Benchmark de Frameworks Node.js para Desenvolvimento de APIs: Avaliação de Desempenho com Express, NestJS, Fastify e Next.js *

Pedro F. Poglia† Prof. Me. Jorge Luis Boeira Bavaresco ‡

2025

Resumo

A popularização do ambiente Node.js para o desenvolvimento de Interfaces de Programação de Aplicações (APIs) que seguem o estilo arquitetural *Representational State Transfer*(REST) na indústria de *software* deu origem ao surgimento de diversos *frameworks*, ampliando — e ao mesmo tempo dificultando — a escolha por parte dos desenvolvedores. Com o objetivo de auxiliar nessa escolha, este trabalho apresenta um *benchmark*, ou seja, uma análise comparativa focada em desempenho, de quatro dos *frameworks* mais populares do mercado utilizados no desenvolvimento de *APIs*: *Express*, *Fastify*, *NestJS* e *Next.js*. Mais detalhadamente, o estudo se concentra em três bibliotecas já consolidadas nessa área de atuação: *Express, Fastify e NestJS*. Além disso, aborda também uma tecnologia consolidada no mercado *front-end* que está, gradualmente, migrando para a criação de APIs e *back-ends*, passando a se tornar uma solução *full stack*: o *Next.js*. A metodologia adotada incluiu a implementação de quatro *APIs REST*, uma em cada *framework*, e a execução de testes de carga e estresse simulando usuários reais. utilizando o *k6*, uma conhecida ferramenta de testes. Os resultados obtidos foram analisados e revelaram que, embora todos os sistemas construídos apresentem desempenho satisfatório, existem diferenças significativas entre eles. Por fim, são apresentadas recomendações práticas para a escolha da abordagem de desenvolvimento mais adequada, conforme o tipo de projeto ou demanda.

Palavras-chaves: Node.js. Next. NestJS. Fastify. Express. benchmark.

*Trabalho de Conclusão de Curso (TCC) apresentado ao Curso de Bacharelado em Ciência da Computação do Instituto Federal de Educação, Ciência e Tecnologia Sul-rio-grandense, Campus Passo Fundo, como requisito parcial para a obtenção do título de Bacharel em Ciência da Computação, na cidade de Passo Fundo, em 2025.

†Instituto Federal de Educação, Ciência e Tecnologia Sul-rio-grandense – IFSul, Bacharelado em Ciência da Computação, Passo Fundo – RS, Brasil. E-mail: pedro.poglia@hotmail.com

‡Instituto Federal de Educação, Ciência e Tecnologia Sul-rio-grandense – IFSul E-mail: jorgebavaresco@ifsul.edu.br

1 Introdução

O *Node.js* foi criado por Dahl (2009). Ele é um ambiente de execução da linguagem de programação *JavaScript* (JS) do lado do servidor. Assim, permite executar código JS sem precisar de um navegador *web* como o *Microsoft Edge*, *Google Chrome* ou *Firefox*. O *Node* é mais comumente usado para desenvolver *APIs*, aplicações em tempo real, sistemas de *streaming* de dados e microsserviços, devido à sua alta performance e capacidade de gerenciar múltiplas conexões simultâneas (KUFFEL; WALTER, 2024). Essa tecnologia utiliza a inovação conhecida como o motor V8, que é um compilador de *JavaScript* de código aberto desenvolvido pelo *Google* e que também é utilizado pelo seu navegador *Google Chrome*.

O V8 (2024), criado em 2008, modernizou a forma de interpretação dos navegadores, que até então utilizavam motores de JS como *SpiderMonkey (Mozilla Firefox)*, *JScript (Internet Explorer)* e *KJS (Konqueror, Linux)*. É importante ressaltar que esses motores eram principalmente interpretadores, ou seja, processavam o código linha a linha, o que resultava em uma demora maior na sua leitura. O motor V8 já é conhecido como um compilador *Just-in-Time (JIT)*, que, diferentemente dos interpretadores, compila as instruções JS diretamente em código de máquina enquanto ele é executado, trazendo um desempenho superior quando exposto a grandes quantidades de requisições, dados e aplicações mais complexas.

A popularidade do *Node* deu origem ao maior ecossistema de pacotes, superando todas as outras ferramentas pré-existentes juntas, como *Maven Central (Java), Nuget (.NET)* e *Packagist (PHP)*. Com uma vasta quantidade de bibliotecas desenvolvidas pela comunidade e mantidas e acessadas através do gerenciador de pacotes do *Node*, o *node package manager (NPM)* (KUFFEL; WALTER, 2024). Essa expressiva quantidade de ferramentas que facilitam e auxiliam nos mais diversos serviços também trazem consigo um revés. Com tantas alternativas à disposição, os programadores muitas vezes se deparam com a dificuldade de escolher a ferramenta mais apropriada para suas demandas. A vasta gama de pacotes à disposição pode complicar a decisão, principalmente quando diferentes bibliotecas proporcionam funcionalidades parecidas, porém diferem em termos de desempenho, compatibilidade e características particulares. Escolher uma tecnologia imprópria pode resultar em problemas de escalabilidade, impacto negativo no rendimento e incremento da complexidade nas futuras manutenções, criando barreiras consideráveis no desenvolvimento e administração de aplicações.

A seleção da estrutura ideal para a criação de *APIs* e aplicações *web* tem se mostrado um desafio cada vez maior, considerando a vasta gama de alternativas disponíveis, incluindo *Express, NestJS e Fastify*. Cada um desses *frameworks* possui suas particularidades: o *Express* é frequentemente usado por sua simplicidade, e adaptabilidade, o *Nest* adota uma abordagem mais sólida, com ênfase em práticas como injeção de dependência e modularização, sendo apropriado para projetos complexos. Por outro lado, o *Fastify* se sobressai tanto da sua simplicidade quanto pelo seu olhar crítico para eficiência de recursos, enfatizando uso reduzido de memória.

Com a evolução das necessidades no desenvolvimento de aplicações, o *Next.js*, que foi inicialmente criado para o *front-end*, passou a oferecer recursos que possibilitam a construção de *APIs* e camadas de *back-end*. Suas rotas, *middlewares* e outras funcionalidades tornaram-no

uma opção viável para quem busca uma solução completa para o desenvolvimento de aplicações sem recorrer a *frameworks back-end* tradicionais. Nesse sentido, o *Next.js* se torna uma escolha interessante para equipes que preferem uma abordagem integrada, ou seja, uma abordagem *fullstack*.

Com isso, este estudo se propõe a realizar uma análise comparativa entre o *Next* e outros *frameworks* do ecossistema *Node.js*, *Express, NestJS e Fastify*, focando nas suas capacidades para o desenvolvimento de *back-end*. O objetivo é entender o desempenho do *Next.js* em comparação com essas opções mais estabelecidas, fornecendo informações úteis para os desenvolvedores escolherem a melhor ferramenta para suas necessidades específicas no desenvolvimento.

