Análise de Desempenho de APIs: Um Estudo Comparativo entre Node.js, Java e Python

Tom Jones Silva Gomes Ramos
Universidade Federal Rural de Pernambuco

RESUMO

O crescente uso de APIs REST em aplicações web e corporativas torna o desempenho dessas interfaces um fator essencial para a experiência do usuário e para a eficiência dos sistemas. A escolha da linguagem de programação e do framework influencia diretamente métricas como o tempo de resposta, a utilização de recursos e a escalabilidade. Diante disso, este trabalho tem como objetivo analisar o desempenho de três APIs REST desenvolvidas em Node.js, Java e Python, bem como estruturar um processo sistemático para a realização de testes de desempenho de APIs, buscando compreender como cada tecnologia se comporta sob diferentes tipos de carga e cenários operacionais. Foram avaliadas métricas de tempo de resposta, vazão, utilização de CPU e memória em quatro cenários distintos, abrangendo operações de escrita, leitura, geração de relatórios e cálculos intensivos de CPU. As APIs foram implementadas com os frameworks NestJS, Spring Boot e FastAPI e testadas em ambiente controlado com o Apache JMeter. Os resultados obtidos permitem observar as vantagens e limitações de cada linguagem em diferentes contextos, oferecendo subsídios técnicos para desenvolvedores e gestores de tecnologia na escolha da linguagem mais adequada a projetos com diferentes demandas de desempenho.

Palavras-chave: Avaliação de desempenho, Node.js, Java, Python, comparação de linguagens de programação.

1 Introdução

Com o crescimento das aplicações web e móveis, as APIs (Application Programming Interfaces) tornaram-se componentes essenciais para a integração e comunicação entre sistemas. Nesse contexto, a linguagem de programação escolhida para o desenvolvimento dessas interfaces influencia diretamente o desempenho, a escalabilidade e a utilização de recursos computacionais (IVANOV, 2022). Uma escolha adequada pode otimizar o tempo de resposta ao usuário e reduzir custos operacionais, especialmente em aplicações de alta demanda.

No desenvolvimento back-end, Node.js, Java e Python destacam-se por sua popularidade, suporte comunitário e diversidade de frameworks. Cada uma apresenta características distintas: o Node.js é reconhecido por sua arquitetura assíncrona orientada a eventos, que proporciona alta escalabilidade em tarefas de I/O intensivas (CRESTANI, 2024); o Java, pelo desempenho robusto da JVM e seu ecossistema corporativo consolidado (CAVALCANTE, 2022); e o Python, pela simplicidade e produtividade, sobretudo em projetos voltados à análise de dados e prototipagem rápida (ZIMMER, 2023).

A linguagem de programação impacta não apenas o tempo de execução, mas também o consumo energético, a utilização de memória e a complexidade do desenvolvimento (TÜRKMEN et al., 2024; BEIERLEIB et al., 2023). Diversos estudos comparativos já analisaram o desempenho de linguagens em contextos como o Big Data (CAVALCANTE, 2022), o acesso a bancos de dados (ZIMMER, 2023) e o processamento paralelo (SOUZA et al., 2023; COSTANZO et al., 2021). Contudo, ainda são escassas as análises experimentais que avaliam APIs REST em condições controladas e equivalentes.

As APIs REST (Representational State Transfer) são interfaces baseadas no protocolo HTTP, amplamente utilizadas por sua simplicidade e escalabilidade. Elas seguem o modelo cliente-servidor e utilizam métodos padronizados para comunicação entre sistemas, sendo fundamentais para a integração de aplicações modernas.

Diante desse cenário, este trabalho propõe uma análise comparativa do desempenho de APIs desenvolvidas em Node.js, Java e Python, considerando métricas como tempo de resposta, vazão, utilização de CPU e memória. Como contribuição, espera-se apresentar os resultados da avaliação de desempenho em diferentes cenários de APIs desenvolvidas em Node.js, Java e Python , para auxiliar desenvolvedores e pesquisadores na escolha da linguagem mais adequada a diferentes contextos de desenvolvimento de APIs.

.

