `.NET` registrou os maiores valores para a métrica de consumo de recursos. O `Node.js` apresentou um consumo de CPU menor, com uma diferença de aproximadamente 6% em relação ao `Django`. No consumo de RAM, o `Node.js` e o `Django` tiveram resultados semelhantes.

QP$_6$: *"Qual é o comportamento dos frameworks com relação ao tempo de resposta no cenário de teste de resistência?"*. A Figura 6 apresenta os resultados médios do tempo de resposta agrupados por métodos HTTP dos *frameworks* durante os testes de resistência. Foi observado que todos os tempos ficaram abaixo dos 100ms, exceto o método PUT do `Django`, ficando com média de 108,80ms. O `Node.js` e o `.NET` obtiveram os melhores tempos no geral, destacando o `.NET` que teve o melhor tempo no método POST com 12,30ms. O

Django, por sua vez, teve melhor resultado no método POST do que o Node.js.

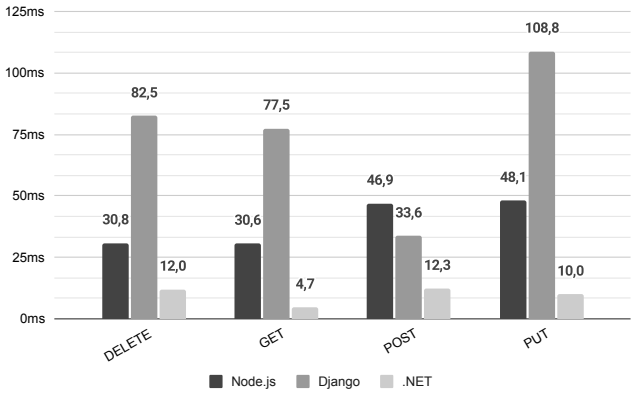


Figura 6. Tempo de resposta médio dos métodos HTTP - teste de resistência

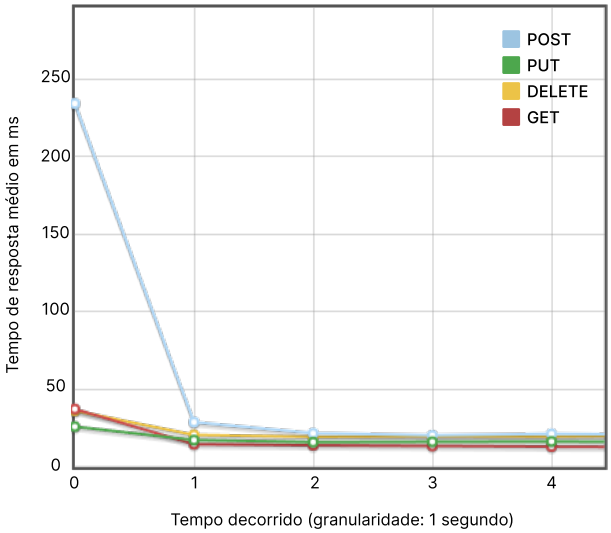


Figura 7. Inicialização do .NET no teste de carga crescente

QP$_6$. Na métrica de tempo de resposta, no cenário de teste de resistência, o .NET mostrou os melhores resultados em todos os métodos em comparação com os outros *frameworks*. O Node.js se destacou-se como melhor opção nos métodos DELETE, GET e PUT em comparação com o Django. No método POST, o Django apresentou uma pequena vantagem em relação ao Node.js.

4.4 Outras observações

Durante os testes práticos, identificou-se em alguns cenários que o .NET, apesar de apresentar os melhores tempos de resposta, exibiu um tempo de inicialização mais longo em comparação com os outros *frameworks*. A Figura 7, apresentando os 4 primeiros segundo da primeira execução do cenário de teste de carga crescente, ilustra um dos casos nos quais essa particularidade foi observada. Nela, é observado que a primeira requisição do conjunto, requisição POST, chega a ter um tempo de resposta acima de 200ms. As marcações no tempo decorrido 0 representam os tempos de resposta das primeiras requisições do teste.

Foi observado também, durante os testes práticos, que nenhum dos *frameworks* demonstrou falhas relevantes nas requisições. Em todos os cenários avaliados, a taxa de erros de requisição foi menor que 0,5%. Com estes resultados concluise que, para os cenários modelados, os *frameworks* possuem boa confiabilidade.