Para facilitar a compreensão deste estudo o documento está organizado da seguinte forma: a seção 2, Fundamentação Teórica, fornece uma visão geral de alguns conceitos preliminares nos quais este artigo se baseia. Mais precisamente, são introduzidas as tecnologias de origem e apresentados os *frameworks* do estudo. Em seguida, na seção 3, Ambiente de Testes, é descrito o ambiente de execução adotado para os testes e explicado o porquê de sua escolha, comparando-o com as outras alternativas analisadas e descartadas. Na seção 4, Metodologia, é introduzida a metodologia utilizada, assim como todo o processo de desenvolvimento e execução dos testes. Por fim, na seção 5, Resultados e Discussões, os resultados obtidos são apresentados, discutidos e interpretados, levando às conclusões na seção 6, Considerações Finais.

2 Fundamentação Teórica

Esta seção são abordadas as tecnologias e ferramentas centrais para a análise proposta, começando pela linguagem JavaScript e o ambiente de execução Node.js, que representam a base comum dos frameworks estudados. Em seguida, são detalhados individualmente os quatro frameworks selecionados para a comparação — *Express, Fastify, Nest.js* e *Next.js* — destacando suas características, arquiteturas e principais usos no desenvolvimento de aplicações web e APIs. Por fim, é apresentada a ferramenta k6, utilizada nos testes de desempenho, com foco nas suas funções em cenários de benchmark.

Essa contextualização teórica é essencial para compreender as decisões de implementação, a escolha dos frameworks e a análise dos resultados obtidos, proporcionando uma base sólida para o entendimento das comparações de desempenho realizadas ao longo do trabalho.

2.1 JavaScript

O *JavaScript* é, sem dúvida, uma das linguagens mais acessíveis para quem deseja se tornar desenvolvedor de *software*. Sua sintaxe simples e direta facilita o aprendizado, tornando-a extremamente versátil, pois pode ser utilizada tanto no *front-end* quanto no *back-end*. Isso a torna uma escolha ideal para iniciantes, mas também para profissionais que querem expandir suas habilidades e explorar diferentes áreas da programação.

Comumente abreviado para JS, o *JavaScript* é uma linguagem interpretada, leve e orientada a objetos, que também permite o uso de funções de alta complexidade. Embora seja amplamente reconhecida como a principal linguagem de *script* para páginas *web*, sua utilização

vai muito além dos navegadores, sendo empregada também em plataformas como *Node.js, Apache CouchDB* e *Adobe Acrobat*. A linguagem é notável por ser fundamentada em protótipos, dinâmica e multi-paradigma, permitindo a aplicação de vários estilos de programação, como os orientados a objetos, imperativos e declarativos, além de proporcionar suporte à programação funcional (MDN, 2024).

A linguagem tem um papel crucial na criação de aplicativos web dinâmicos e interativos. Ela é frequentemente implementada por navegadores de renome, como *Google Chrome, Mozilla Firefox* e *Microsoft Edge*, que possuem motores especializados para aprimorar a execução do código. Esses motores asseguram que as aplicações JS operem de maneira eficaz e constante, proporcionando um desempenho satisfatório em várias plataformas. Além de seu uso em sites e aplicativos, também desempenha um papel central em diversas outras áreas, como no processamento de dados complexos. Um exemplo, é sua aplicação em *widgets* de visualização interativa de dados, como os utilizados em pesquisas científicas e genômicas, destacando a flexibilidade e o alcance da linguagem para além do desenvolvimento *web* tradicional (PEARCE; NIKIFOROVA; ROY, 2019).

2.2 Node.js

O *Node.js* é uma plataforma de desenvolvimento amplamente apreciada pela sua eficácia e adaptabilidade no contexto de criação de aplicações contemporâneas. Desde que foi lançada em 2009, ela se sobressaiu ao usar *JavaScript* no lado do servidor, possibilitando uma integração constante entre o *back-end* e o *front-end*.

No âmbito teórico, *Node.js* simboliza um progresso na maneira como as estruturas de *software* gerenciam a performance e a administração de recursos. Ao optar por um modelo de entrada/saída (I/O) não bloqueado e orientado a eventos, a plataforma proporciona uma alternativa sólida às estratégias convencionais baseadas em *multithreading*, particularmente em aplicações que demandam grande capacidade de processamento simultâneo, como *APIs* em tempo real, bate-papos e transmissão de dados em tempo real (NODE.JS, 2024).

O motor V8, criado pelo *Google*, é um dos pilares fundamentais do *Node.js*, convertendo o código JS em linguagem de máquina de alta performance. Essa eficácia possibilita que os desenvolvedores obtenham tempos de resposta mais curtos e aproveitem de maneira mais veloz os recursos do servidor, sendo perfeita para companhias que gerenciam grandes quantidades de dados.

A estrutura modular do *Node.js*, fundamentada em pacotes acessíveis através do gestor NPM, também tem um papel importante na sua adoção em larga escala. Essa característica facilita a integração de ferramentas e bibliotecas, diminuindo o tempo de desenvolvimento e incrementando a produtividade (NODE.JS, 2024).

Portanto, o ambiente de execução não é apenas uma ferramenta, mas um paradigma que exemplifica a transição para arquiteturas orientadas a eventos e o uso unificado de linguagens no desenvolvimento *web*. Ele continua a ser um referencial importante para pesquisas sobre eficiência de *frameworks*, adaptação a demandas modernas e o impacto de tecnologias *open source* no

mercado.

2.3 Express

O *Express.js*, tecnologia popular no ecossistema *Node.js*, destaca-se por sua abordagem extremamente simples e concisa, o que possibilita a criação de aplicações *web* personalizadas e eficientes com um número menor de linhas de código. A ferramenta contribui para a otimização do desempenho e a agilidade no desenvolvimento de projetos. Sua abordagem minimalista é altamente apreciada, pois permite que os programadores incluam apenas os recursos indispensáveis para atender às demandas específicas de cada projeto.

Projetado para simplificar o desenvolvimento de aplicativos *web* e *APIs*, a solução oferece uma camada de abstração sobre o núcleo *Hypertext Transfer Protocol* (HTTP) do *Node.js*, permitindo que os desenvolvedores criem *APIs* robustas de forma rápida e eficiente. Além disso, a versão 5.0 já está com a documentação disponível, enquanto a versão $4.x$ continua evoluindo.

De acordo com (MARDAN, 2014), o *Express* é construído sobre o núcleo do módulo *HTTP* e utiliza *middlewares* para conectar as diversas funcionalidades da aplicação. Isso dá ao desenvolvedor a liberdade de escolher bibliotecas e pacotes, tornando o *Express* altamente flexível e personalizável. Essa abordagem minimalista permite que o desenvolvedor adicione apenas as funcionalidades necessárias, sem sobrecarregar o sistema, o que é especialmente vantajoso para quem busca agilidade no desenvolvimento.