2. Trabalhos Relacionados

A escolha da linguagem de programação para o desenvolvimento de APIs é um fator determinante para o desempenho, a escalabilidade e a eficiência dos sistemas. Embora existam diversos estudos de benchmark entre linguagens, muitos desenvolvedores ainda selecionam tecnologias com base na familiaridade, sem considerar métricas objetivas de desempenho. Assim, pesquisas comparativas são essenciais para compreender o comportamento das linguagens em contextos reais, especialmente na implementação de APIs REST sob diferentes níveis de carga e de concorrência.

Diversos estudos têm investigado o desempenho de linguagens sob múltiplas perspectivas, abrangendo desde aplicações de Big Data e eficiência energética até tarefas de processamento paralelo e operações intensivas de CPU e de I/O. Esses trabalhos contribuem significativamente para o entendimento do impacto das linguagens e frameworks no desempenho computacional, mas diferem quanto a objetivos, metodologias e ambientes experimentais — o que reforça a importância de análises voltadas a cenários específicos, como o proposto neste estudo.

Cavalcante (2022) comparou Java e Scala em aplicações de Big Data e observou melhor desempenho do Scala em grandes volumes de dados, embora o uso do Apache Spark nem sempre resultasse em ganhos expressivos devido à sobrecarga de configuração. Zimmer (2023) analisou C e Python em operações com MySQL e verificou que, com o uso adequado de transações e drivers, o Python alcançou desempenho próximo ao de C, o que mostra que sua natureza interpretada não necessariamente implica ineficiência.

Ivanov (2022) comparou Rust e C++, constatando que Rust foi superior no algoritmo *Merge Sort*, enquanto C++ teve melhor desempenho no *Insertion Sort* e em operações de dicionário. De forma semelhante, Costanzo et al. (2021) demonstraram que Rust atinge desempenho próximo ao do C em problemas *N-Body*, especialmente em arquiteturas multicore, destacando sua facilidade de paralelização e menor esforço de programação.

Türkmen et al. (2024) avaliaram Python, R, Java e Julia em regressão linear com 1 milhão de registros, apontando o Python como o mais rápido e conciso, seguido por Java e Julia. Já Crestani (2024) comparou C++, Go, Rust, Python e JavaScript, verificando que Rust foi o mais rápido em quatro dos sete testes, enquanto Python apresentou o pior desempenho, chegando a ser até 40 vezes mais lento do que C++.

Beierleib et ~~al.~~ (2023) analisaram Python, R e Rust em tarefas de agregação de dados de 65 GB e observaram que Rust foi substancialmente mais rápido, enquanto R foi o mais lento. Dymora e Paszkiewicz (2020) compararam Go, Java e Python em algoritmos de árvores de decisão e constataram que Python foi mais eficiente em grandes volumes de dados, Java se destacou em bases menores e Go apresentou menor utilização de memória.

Pereira et al. (2021) exploraram a eficiência energética de 27 linguagens e concluíram que linguagens compiladas, como C, Rust e C++, são mais eficientes em termos de tempo de resposta e de energia do que linguagens interpretadas, como Python e Ruby. Souza et al. (2023) também compararam Go, Java, C# e Python em algoritmos paralelos, identificando C# como a linguagem mais rápida e Go como a de menor utilização de CPU e de memória.

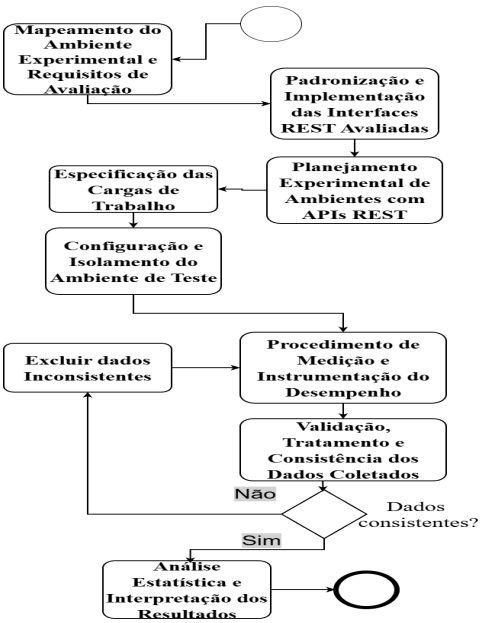
Apesar desses avanços, ainda há escassez de estudos experimentais voltados especificamente a APIs REST, em ambientes equivalentes e com condições controladas. Di Meglio e Starace (2024) analisaram frameworks REST em diferentes linguagens, avaliando o tempo de resposta, a utilização de CPU, a utilização de memória e o consumo de energia, mas concentraram-se em cenários genéricos de carga de trabalho.