5 Discussões

Conforme destacado na Seção 3.4, os testes conduzidos foram modelados e executados por meio da ferramenta JMeter, adotando uma abordagem padrão na modelagem dos cenários, onde simulações de usuários virtuais foram executadas de forma simultânea. Esse cenário é conhecido como "requisições simultâneas", caracterizado pela sincronização de múltiplas interações com o sistema. No contexto dos métodos empregados nos testes, mencionados na Seção 3.3, vale ressaltar que tais operações não demandaram um significativo poder computacional, uma vez que foram projetadas para representar simulações realistas de requisições simultâneas, sem complexidades computacionais internas significativas.

Os testes realizados com os *frameworks* revelaram uma performance notável do Node.js com relação ao consumo de recursos em situações de entrada e saída (I/O). A abordagem assíncrona do Node.js, com seu modelo *single-thread* e a eficiente utilização do Event Loop, permitiu uma manipulação ágil e eficaz de requisições simultâneas. A capacidade de continuar a execução enquanto aguarda operações de I/O foi particularmente benéfica, resultando em uma eficiência destacada, especialmente em cenários onde a concorrência de requisições é uma consideração crítica. A arquitetura do Node.js mostrou-se ideal para ambientes onde a escalabilidade horizontal é crucial, oferecendo uma resposta eficiente a picos de carga e requisitos de operações concorrentes.

Durante os testes, observou-se que o *framework* .NET ofereceu os melhores tempos de resposta. No entanto, essa melhoria veio acompanhada de um maior consumo de recursos do sistema. O .NET não se provou eficiente com relação ao consumo de recursos no processamento de tarefas nos cenários de testes modelados. Diferente dos dados obtidos do tempo de resposta, onde se mostrou ser a melhor opção. Contudo, suas características *multithread* e o tempo necessário para a compilação Just-In-Time (JIT) podem afetar seu desempenho em determinadas situações, como as observadas nos testes. Especificamente na inicialização do servidor, o

.NET não se mostrou tão ágil, com tempos de resposta superiores aos demais. Portanto, pode não ser a opção ideal para projetos que priorizem tempos rápidos de *cold start*.

O Django se destacou como uma alternativa notável ao Node.js no contexto de consumo de recursos, exibindo valores comparáveis, especialmente no que diz respeito ao uso eficiente da RAM. No entanto, é crucial observar que, apesar dessa semelhança em consumo de recursos, o Django se posicionou como a opção menos favorável quando a prioridade recai sobre o tempo de resposta. O Django, embora ofereça uma estrutura robusta para o desenvolvimento de APIs RESTful e demonstre eficácia no consumo de recursos, pode apresentar desafios em situações onde a resposta rápida é uma prioridade crucial.

Essas observações ressaltam a importância de considerar não apenas métricas isoladas, como o consumo de recursos, que oferecem apenas uma visão parcial do desempenho. Portanto, é crucial analisar também o desempenho geral em cenários específicos de aplicação, levando em conta as particularidades de cada contexto. O equilíbrio entre consumo de recursos e tempo de resposta é essencial para determinar a escolha mais adequada.

Com relação aos métodos HTTP, foi possível notar uma menor diferença nos valores entre os *frameworks* do tempo de resposta no teste de pico quando comparado com o teste de carga crescente (ver Figuras 8a e 8b). No teste de resistência, foi identificado o menor coeficiente de variação dos tempos de resposta entre os *frameworks*, sendo 372,6% no teste de pico, 405,6% no teste de carga crescente e 331,5% no teste de resistência (ver Tabela 1). Isto indica que, para os 3 cenários, o teste de resistência foi o que apresentou a maior proximidade entre os tempos de resposta dos métodos entre os *frameworks* (ver Figura 8c).

	T1	T2	T3
DELETE	72,2%	92,7%	87,4%
GET	70,4%	89,5%	98,1%
POST	144,9%	124,9%	56,4%
PUT	85,1%	98,5%	89,6%
Soma Total	372,6%	405,6%	331,5%

Tabela 1. Coeficiente de variação do tempo médio de resposta dos métodos entre os *frameworks*.