O *Express* também é perfeito para a organização de rotas e para a apresentação de páginas *HTML* interativas. Ademais, ele simplifica a administração de recursos habituais em aplicações *web*, tais como corpos de pedidos *HTTP*, *cookies* e sessões. A sua estrutura modular possibilita a incorporação simples de *middlewares*, que podem ser empregados para solucionar questões específicas de cada aplicativo. A clareza da documentação oficial do *Express* fazem dele uma das opções preferidas para o desenvolvimento *web* em *Node.js*, proporcionando um equilíbrio entre simplicidade e eficiência (HAHN, 2016).

Essa tecnologia se sobressai na programação *web* devido à sua metodologia simples, oferecendo alto desempenho sem prejudicar a sua função. A sua estrutura modular e a capacidade de suportar *middlewares* possibilitam que os programadores desenvolvam aplicações de fácil expansão e manutenção. O *framework* ainda é amplamente utilizado em projetos de várias dimensões, o que o torna uma opção excelente para quem procura eficácia na criação de *APIs* e sistemas *web* contemporâneos.

2.4 Fastify

O *Fastify* é um *framework web* que se destaca por focar na experiência do desenvolvedor, apresentando mínimo *overhead* e uma arquitetura de *plugins* robusta. Inspirado nas tecnologias *Hapi* e *Express*, ele se apresenta como um dos *frameworks web* mais rápidos disponíveis atualmente (FASTIFY, 2024).

O *Fastify* também se destaca pela sua facilidade de uso. Criado para ser intuitivo, ele permite que os desenvolvedores realizem tarefas diárias de forma simples, sem comprometer

a segurança. Adicionalmente, oferece suporte nativo ao *TypeScript*, facilitando a integração com ambientes modernos de desenvolvimento. Apesar dessas qualidades e de seu potencial em otimizar o desenvolvimento, é relevante notar que o *Fastify* ainda possui uma escassez de artigos científicos e estudos aprofundados que o abordem, o que pode limitar sua visibilidade em contextos acadêmicos.

2.5 NestJS

NestJS é mais uma tecnologia da família *Node.js*, projetado para criar aplicações escaláveis e de alta *performance*, e se destaca pela sua abordagem modular que facilita a manutenção e organização de código. Trata-se da principal escolha para aqueles que já estão habituados ao *framework Angular* por fundamentar seus princípios de *design*. Sua estrutura facilita o desenvolvimento de sistemas grandes e complexos, promovendo uma organização clara do código.

Ele se distingue da maioria pelo uso do *TypeScript*, o que oferece maior proteção e simplifica a criação de código organizado e de fácil compreensão. Ao utilizar uma estrutura modular, o *NestJS* estrutura o código de forma transparente, simplificando a manutenção e a inclusão de novas funcionalidades sem prejudicar a estabilidade do sistema (NESTJS, 2024).

Além disso, *Nest* integra de forma simples tecnologias como *APIs RESTful*, *microservices* e *GraphQL*, tornando-se uma ferramenta versátil para o desenvolvimento de sistemas mais sofisticados. A modularidade e o uso de injeção de dependência são elementos chave que permitem o desacoplamento dos componentes do sistema, facilitando o teste, manutenção e a integração de novos recursos. Por fim, o *Nest* é compatível com ferramentas de teste automatizado, como testes unitários e de integração, fundamentais para garantir a qualidade do código em projetos de grande porte, reforçando sua utilidade em ambientes empresariais (NESTJS, 2024).

2.6 Next.js

O *Next.js* é uma biblioteca desenvolvida para ser utilizada com a tecnologia *React* (um *framework JavaScript*), e opera de maneira semelhante a outras ferramentas do mesmo ecossistema. Ele facilita o desenvolvimento de aplicações *web* escaláveis e de alto desempenho, especialmente por sua capacidade de renderização do lado do servidor (*Server-Side Rendering* ou *SSR*). Esse recurso permite gerar páginas dinâmicas no servidor antes de enviá-las ao cliente, o que melhora a *performance* e otimiza os mecanismos de busca, já que o conteúdo é enviado ao navegador de forma pré-renderizada.

Além disso, ele permite a construção de páginas dinâmicas e estáticas dentro do mesmo projeto, oferecendo flexibilidade no gerenciamento de dados e conteúdo (VERCEL, 2024). A estrutura do *Next* também se destaca pela simplicidade de uso, com um sistema de roteamento baseado em arquivos que facilita a criação de rotas e componentes, sem necessidade de configuração complexa. O *framework* ainda oferece integração com *APIs*, otimizações automáticas e um sistema de pré-carregamento de páginas, permitindo que os desenvolvedores criem aplicativos de alto desempenho com facilidade (VERCEL, 2024).

Além de suas características voltadas para o desenvolvimento no *front-end*, recentemente

ele também vem conquistando seu espaço no cenário de *back-end*. A partir da versão 9.3, o *Next.js* permitiu a criação de funções de *API* no próprio *framework*, possibilitando aos programadores a criação de rotas de *API* diretamente nas aplicações, sem a exigência de configurar um servidor separado. Isso significa que a tecnologia agora também pode ser empregada na criação do *back-end* de uma aplicação, o que diminui a complexidade do projeto. Isso ocorre porque o *Next.js* pode ser usado em todo o sistema, eliminando a necessidade da equipe de desenvolvimento dominar mais de uma linguagem ou biblioteca para montar um serviço completo.

Devido a esta novidade, o *Next.js* se destaca no presente trabalho, que tem como objetivo realizar um *benchmark* de *frameworks Node.js*. A inclusão dessa funcionalidade no *Next.js* contribui para a ampliação de seu escopo, o que o coloca como uma alternativa viável não apenas para o *front-end*, mas também para o *back-end* de aplicações, fortalecendo sua posição no cenário de *frameworks* para a construção de *APIs* e aplicações *web*.

2.7 k6

O *k6* é uma ferramenta moderna e de código aberto voltada para testes de carga e desempenho. Diferentemente de ferramentas tradicionais com interface gráfica, ele adota uma abordagem baseada em *scripts* escritos em JS, o que proporciona maior facilidade de versionamento dos testes. Assim, sendo útil em *pipelines* de *DevOps* devido à sua abordagem e sua facilidade de ser adicionado em ambientes automatizados.

Com o *k6*, é possível simular o comportamento de usuários reais acessando *APIs REST*, páginas web ou outros serviços, configurando o número de usuários virtuais, a duração do teste, os padrões de carga e as validações esperadas para as respostas. A ferramenta oferece suporte nativo a métricas como tempo de resposta, taxa de sucesso, *throughput*, erros por segundo e percentis, além de permitir a exportação desses dados para sistemas de monitoramento e análise, como *Grafana* e *InfluxDB* (LABS, 2025).

A simplicidade na escrita de scripts, combinada com a robustez dos relatórios gerados, permite uma análise clara e detalhada sobre o desempenho da aplicação sob diferentes níveis de estresse. O *k6* também possibilita a execução de testes distribuídos, ampliando seu uso em ambientes com grande necessidade de simulação de tráfego (LABS, 2025).

Assim, essa ferramenta se destaca como uma alternativa moderna e eficiente para testes de carga, sendo uma opção leve, fácil de configurar e com grande capacidade de integração com fluxos de trabalho automatizados de desenvolvimento.