Diferentemente desses trabalhos, esta pesquisa propõe uma análise comparativa direcionada entre Node.js, Java e Python, avaliando seu comportamento sob diferentes tipos de carga de trabalho, abrangendo tanto tarefas I/O-bound (intensivas em entrada e saída) quanto CPU-bound (processamento intensivo). Essa abordagem busca preencher uma lacuna na literatura, oferecendo uma avaliação de desempenho de APIs REST, contribuindo para decisões quanto à escolha da linguagem de programação e ao equilíbrio entre produtividade, familiaridade e eficiência.

3 Metodologia para Avaliação de Desempenho de APIs REST

A Metodologia de Avaliação de Desempenho de APIs REST proposta neste trabalho foi estruturada em etapas que orientam a condução de experimentos de desempenho. Essa metodologia define um conjunto de atividades como Mapeamento do Ambiente Experimental e Requisitos de Avaliação; Padronização e Implementação das Interfaces REST Avaliadas; Planejamento Experimental de Ambientes com APIs REST; Especificação das Cargas de Trabalho; Configuração e Isolamento do Ambiente de Teste; Procedimento de Medição e Instrumentação do Desempenho; Validação, Tratamento e Consistência dos Dados Coletados; Análise Estatística e Interpretação dos Resultados. Essas atividades podem ser replicadas em diferentes estudos, independentemente dos valores específicos de carga, ferramentas ou ambientes utilizados. O objetivo é garantir rigor, reprodutibilidade e comparabilidade entre estudos que avaliam o desempenho de APIs desenvolvidas em diferentes tecnologias. A sequência dessas atividades é ilustrada na Figura 1.

Figura 1 - Metodologia para Avaliação de Desempenho de APIs REST



3.1 Mapeamento do Ambiente Experimental e Requisitos de Avaliação

A primeira atividade consiste na compreensão detalhada do ambiente onde os experimentos serão executados. Nessa fase, devem ser identificados e descritos os recursos de hardware, software, rede, sistema operacional e ferramentas disponíveis. Também são definidos os parâmetros gerais do ambiente de execução das APIs, como versão das tecnologias utilizadas, arquitetura da aplicação, dependências necessárias e configurações básicas que assegurem condições controladas e equivalentes entre as execuções. Além disso, esta atividade estabelece os objetivos do estudo e as métricas de desempenho que serão avaliadas, garantindo que todo o experimento seja planejado de forma coerente com o ambiente escolhido.

3.2 Padronização e Implementação das Interfaces REST Avaliadas

Nesta etapa, é realizada a implementação das APIs que serão avaliadas no estudo, garantindo que todas apresentem funcionalidades, rotas e estruturas equivalentes, independentemente da linguagem ou framework adotado. O objetivo é padronizar o comportamento funcional das interfaces, assegurando que diferenças observadas nos resultados de desempenho sejam atribuídas exclusivamente às tecnologias avaliadas, e não a variações de implementação. Assim, a metodologia pode ser aplicada a qualquer conjunto de APIs, desde que sejam desenvolvidas seguindo o mesmo conjunto de requisitos funcionais e estruturais definidos previamente.

3.3 Planejamento Experimental de Ambientes com APIs REST

O planejamento do experimento define as variáveis que serão observadas, as métricas de desempenho que serão coletadas e o conjunto de fatores e níveis que compõem o estudo. Nesta atividade, identificam-se fatores como linguagem ou framework utilizado, tipo de operação da API (por exemplo, leitura, escrita ou processamento), intensidade da carga aplicada e configuração do ambiente de execução. Para cada fator, são estabelecidos níveis possíveis, como diferentes linguagens, diferentes volumes de requisições simultâneas ou diferentes cenários de uso. Também são definidas as métricas a serem medidas, incluindo tempo médio de resposta, vazão (throughput), taxa de erros, utilização de CPU, utilização de memória. Além disso, especificam-se os tipos de carga a serem aplicados, a forma de distribuição das requisições, o número de replicações necessárias e os critérios de controle que garantem consistência e reprodutibilidade aos resultados. Os valores específicos adotados para cada nível podem variar conforme o ambiente ou interesse do pesquisador, desde que a estrutura metodológica seja mantida.