T1 - teste de pico; T2 - teste de carga crescente; T3 - teste de resistência

A escolha de utilizar a média de três execuções para cada cenário de teste foi fundamentada na consistência dos resultados, evidenciada pelo coeficiente de variação que se manteve abaixo de 10% em todos os casos, conforme apresentado na Tabela 2.

	T1	T2	T3
Node.js	4,2%	7,7%	0,3%
Django	0,6%	6,7%	1,5%
.NET	7,5%	0,3%	9,6%

Tabela 2. Coeficiente de variação do tempo médio de resposta entre as execuções.

Ao analisar os resultados nos três cenários considerados, foi possível identificar distintas características entre os *frameworks* avaliados. O Django se destaca por apresentar os maiores tempos de resposta em média, indicando uma performance relativa inferior em comparação com .NET e Node.js. Por sua vez, o .NET demonstra ser mais intenso em termos de consumo de recursos, tanto de CPU quanto de RAM, em comparação com os outros *frameworks*. Em contrapartida, o Node.js revela-se como uma opção mais eficiente em termos de consumo de recursos, com valores semelhantes ao Django no uso de RAM e superando ambos, especialmente o .NET, no quesito tempo de resposta. Além disso, o Node.js exibe o menor desvio padrão entre as características, com uma consistência superior em seu desempenho nos diferentes cenários (ver Figura 9).

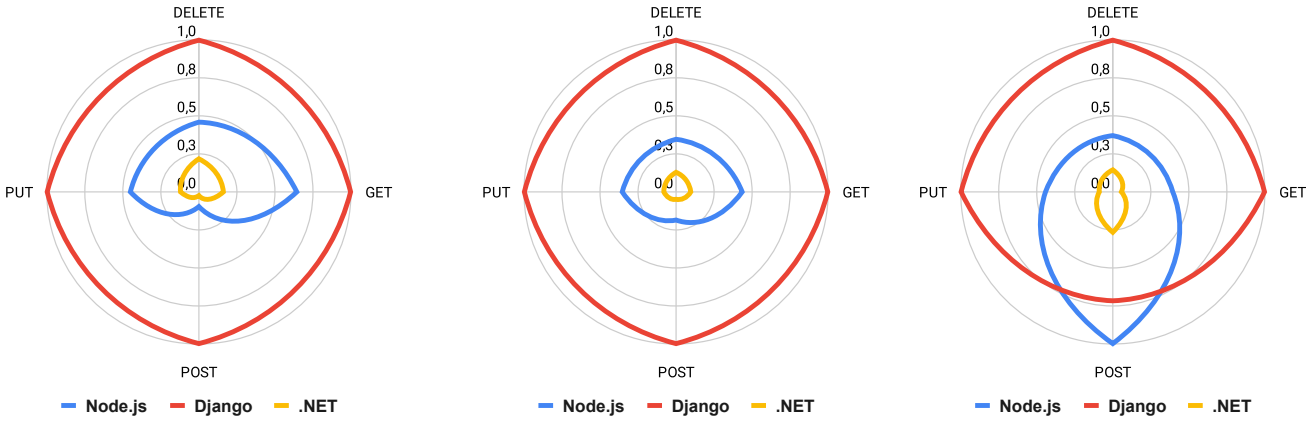
6 Trabalhos relacionados

Existem alguns estudos avaliando o desempenho de *frameworks* para sistemas de informação Web.

No estudo de Sharma et al. [15], foram conduzidos diversos testes de carga com o objetivo de avaliar o desempenho de diferentes *frameworks* Web, destacando Node.js e Django. A ênfase foi sobre a capacidade de suporte a usuários simultâneos, e os resultados indicaram que, em resumo, o Node.js apresentou um desempenho superior em termos de tempo médio de requisições e quantidade total de solicitações, quando comparado ao Django. A conclusão principal do estudo sugere que o Django pode ser mais vantajoso em situações que demandam um nível de simultaneidade mais elevado, enquanto o Node.js se destaca em eficiência e desempenho em cenários com um grande número de requisições simultâneas.