3 Ambiente de Testes

Nesta seção, descrevemos o ambiente utilizado para a execução dos testes de desempenho entre os *frameworks* selecionados: *Express, NestJS, Fastify e Next.js*. O objetivo é garantir condições controladas, reproduzíveis e compatíveis com o cenário real de uso em que aplicações geralmente são desenvolvidas e executadas.

Para manter a fidelidade a um contexto prático comum entre desenvolvedores indepen-

dentes e pequenas equipes, optou-se por realizar os testes diretamente em uma máquina pessoal, simulando um ambiente local de desenvolvimento.

3.1 Justificativa do Ambiente

O ambiente de testes utilizado foi o notebook pessoal do autor, com as seguintes especificações:

- Processador: 11ª Geração Intel (R) Core (TM) i5-1135G7 @ 2.40GHz;
- Memória RAM instalada: 16 GB (15,7 GB utilizável);
- Tipo de sistema: 64 bits, processador baseado em x64;
- Sistema operacional: Windows.

O uso da máquina pessoal como ambiente de testes é justificado por refletir um cenário comum a desenvolvedores que trabalham de forma independente ou em equipes pequenas, onde a maioria dos softwares é criada. Esse ambiente normalmente não se dispõe de infraestrutura em nuvem ou servidores dedicados apenas para *benchmarking*. Além disso, proporciona agilidade na execução dos testes e maior controle sobre o sistema operacional, os processos ativos e a rede local.

A máquina utilizada possui recursos compatíveis com o perfil médio de uso para desenvolvimento web moderno, com um processador da 11ª geração da *Intel* e 16 *gigabytes (GB)* de *Random Access Memory (RAM)*, o que é suficiente para executar múltiplas instâncias de servidores e clientes simultaneamente sem gargalos significativos. Isso garante que os testes não sejam limitados por restrições de hardware, permitindo uma avaliação justa do desempenho dos *frameworks*.

3.2 Banco de Dados

A escolha de manter o banco de dados em ambiente local durante os testes também é intencional. Isso reduz a influência de fatores externos como latência de rede, variações de carga em servidores remotos e interrupções de conexão, que poderiam comprometer a precisão das medições. Dessa forma, é possível isolar o desempenho dos *frameworks*, garantindo que os resultados reflitam o comportamento dos mesmos sob condições semelhantes.

O *PostgreSQL* foi o banco escolhido por se tratar de um banco de dados relacional, isto é, que usa a *Structured Query Language (SQL)*, *open source* e amplamente utilizado em ambientes de produção, inclusive em conjunto com aplicações *Node*. Bancos de dados relacionais são sistemas que armazenam dados em tabelas interligadas por relacionamentos, seguindo um modelo estruturado e com forte apoio à consistência dos dados. Diferente dos bancos *Not Only SQL (NoSQL)*, os relacionais trabalham com esquemas definidos (*schemas*), o que permite um controle mais rígido sobre o formato e os tipos de dados armazenados.

Essa abordagem foi adequada ao contexto dos testes, já que muitas aplicações *web* modernas que utilizam os *frameworks* avaliados, frequentemente se integram com bancos relacionais

como *PostgreSQL* ou *MySQL*. Com isso, além de avaliar o desempenho das tecnologias em si, também é possível verificar como elas interagem com operações típicas de leitura e escrita em um banco relacional — uma situação realista e prática para diversos tipos de aplicações.

Além disso, a escolha de um banco de dados padrão nos testes permite comparar as bibliotecas sob condições equivalentes, padronizando a camada de persistência de dados e garantindo que as diferenças de desempenho observadas estejam realmente relacionadas à implementação dos *frameworks*, e não à tecnologia de banco utilizada.

4 Metodologia

A presente seção descreve o processo de desenvolvimento, configuração e realização dos testes de desempenho utilizados para avaliar os *frameworks* selecionados. O objetivo foi comparar sua *performance* em operações: *Create* (Criar), *Read* (Ler), *Update* (Atualizar) e *Delete* (Excluir) o *CRUD*, com foco em simular cenários práticos, controlados e padronizados.

Inicialmente, foi criado um banco de dados *PostgreSQL* contendo uma única tabela chamada *users* (usuários), composta por três campos: *id* (chave primária da tabela), *name* e *email*. Essa estrutura simplificada foi intencionalmente adotada para reduzir o impacto de operações complexas de banco de dados sobre os resultados dos testes, permitindo que o foco permanecesse na performance dos *frameworks* propriamente ditos. Utilizar um banco de dados mais complexo com relacionamentos, *joins* ou *triggers* poderia enviesar a análise ao transferir a carga computacional para a camada de persistência, desviando o foco da comparação.

Com o banco de dados criado, foram implementadas quatro *APIs REST* distintas, cada uma utilizando uma das bibliotecas estudadas. Todas as *APIs* seguiram a mesma estrutura e expuseram as quatro operações fundamentais do padrão *CRUD*, que, conforme as boas práticas de design de *APIs RESTful*, são mapeadas diretamente para os métodos *HTTP: POST, GET, PUT e DELETE*, respectivamente, assim como na Figura 1.

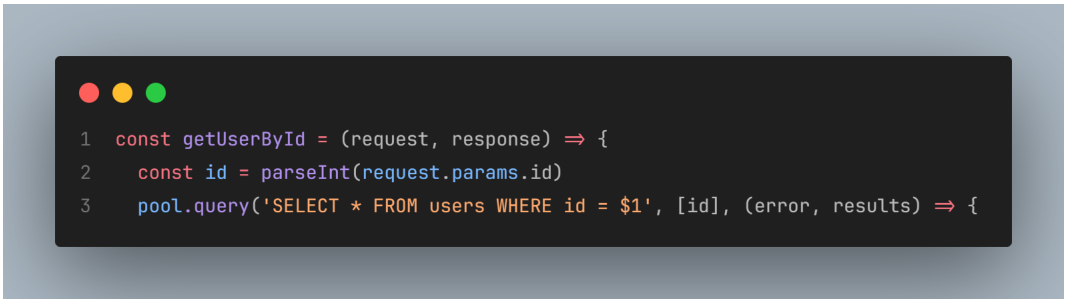
Figura 1 – Mapeamento CRUD para Rotas RESTful

Fonte: Do Autor, 2025

As operações foram implementadas da forma mais simples e direta possível, com base nas documentações oficiais de cada *framework*, com o objetivo de evitar a introdução de complexidades ou otimizações específicas que poderiam afetar negativamente a imparcialidade dos testes.

Além disso, optou-se por não utilizar *Object-Relational Mappers (ORM)*, como *Sequelize, Prisma* ou *TypeORM*, a fim de evitar que suas abstrações impactassem os resultados. Em vez disso, a comunicação com o banco de dados foi realizada diretamente por meio da biblioteca nativa *pg*, do ecossistema *Node*. Essa abordagem permitiu maior controle sobre as consultas *SQL* utilizadas, garantindo uniformidade na execução de operações em cada uma das *APIs*. As *queries* foram inseridas diretamente no código-fonte, sendo abstraidas apenas as variáveis que seriam passadas por parâmetro nas funções como mostra a Figura 2.