3.4 Especificação das Cargas de Trabalho

A Especificação das Cargas de Trabalho envolve estabelecer como o sistema será exercitado durante os experimentos, considerando diferentes intensidades e tipos de requisições. Nesta atividade, o pesquisador deve selecionar quais operações serão avaliadas, como leitura, escrita ou processamento intensivo e definir níveis progressivos de carga que permitam observar o comportamento da API em cenários variados de demanda.

Além disso, esta atividade contempla a geração da carga de trabalho, que consiste em configurar a ferramenta escolhida para distribuir as requisições ao sistema conforme os níveis definidos. Diversas ferramentas podem ser adotadas para esse propósito, como Apache JMeter, Locust, k6, Gatling ou Artillery, entre

outras, permitindo a simulação de usuários virtuais, definição da taxa de envio de requisições, cenários de estresse e picos de carga. A configuração dessas ferramentas deve especificar parâmetros como quantidade de usuários virtuais, frequência das requisições, tempo de execução e eventuais variações de cenário, garantindo que os testes representem adequadamente o perfil de carga pretendido.

3.5 Configuração e Isolamento do Ambiente de Teste

Esta atividade consiste na preparação e padronização do ambiente necessário para a execução dos experimentos. Nessa etapa, o pesquisador deve definir quais recursos de hardware e software serão utilizados, configurar as ferramentas de geração de carga e de monitoramento de métricas. A metodologia orienta que o ambiente seja isolado de processos externos, que suas dependências sejam documentadas e que os procedimentos de controle como inicialização, reinicialização entre testes e verificação de integridade, sejam seguidos de forma consistente. Os detalhes técnicos específicos, como modelo de hardware, sistema operacional, ferramentas utilizadas ou eventuais limitações da infraestrutura, podem variar conforme o estudo, desde que o pesquisador mantenha critérios claros de controle e reprodutibilidade.

3.6 Procedimento de Medição e Instrumentação do Desempenho

A medição de desempenho compreende a execução dos testes e a coleta sistemática das métricas definidas previamente no planejamento experimental. Nesta atividade, estabelecem-se o intervalo de medição, a taxa de amostragem, os procedimentos de verificação de logs e o monitoramento contínuo dos recursos utilizados, garantindo o registro consistente dos dados. Também é realizada a validação preliminar dos resultados, de modo a identificar e descartar eventuais anomalias, falhas de execução ou variações atípicas que comprometam a confiabilidade do experimento. Para essa etapa, podem ser utilizadas ferramentas de monitoramento de recursos, como Server Agent, Prometheus, Grafana, htop ou Perf, complementando as ferramentas de geração de carga.

3.7 Validação, Tratamento e Consistência dos Dados Coletados

A etapa de Validação, Tratamento e Consistência dos Dados Coletados tem como objetivo assegurar que os resultados obtidos durante a execução dos testes sejam confiáveis, representativos e adequados para análise estatística. Inicialmente, são verificados a integridade e o formato dos arquivos gerados pelas ferramentas de medição, incluindo logs de requisições, registros de uso de recursos e relatórios de desempenho. Essa verificação busca identificar falhas de execução, interrupções inesperadas, erros de rede ou inconsistências que possam comprometer a interpretação dos resultados.

Em seguida, procede-se ao tratamento preliminar dos dados, que pode envolver a remoção de execuções inválidas, filtragem de valores ausentes ou corrompidos, e identificação de outliers resultantes de oscilações anormais do ambiente de teste. Os critérios de exclusão devem ser previamente definidos no planejamento experimental, de modo a evitar vieses na seleção dos dados. Também é avaliada a consistência interna dos conjuntos de métricas, comparando resultados entre replicações para verificar estabilidade e reprodutibilidade das medições.

Por fim, os dados validados são organizados em planilhas ou estruturas padronizadas, assegurando uniformidade no formato e facilitando sua posterior análise estatística. O resultado desta etapa é um conjunto de dados consistentes e

prontos para serem utilizados na etapa de interpretação e comparação dos resultados experimentais.