No estudo de K.Lei et. al. [7], os autores realizaram testes de *benchmark* e cenários de testes. Foi utilizado o Apachebean para os testes de *benchmark* e o Loadrunner para simular cenários de carga, os desenvolvedores identificaram claramente o Node.js como uma escolha mais leve e eficiente em comparação com PHP e Python. Com base nos resultados desses testes, destacou-se a vantagem do Node.js, evidenciando sua notável eficiência, especialmente em ambientes que apresentam uma significativa demanda por operações de entrada e saída (E/S).

Outros estudos também foram realizados sobre testes de carga para a avaliação de performance e desempenho de sistemas. D. Widodo, P. Kristalina, M. Z. S. Hadi e A. D. Kurniawati et. al. [19] avaliaram o desempenho de um sistema de aplicação Web utilizando JMeter. Foram realizados testes



(a) Teste de pico **(b)** Teste de carga crescente **(c)** Teste de resistência

Figura 8. Tempos de resposta dos cenários de teste

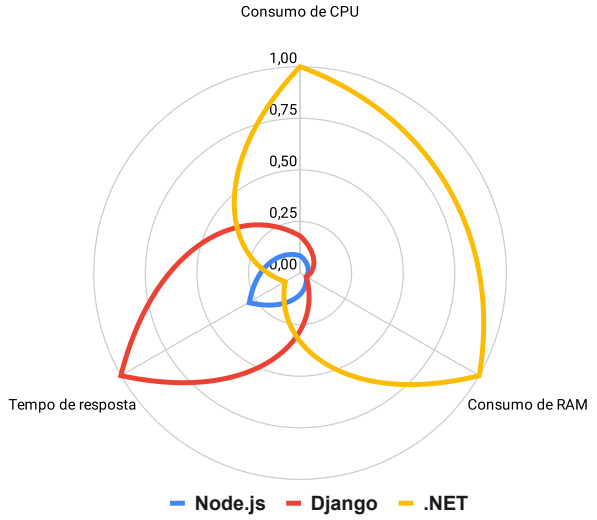


Figura 9. Médias totais

de estresse, resistência e pico para monitorar o uso de CPU e memória e transferência de dados.

Wang e Wu et. al. [18] conduziram uma pesquisa sobre a tecnologia de teste de desempenho automatizado baseada no JMeter. Foram citados conceitos e classificações dos testes de performance, como testes de carga e estresse. Além de indicadores utilizados como métricas, como tempo de resposta e consumo de recursos.

Outro estudo conduzido por Jha et. al. [5] avaliou contêineres Docker para interferência em microserviços. Eles empregaram o JMeter para a configuração dos testes. Foram modelados testes de estresse para coletar informações sobre o desempenho e a resposta da aplicação em tempo real.

Por último, Pradeep e Sharma et. al. [14] conduziram uma avaliação de tecnologias de teste de estresse e desempenho para aplicações baseadas na Web. Foram comparadas diversas ferramentas, incluindo o JMeter, e avaliaram sua eficácia na detecção de problemas de desempenho.

7 Ameaças à validade

Os procedimentos experimentais foram conduzidos conforme os passos descritos na Seção 3. No entanto, existem algumas ameaças à validade que podem comprometer a validade dos resultados. Nos tópicos a seguir são discutidos cada um dos quatro tipos de ameaças, com respectivos tratamentos, listados por Wolin [20]: validade de construção, interna, conclusão e externa.

A validade de construção deste estudo foi considerada durante a modelagem dos testes, que foram projetados para refletir o cenário real de utilização com métodos REST. Primeiramente, os testes foram realizados no sistema operacional Linux, esta escolha permitiu uma avaliação mais precisa do consumo de recursos (CPU e RAM) pelos *frameworks*. O Linux é amplamente utilizado em ambientes de desenvolvimento e produção, tornando essa escolha alinhada com cenários do mundo real.

Além disso, a ferramenta JMeter foi empregada para a execução dos testes e coleta de dados de desempenho. O JMeter fornece uma abordagem abrangente para a avaliação de APIs, possibilitando a medição do tempo de resposta e a taxa de erros de requisição. Sua utilização contribuiu para a precisão e confiabilidade dos dados obtidos.