Figura 2 – Exemplo de código com SQL integrado



Fonte: Do Autor, 2025

Essa abordagem elimina qualquer sobrecarga adicional causada por bibliotecas intermediárias e possibilita uma comparação mais fiel do desempenho bruto dos *frameworks*, especialmente no tratamento de requisições, resposta ao cliente e gerenciamento de conexões simultâneas.

As rotas das *APIs* seguiram um padrão uniforme, conforme exemplificado anteriormente na Figura 1. O uso de nomenclaturas e estruturas idênticas para as rotas em todos os *frameworks* garante que o comportamento das aplicações fosse comparável, assegurando que eventuais variações de desempenho estivessem relacionadas exclusivamente às diferenças intrínsecas de cada *framework* e também facilitando na etapa dos testes.

Para a execução dos testes de carga, foi empregada a biblioteca *k6*, conforme detalhado na subseção 2.7. Os testes foram estruturados para simular diferentes níveis de demanda, abrangendo as operações *GET, POST e PUT* da API. Para essas rotas, foram realizados dois cenários distintos de carga: um com 100 usuários virtuais simultâneos (VUs) e outro com 500 VUs, ambos com duração de 30 segundos. Essa abordagem permitiu-nos avaliar o desempenho sob cargas médias e intensas.

Para garantir a confiabilidade dos resultados, cada teste foi repetido três vezes em um curto intervalo de tempo. A média dos resultados obtidos nessas execuções foi utilizada para a análise final, totalizando aproximadamente 100 testes.

4.1 Abordagem Específica para a Rota DELETE

A rota *DELETE* exigiu uma abordagem particular devido a um conflito observado durante testes com múltiplos VUs. Quando testada com 100 VUs, múltiplas requisições de delete eram direcionadas aos mesmos dados. Embora apenas uma dessas requisições resultasse na

exclusão efetiva, todas eram registradas como requisições aceitas, mascarando a real capacidade de processamento da rota.

Para contornar essa questão, o teste da rota *DELETE* foi executado com apenas 1 VU durante os 30 segundos e padronizadamente, foram feitos três testes e extraídas suas médias. Com essa configuração, a lógica do teste envolve primeiramente uma requisição *GET* para uma rota */any*, que retorna um dado aleatório a ser excluído. Em seguida, esse dado é utilizado na requisição *DELETE*. O objetivo aqui foi avaliar o número de operações de exclusão bem-sucedidas que a rota conseguiu processar nesse período, garantindo que cada *delete* correspondesse a uma exclusão real e individual. A métrica principal a ser analisada para essa rota é o número de exclusões realizadas e a sua quantidade de exclusões por segundo.

5 Resultados e Discussões

Nesta seção, são apresentados e analisados os resultados obtidos a partir da execução dos testes de carga realizados com as quatro *APIs REST* desenvolvidas. O objetivo principal é comparar o desempenho das implementações com base em métricas fornecidas pela ferramenta de testes *k6*, como tempo de resposta, taxa de requisições por segundo, percentis de latência e taxa de falhas.

Cada teste foi executado sob diferentes níveis de carga: 100 e 500 VUs, abrangendo as operações *GET, POST e PUT* do *CRUD*. Para a operação *DELETE*, a metodologia de teste foi adaptada para utilizar apenas 1 VU, focando na capacidade de processamento individual, conforme detalhado na Seção 4. Essa abordagem segmentada permite uma compreensão abrangente da performance de cada *framework* em cenários que simulam tanto cargas médias quanto intensas.

Os relatórios gerados pelos *scripts* do *k6* são bastante completos, possuindo dezenas de informações que podem ser visualizadas. Mas para manter o foco e do artigo, são mensuradas apenas as métricas mais relevantes para a análise de *benchmark*.

- Média (ms): Indica a média geral dos tempos de retorno das requisições, fornecendo uma visão rápida da performance.
- Média p (95) (ms): Representa o tempo abaixo do qual 95% das requisições foram concluídas. Essa métrica é crucial pois revela a consistência do desempenho, mostrando que mesmo 95% dos usuários experimentarão um tempo de resposta dentro desse limite, e não apenas a média.
- Requisições (quantidade): Avalia a capacidade de cada *framework* de lidar com volume. A divisão desse valor pelo tempo resulta no *throughput*, uma métrica crucial para o *benchmark*.
- Falhas %: A porcentagem de requisições que resultaram em erro. Idealmente, esse valor deve ser zero ou muito próximo, indicando a robustez da aplicação.

5.1 Abordagem Inicial (100 VUs)

Nesta seção, os resultados são apresentados em gráficos e tabelas, acompanhados por análises que interpretam o desempenho de cada *framework* em relação aos diversos contextos de uso e operações *CRUD*. A análise começa com a primeira abordagem, que engloba as operações *GET, POST e PUT*. Primeiramente, os dados são exibidos em tabelas gerais, sendo posteriormente segmentados para uma melhor compreensão.

Figura 3 – Tabela de GET, POST e PUT Com 100 VUs

Framework	Operação	VUs	Média (ms)	Requisições(quantidade)	Média p(95) (ms)	Falhas %
Express	GET	100	48.28	62111.33	80.86	0.00%
Express	POST	100	27.56	108506.00	32.80	0.00%
Express	PUT	100	31.36	95355.33	41.43	0.00%
Fastify	GET	100	21.22	140756.00	25.62	0.00%
Fastify	POST	100	15.29	195096.67	18.98	0.00%
Fastify	PUT	100	18.22	164937.67	23.49	0.00%
Nest	GET	100	29.66	101695.33	57.82	0.00%
Nest	POST	100	29.86	100192.67	40.42	0.00%
Nest	PUT	100	36.02	84013.00	54.51	0.00%
Next.js	GET	100	4800.00	661.67	5710.00	0.00%
Next.js	POST	100	4740.00	703.00	5580.00	0.00%
Next.js	PUT	100	4780.00	700.00	5440.00	0.00%

Fonte: Do Autor, 2025

A Figura 3, que avalia o desempenho sob 100 VUs (carga moderada), demonstra que todos os *frameworks* registraram 0.00% de falhas nas operações. Essa ausência total de falhas é um indicador altamente positivo, pois atesta que, sob essa demanda, a estabilidade das aplicações não foram comprometidas.

Para facilitar a análise e a visualização dos resultados obtidos, e evitar a sobrecarga de informações em representações gráficas individuais, os dados de desempenho serão aglutinados por *framework* em uma tabela consolidada. Essa abordagem permitirá uma comparação mais eficiente entre as arquiteturas, destacando as diferenças sem comprometer a clareza da apresentação.

Tabela 1 – Dados Aglutinados das Requisições GET POST e PUT com 100 VUS

Frameworks	Média (ms)	p(95) (ms)	Requisições	falhas %
Express	35.73	51.70	88657.55	0.00%
Fastify	18.24	23.23	166930	0.00%
NestJS	31.85	50.92	95300	0.00%
Next	4773	5576	688.22	0.00%

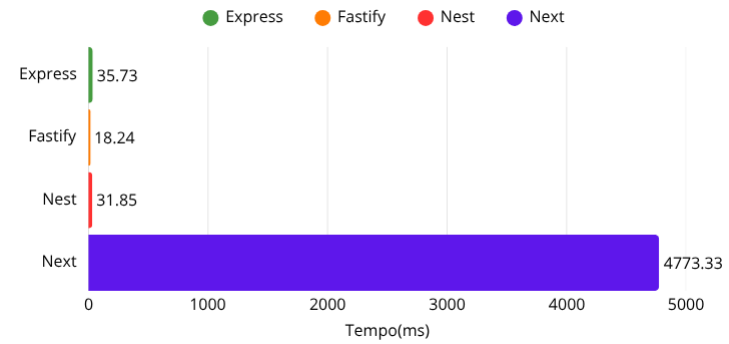
Fonte: Do Autor, 2025

Considerando os dados, o *Next.js* demonstrou um desempenho inversamente proporcional aos demais frameworks: enquanto os demais apresentaram tempos de resposta baixos e alto volume de requisições, o *Next.js* exibiu tempos de resposta maiores e um volume de requisições menores. Também já é possivel visualizar uma leve vantagem da tecnologia *Fastify* sobre as demais com tempos de resposta médios e percentis mais velozes.

5.1.1 Média

Nas médias de tempo de resposta, o *Fastify* se sobressai com 18.24 ms, demonstrando agilidade superior. *Express* (35.73 ms) e *Nest* (31.85 ms) exibem médias mais elevadas, mas ainda eficientes. O *Next.js*, contudo, registra um tempo de 4.77 segundos, revelando uma performance drasticamente inferior. Conforme fica evidente na figura Figura 4.

Figura 4 – Gráfico do Tempo Médio de Resposta por *framework*

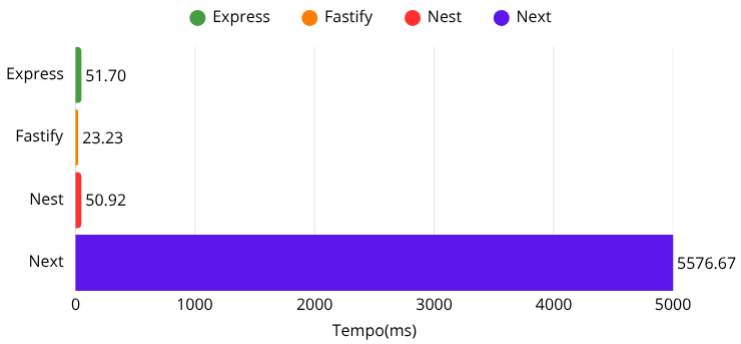


Fonte: Do Autor, 2025

5.1.2 Tempo de Resposta p(95)

O *Fastify*, com um p(95) de 23.23 ms, demonstra a menor variabilidade, indicando que a vasta maioria das requisições é processada de forma consistentemente rápida. *Nest* (50.92 ms) e *Express* (51.70 ms) apresentam valores de p(95) maiores que os do *Fastify*, sugerindo maior dispersão nos tempos de resposta para uma parcela das requisições, embora ainda em patamares que garantem a estabilidade. Por fim, o *Next.js*, com um p(95) alarmante de 5576 ms, reafirma que a latência elevada é uma característica persistente para as suas operações, e não apenas um evento esporádico. Como ilustrado na Figura 5, percebe-se essa diferença expressiva.

Figura 5 – Gráfico do Tempo de Resposta no Percentil 95 (p95) por Framework

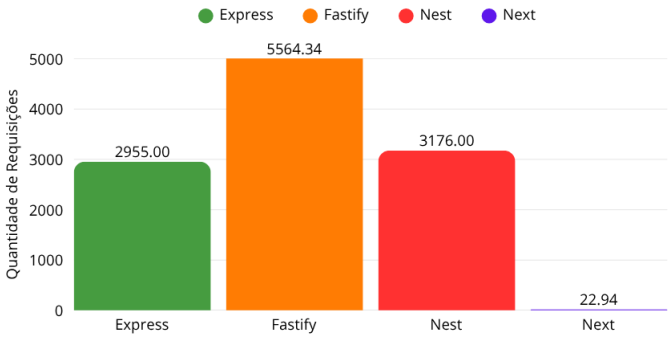


Fonte: Do Autor, 2025

5.1.3 Throughput (Requisições por Segundo)

A coluna "Requisições" na Tabela 1 representa o volume total de requisições processadas em um período de 30 segundos. Para uma análise da capacidade de vazão (*throughput*), calculamos as Requisições por Segundo (RPS). O *Fastify* demonstra uma *performance* excepcional, alcançando um *throughput* de 5564.34 RPS. *Nest* (3176.68 RPS) e *Express* (2955.25 RPS) apresentam capacidades competitivas. Em contraste, o *Next.js* exibe um *throughput* dramaticamente baixo de apenas 22.94 RPS, corroborando as observações de sua alta latência e indicando uma severa limitação em sua capacidade de processar requisições sob a carga imposta. Como ilustrado na Figura 6.

Figura 6 – Gráfico de Throughput



Fonte: Do Autor, 2025

5.2 Abordagem Inicial (500 VUs)

A Figura 7 detalha o desempenho dos *frameworks* sob uma carga mais intensa, com 500 usuários virtuais simultâneos. Este cenário mais exigente foi crucial para identificar os limites e a resiliência de cada tecnologia. Da mesma forma que foi feito com 100 VUs, também foram aglutinados os resultados para uma melhor análise como mostra a Tabela 2.

Figura 7 – Tabela de GET, POST, PUT Com 500 VUs

Framework	Operação	VUs	Média (ms)	Requisições(quantidade)	Média p(95) (ms)	Falhas %
Express	GET	500	256.72	58942.33	342.33	1.08%
Express	POST	500	146.06	102631.67	170.57	0.38%
Express	PUT	500	170.16	88315.33	225.76	0.60%
Fastify	GET	500	63.05	237587.33	81.26	0.05%
Fastify	POST	500	80.64	185793.33	99.73	0.01%
Fastify	PUT	500	92.01	162831.33	112.70	0.00%
Nest	GET	500	133.82	112114.00	162.93	0.25%
Nest	POST	500	157.48	95311.67	181.60	0.14%
Nest	PUT	500	178.52	84368.33	221.65	0.11%
Next.js	GET	500	16030.00	1665.67	31900.00	67.86%
Next.js	POST	500	22330.00	1067.00	31560.00	52.20%
Next.js	PUT	500	20160.00	1522.00	31960.00	59.37%

Fonte: Do Autor, 2025

Tabela 2 – Dados Aglutinados das Requisições GET POST e PUT com 500 VUS

Frameworks	Média (ms)	p(95) (ms)	Requisições	falhas %
Express	190.98	246.22	83296.44	0.69%
Fastify	78.57	97.90	195404	0.02%
NestJS	156.61	188.73	97264	0.17%
Next	19506.67	31806.67	1418.22	59.81%