No escopo de validação interna do estudo, é importante adotar medidas para garantir a consistência e a confiabilidade dos resultados, visto que são *frameworks* diferentes. Para isto, as APIs desenvolvidas para cada *framework* foram implementadas com as mesmas funcionalidades a serem testadas, assegurando que o escopo dos testes permanecesse uniforme.

Além disso, para garantir a uniformidade no ambiente de dados, todas as APIs utilizaram a mesma tecnologia de

banco de dados (MySQL). Para evitar que o banco de dados se tornasse um limitador durante os testes, o limite de conexões ao banco foi aumentado[11]. Para garantir a estabilidade dos resultados, cada teste foi executado três vezes de forma independente, ao final foi calculado a média. Essa abordagem foi útil para reduzir as variações não controladas durante as execuções individuais.

No contexto do estudo comparativo entre os *frameworks* `Node.js`, `Django` e `.NET`, é crucial garantir a interpretação correta dos resultados ao escolher métricas que refletem seu desempenho essencial. Caso contrário, haveria a possibilidade de enviesamento nos resultados caso as métricas escolhidas não refletissem de maneira precisa o desempenho dos *frameworks*. Para isto, foram realizados estudos bibliográficos para a escolha das métricas que estivessem alinhadas com os objetivos do estudo, como discutidas na Seção 3.2.

Para validar externamente este estudo, foram considerados diferentes cenários de testes para maximizar a generalização dos resultados. Três cenários distintos foram considerados: teste de pico, teste de carga crescente e teste de resistência. Essa abordagem teve o propósito de viabilizando a aplicabilidade das conclusões em diferentes contextos.

8 Conslusão

Neste trabalho, realizou-se uma análise comparativa prática entre os *frameworks* de *back-end* `Node.js`, `Django REST Framework` e `.NET`. Tal análise envolveu a criação de três APIs nos *frameworks* em estudo. A comparação foi realizada com base em métricas de performance e desempenho, sendo elas consumo de recursos (CPU e Memória) e tempo de reposta. Essas métricas foram obtidas por meio da execução de três cenários de testes:

- Teste de pico;
- Teste de carga crescente;
- Teste de resistência;

Durante a análise dos testes práticos, observou-se que tanto o `Node.js` e o `Django` apresentaram melhores resultados em termos de eficiência no consumo de recursos, demonstrando uma utilização mais econômica em relação ao uso de CPU e memória em comparação com o `.NET`. Por outro lado, o `.NET` se destacou ao demonstrar os menores tempos de resposta em todos os cenários de teste. Esses resultados destacam as diferentes forças e características de cada *framework*, com o `Node.js` e o `Django` oferecendo uma eficiência de recursos e o `.NET` otimizando o desempenho e a latência nas respostas.

Como trabalhos futuros é sugerido a replicação deste estudo em outros cenários de testes, para verificar como é o comportamento dos *frameworks* em diferentes casos. Em outra configuração também é possível a avaliação utilizando outros bancos de dados, como por exemplo algum banco não relacional. Por fim, é sugerida a comparação dos *frameworks* em outros fluxos de requisições, como por exemplo a funcionalidade de *login*.

Em resumo, este estudo fornece uma visão acerca das diferenças entre os *frameworks* de *back-end* populares. Cada *framework* tem suas próprias vantagens e desvantagens, e a escolha ideal dependerá dos requisitos específicos do projeto e das restrições de recursos. Este estudo contribui para o entendimento das implicações práticas das escolhas de *frameworks* de *back-end* e espera-se que ele ajude os desenvolvedores a tomar decisões informadas em seus projetos futuros de sistemas de informação para Web.

[11] https://dev.mysql.com/doc/refman/8.0/en/too-many-connections.html

Referências

[1] V.R. Basili and H.D. Rombach. 1988. The TAME project: towards improvement-oriented software environments. *IEEE Transactions on Software Engineering* 14, 6 (1988), 758–773. https://doi.org/10.1109/32.6156

[2] Django REST Framework. 2023. *Documentação do Django REST Framework.* Acessado em 22-abril-2023, disponível em https://www.django-rest-framework.org/.

[3] Itti Hooda and Rajender Singh Chhillar. 2015. Software Test Process, Testing Types and Techniques. *International Journal of Computer Applications* 111, 13 (February 2015), 10–14.