Fonte: Do Autor, 2025

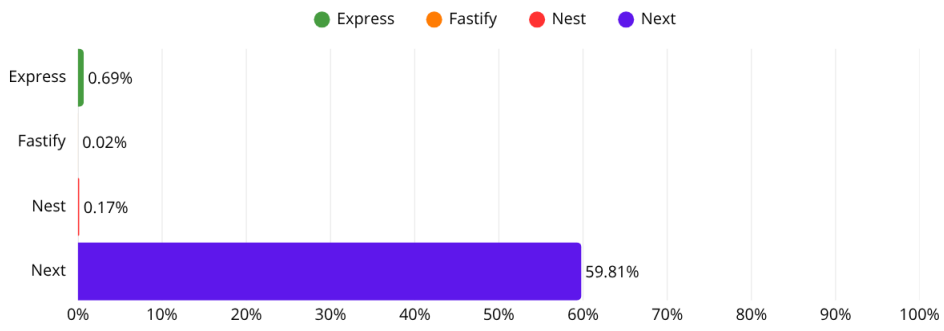
Expandindo a análise para 500 VUs, é natural observar um aumento nos tempos médios de resposta para todos os *frameworks*. Este fenômeno é inerente ao escalonamento de sistemas, onde o incremento no volume de requisições eleva a demanda por recursos computacionais (*CPU*, memória, rede), podendo levar a filas de processamento e, consequentemente, a um maior tempo para concluir cada operação.

No entanto, a capacidade de cada *framework* de absorver esse aumento de carga sem um declínio drástico no desempenho varia significativamente. Enquanto *Fastify*, *Express* e *Nest* demonstram um aumento gerenciável em seus tempos de resposta, o *Next.js*, por sua vez, exibe um salto exponencial. Seus tempos disparam para patamares inviáveis nesses testes, atingindo aproximadamente 19 segundos por requisição, reforçando as limitações de sua arquitetura para lidar com concorrência elevada em comparação com os demais.

Visando clareza e concisão, não foram feitas formas de visualização adicionais para métricas de desempenho (tempo médio, p(95), *throughput*) referentes a 500 VUs. O aumento de 400% (100 VUs para 500 VUs) na carga é compatível com o incremento esperado nesses valores, e apresentá-los novamente em figuras separadas seria redundante.

No cenário de 500 VUs, o ponto de maior relevância para avaliar a resiliência dos *frameworks* sob estresse elevado é a taxa de falhas, que aponta diretamente para a capacidade do sistema de entregar respostas válidas de forma consistente. Sob 100 VUs, todos os *frameworks* apresentaram 0.00% de falhas. Contudo, ao escalar para 500 VUs, a estabilidade de cada *framework* é posta à prova de forma mais rigorosa.

Figura 8 – Gráfico de Porcentagem de falhas sobre o total das requisições



Fonte: Do Autor, 2025

- *Fastify* e *Nest* mantêm taxas de falhas extremamente baixas (0.02% e 0.17%, respectivamente), o que demonstra a eficiência na gestão de recursos mesmo sob maior estresse. Isso indica que conseguem processar um volume muito maior de requisições sem comprometer a integridade das operações.

- *Express* apresenta uma taxa de falhas de 0.69%. Embora superior a *Fastify* e *Nest*, ainda é uma porcentagem baixa e aceitável, demonstrando que o *framework* é capaz de lidar com a carga, mas com uma pequena desvantagem em comparação aos líderes.

- *Next*, por outro lado, registra uma alarmante taxa de falhas de quase 60% (59.81%). Uma taxa de falhas tão elevada sugere que a aplicação não consegue processar a vasta maioria das requisições, resultando em *timeouts*, erros internos e sobrecarga, tornando-o impraticável para cenários com essa demanda de usuários.

5.3 Abordagem Específica

Como foi detalhado na subseção 4.1. Para assegurar a avaliação precisa da capacidade de processamento de exclusões individuais, os testes foram conduzidos com apenas 1 VU durante 30 segundos. Essa configuração permitiu que cada operação *DELETE* correspondesse a uma exclusão real de um dado previamente obtido, evitando requisições que não resultassem em uma remoção efetiva.

As métricas primárias focam na quantidade total de exclusões realizadas e na sua capacidade de chamadas por segundo (RPS). Os dados abaixo referem-se a média dos três testes feitos com cada tecnologia.

Tabela 3 – DELETE Com 1 VU durante 30 segundos

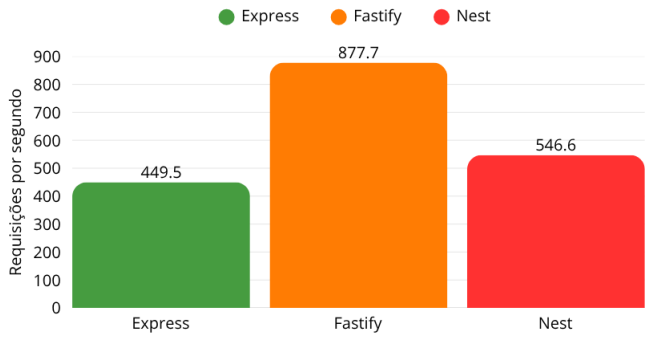
Frameworks	Exclusões	Exclusões por Segundo (RPS)
Express	13486	449.52
Fastify	26331	877.68
NestJS	16399	546.60
Next	101.17	3.35

Fonte: Do Autor, 2025

Considerando o desempenho inferior consistente do *Next* nas análises anteriores, e agora com sua limitação de aproximadamente 3 exclusões por segundo na rota *DELETE*, em nítido contraste com as centenas de requisições por segundo processadas pelos demais *frameworks*, sua inclusão nos gráficos desta rota poderia distorcer a visualização e dificultar a comparação clara entre *Express, Fastify e Nest*. Portanto, para otimizar a legibilidade e focar nas diferenças de desempenho mais relevantes, os gráficos subsequentes apresentarão apenas os resultados dessas três tecnologias.

A análise do Gráfico de Requisições por Segundo (RPS) na rota *DELETE* revela uma clara distinção na capacidade de vazão entre os *frameworks*. O *Fastify* se estabelece como líder incontestável em *throughput*, demonstrando uma notável eficiência na execução de operações de exclusão com uma média de 877.68 RPS, superando substancialmente *Nest* (546.60 RPS) e

Figura 9 – Gráfico de Throughput (RPS) na rota DELETE



Fonte: Do Autor, 2025

Express (449.52 RPS). Essa *performance* superior sinaliza a robustez do *Fastify* em cenários que demandam alta agilidade para a manipulação de dados individuais.

6 Considerações finais

Este trabalho teve como objetivo principal realizar um *benchmark* de quatro *frameworks* do ecossistema *Node.js* — *Express*, *NestJS*, *Fastify* e *Next.js* — com foco no desempenho em operações *REST* típicas. Por meio de testes padronizados, foi possível extrair métricas claras e comparáveis sobre o tempo de resposta, taxa de requisições, percentis de latência e taxa de falhas, utilizando a ferramenta de testes de carga *k6*.

A metodologia adotada demonstrou-se eficaz para isolar o comportamento dos *frameworks* e oferecer uma visão precisa de seus pontos fortes e limitações sob diferentes níveis de carga. O uso direto de *SQL* e banco de dados local, sem intermediários como *ORMs*, permitiu avaliar o desempenho bruto das ferramentas.

Dentre os *frameworks* analisados, o *Fastify* se destacou de forma consistente em todos os testes, apresentando tempos de resposta mais baixos, maior *throughput* e estabilidade mesmo sob cargas mais elevadas. O *NestJS* também obteve bons resultados, equilibrando desempenho mesmo com sua arquitetura mais robusta, sendo uma opção sólida para projetos maiores e mais complexos. O *Express*, embora um pouco abaixo em alguns indicadores, manteve um desempenho aceitável e continua sendo uma escolha viável pela sua simplicidade.

O *Next.js*, por sua vez, apresentou desempenho significativamente inferior aos demais *frameworks* testados, especialmente em cenários de alta concorrência. Apesar disso, é importante ressaltar que o *Next* ainda está em processo de consolidação como ferramenta de *back-end*. Sua proposta de oferecer uma plataforma *full stack* integrada, unificando *front-end* e *back-end* em um único ambiente, é extremamente útil em projetos onde simplicidade e integração são prioridades, como *MVPs* (Arquitetura de *Model-View-Controller*)ou aplicações que não demandam muito volume. Com o avanço contínuo da plataforma, é esperado que melhorias de desempenho sejam implementadas ao longo do tempo, tornando-a ainda mais competitiva.

Assim, todos os *frameworks* têm seu espaço no desenvolvimento *web* moderno. A escolha entre eles deve levar em conta não apenas o desempenho, mas também fatores como experiência da equipe, complexidade do projeto, necessidade de organização arquitetural e velocidade de desenvolvimento. Este trabalho contribui para esse processo decisório, fornecendo dados objetivos que ajudam desenvolvedores e equipes técnicas a escolherem a tecnologia mais adequada às suas necessidades específicas.

Recomenda-se, em futuras pesquisas, aprofundar a análise considerando novos indicadores, como o consumo de memória e uso de *CPU*. Durante a realização dos testes, observou-se que o *Fastify*, embora tenha sido o *framework* com melhor desempenho, apresentou um consumo elevado de *CPU*, bem acima dos demais — o que merece ser investigado de forma mais detalhada. Além disso, seria interessante expandir os testes para ambientes mais amplos, como máquinas dedicadas ou plataformas em nuvem, para compreender o comportamento dos *frameworks* em cenários mais próximos da produção.

Benchmark of Node.js Frameworks for API Development: Performance Evaluation with Express, NestJS, Fastify, and Next.js

Pedro F. Poglia* Prof. Me. Jorge Luis Boeira Bavaresco †

2025

Abstract

The popularization of the Node.js environment for REST API development in the software industry has led to the emergence of several frameworks, expanding — and at the same time complicating — the selection process for developers. Aiming to assist in this decision-making, this work presents a benchmark, that is, a performance-focused comparative analysis of four of the most popular frameworks on the market used for API development. More specifically, the study focuses on three well-established frameworks in this area: Express, Fastify, and NestJS. In addition, it also examines a technology originally consolidated in the front-end market that is gradually transitioning to API and back-end development, becoming afull stack solution: Next.js. The adopted methodology included the implementation of four REST APIs, one for each framework, and the execution of load and stress tests simulating real users using k6, a well-known testing tool. The results obtained were analyzed and revealed that, although all frameworks deliver satisfactory performance, there are significant differences among them. Finally, practical recommendations are provided for selecting the most suitable framework based on the type of project or specific requirements.

Key-words: Node.js. Next. Nest. Fastify. Express. benchmark.

*Instituto Federal de Educação, Ciência e Tecnologia Sul-rio-grandense – IFSul, Bacharelado em Ciência da Computação, Passo Fundo – RS, Brasil. E-mail: pedro.poglia@hotmail.com

†Instituto Federal de Educação, Ciência e Tecnologia Sul-rio-grandense – IFSul E-mail: jorgebavaresco@ifsul.edu.br

Referências

DAHL, R. *Ryan Dahl - Disponível em: <https://en.wikipedia.org/wiki/Ryan_Dahl>. Acesso em: 27 Nov. 2024.* 2009. Citado na página 2.

FASTIFY. *Fastify - The Fast and Low Overhead Web Framework for Node.js - Disponível em: <https://fastify.dev/>. Acesso em: 27 Nov. 2024.* 2024. Citado na página 5.

HAHN, E. M. *Express in Action: Writing, Building, and Testing Node.js Applications*. Shelter Island, NY: Manning Publications, 2016. Disponível em: <https://books.google.com.br/books?id=czkzEAAAQBAJ>. Acesso em: 29 Nov. 2024. Citado na página 5.

KUFFEL, P.; WALTER, B. Performance of node.js backend application frameworks: An empirical evaluation. In: *Proceedings of the 32nd International Conference on Information Systems Development (ISD2024)*. Poznań University of Technology, 2024. Disponível em: <https://aisel.aisnet.org/isd2014/proceedings2024/managingdevops/7>. Acesso em: 29 Nov. 2024. Citado na página 2.

LABS, G. *k6 - Load testing for engineering teams - Disponível em: <https://k6.io/>. Acesso em: 24 Jun. 2025.* 2025. Citado na página 7.

MARDAN, A. *Pro express. js: Master express. js: The node. js framework for your web development*. [S.l.]: Springer, 2014. Citado na página 5.

MDN, M. D. N. *JavaScript - MDN Web Docs - Disponível em: <https://developer.mozilla.org/pt-BR/docs/Web/JavaScript>. Acesso em: 27 Nov. 2024.* 2024. Citado na página 4.

NESTJS, N. O. F. *NestJS - A progressive Node.js framework - Disponível em: <https://nestjs.com/>. Acesso em: 27 Nov. 2024.* 2024. Citado na página 6.

NODE.JS, F. *Node.js - Disponível em: <https://nodejs.org/pt>. Acesso em: 27 Nov. 2024.* 2024. Citado na página 4.

PEARCE, T. M.; NIKIFOROVA, M. N.; ROY, S. Interactive browser-based genomics data visualization tools for translational and clinical laboratory applications. *The Journal of Molecular Diagnostics*, v. 21, n. 6, p. 985–993, 2019. ISSN 1525-1578. Disponível em: <https://www.sciencedirect.com/science/article/pii/S1525157819303484>. Citado na página 4.

V8, V. J. E. T. *V8 Documentation - Disponível em: <https://v8.dev/docs>. Acesso em: 29 Nov. 2024.* 2024. Citado na página 2.

VERCEL, V. I. *Next.js - The React Framework - Disponível em: <https://nextjs.org/>. Acesso em: 27 Nov. 2024.* 2024. Citado na página 6.