[4] IBM. 2023. O que é uma API de REST? Acessado em 10 de julho de 2023: https://www.ibm.com/br-pt/topics/rest-apis.

[5] Devki Nandan Jha, Saurabh Garg, Prem Prakash Jayaraman, Rajkumar Buyya, Zheng Li, and Rajiv Ranjan. 2018. A Holistic Evaluation of Docker Containers for Interfering Microservices. In *2018 IEEE International Conference on Services Computing (SCC)*. 33–40. https://doi.org/10.1109/SCC.2018.00012

[6] Zhen Ming Jiang and Ahmed E. Hassan. 2015. A Survey on Load Testing of Large-Scale Software Systems. *IEEE Transactions on Software Engineering* 41, 11 (2015), 1091–1118. https://doi.org/10.1109/TSE.2015.2445340

[7] Kai Lei, Yining Ma, and Zhi Tan. 2014. 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*. 661–668. https://doi.org/10.1109/CSE.2014.142

[8] MDN. 2023. *A web e seus padrões.* Acessado em 09-junho-2023, disponível em https://developer.mozilla.org/pt-BR/docs/Learn/Getting_started_with_the_web/The_web_and_web_standards.

[9] Microsoft. 2023. *Arquitetura do .NET.* Acessado em 27-abril-2023, disponível em https://learn.microsoft.com/pt-br/dotnet/csharp/tour-of-csharp/#net-architecture.

[10] Khaled M. Mustafa, Rafa E. Al-Qutaish, and Mohammad I. Muhairat. 2009. Classification of Software Testing Tools Based on the Software Testing Methods. In *2009 Second International Conference on Computer and Electrical Engineering*, Vol. 1. 229–233. https://doi.org/10.1109/ICCEE.2009.9

[11] Node.js. 2023. *Documentação do Node.js.* Acessado em 12-abril-2023, disponível em https://nodejs.org.

[12] Omitido para fins de revisão. 2023. *Repositório das APIs implementadas.* Acessado em 27-agosto-2023, disponível em https://artigobackend.github.io/comparacao-backend/.

[13] Omitido para fins de revisão. 2023. *Tabela de artefatos dos cenários de teste.* Acessado em 27-agosto-2023, disponível em https://docs.google.com/spreadsheets/d/1lTfUHosUtCoHn3hhCvFhU-fBZRwufI_J6V25VlhmgRc/edit?usp=sharing.

[14] S. Pradeep and Yogesh Kumar Sharma. 2019. A Pragmatic Evaluation of Stress and Performance Testing Technologies for Web Based Applications. In *2019 Amity International Conference on Artificial Intelligence (AICAI)*. 399–403. https://doi.org/10.1109/AICAI.2019.8701327

[15] Anupam Sharma, Archit Jain, Ayush Bahuguna, and Deeksha Dinkar. 2020. Comparison and Evaluation of Web Development Technologies in Node.js and Django. *International Journal of Science and Research (IJSR)* 9, 12 (December 2020), 1416–1420. https://www.ijsr.net/getabstract.php?paperid=SR201202223534

[16] StackOverflow. 2022. *Stack Overflow Annual Developer Survey*. Acessado em 14-abril-2023, disponível em https://survey.stackoverflow.co/2022/#technology-most-popular-technologies.

[17] W3C. 2021. *A Little History of the World Wide Web*. Acessado em 09-junho-2023, disponível em https://www.w3.org/History.html.

[18] Junmei Wang and Jihong Wu. 2019. Research on Performance Automation Testing Technology Based on JMeter. In *2019 International Conference on Robots & Intelligent System (ICRIS)*. 55–58. https://doi.org/10.1109/ICRIS.2019.00023

[19] Dibyo Widodo, Prima Kristalina, Moch. Zen Samsono Hadi, and Aprilia Dewi Kurniawati. 2023. Performance Evaluation of Docker Containers for Disaster Management Dashboard Web Application. In *2023 International Electronics Symposium (IES)*. 551–556. https://doi.org/10.1109/IES59143.2023.10242411

[20] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. 2012. *Experimentation in Software Engineering*. Springer, Germany. https://doi.org/10.1007/978-3-642-29044-2