3.8 Análise Estatística e Interpretação dos Resultados

A análise estatística consiste na aplicação de métodos quantitativos que permitam interpretar os resultados obtidos. Esta atividade envolve o cálculo de medidas como médias, desvios padrão e intervalos de confiança, além da identificação de tendências e padrões de comportamento entre as métricas coletadas. As técnicas utilizadas podem variar de acordo com os objetivos do estudo, mas devem ser definidas previamente no planejamento experimental, garantindo que os dados sejam analisados de forma rigorosa e sistemática.

4 Estudo de Caso

O estudo de caso tem como objetivo avaliar a metodologia proposta para comparação do desempenho de três APIs REST desenvolvidas nas linguagens Node.js, Java e Python, utilizando respectivamente os frameworks NestJS, Spring Boot e FastAPI. O propósito é identificar como cada linguagem e seu ecossistema respondem sob diferentes tipos de carga como I/O-bound (intensivas em entrada e saída de dados) e CPU-bound (intensivas em processamento).

A análise busca oferecer evidências práticas que auxiliem desenvolvedores e profissionais de tecnologia na tomada de decisão sobre a linguagem mais adequada a aplicações com diferentes perfis de processamento, considerando tempo de resposta, vazão (throughput) e uso de CPU e memória.

4.1 Mapeamento do Ambiente Experimental e Requisitos de Avaliação

Os testes foram realizados em um ambiente controlado, cujo inventário foi registrado conforme a atividade de mapeamento. O equipamento utilizado consistiu em um notebook com processador Intel Celeron N4000 (1,10 GHz), 4 GB de RAM (2133 MHz) e armazenamento em disco 64GB(SSD). O sistema operacional empregado foi Windows 11 Home Single Language. O Sistema Gerenciador de Banco de Dados utilizado foi o MySQL 8.1, e as APIs foram executadas nas seguintes versões de ambiente: Node.js 22.19.0 (com NestJS 11.1.3), Java 24 (com Spring Boot 3.5.3) e Python 3.13.5 (com FastAPI 0.118.0). Foram definidos como requisitos de avaliação as métricas de interesse tempo médio de resposta, throughput (req/s), taxa de erros, utilização de CPU, utilização de memória e critérios de aceitação e repetibilidade (30 repetições por cenário) para garantir robustez nas medições.

4.2 Padronização e Implementação das Interfaces REST Avaliadas

As interfaces REST avaliadas foram padronizadas de modo a garantir equivalência funcional e permitir a comparação entre as diferentes tecnologias. Para isso, cada implementação, independente do framework ou linguagem utilizada, seguiu o conjunto de operações: (i) escrita de dados, (ii) leitura de dados, (iii) cálculo de juros compostos e (iv) geração de relatório. Cada conjunto de operações foi projetado para executar a mesma lógica de negócio em todas as APIs, mantendo contratos de entrada e saída consistentes, além de estruturas de dados e operações equivalentes no banco de dados. As decisões referentes a rotas, modelos de

payload, uso do SGBD e tratamento de erros foram mantidas uniformes entre as aplicações, assegurando que diferenças observadas nos resultados de desempenho reflitam exclusivamente as tecnologias avaliadas, e não variações de implementação.A caracterização dos tipos de carga utilizados nos experimentos, bem como seus respectivos cenários de teste e descrições técnicas, encontra-se detalhada na Tabela 1.

Tabela 1 - Tabela de operações

Tipo de Carga	Cenário	Descrição Técnica
Leitura de dados (I/O-bound)	**API de consulta de produtos**	Um endpoint que faz consultas simples de produtos ao banco de dados. GET /produtos
Escrita de dados (I/O-bound)	**Cadastro de pedidos**	Inserção de registros com dados recebidos no corpo da requisição.POST /produtos
Processamento matemático (CPU-bound)	**Cálculo de métricas financeiras com juros-sobre-juros**	Simulação de Retorno de Investimento com Reinvestimento Mensal (Juros sobre Juros).POST /juros-compostos
Processamento de lote I/O-bound + CPU-bound:	**Geração de relatórios**	Endpoint que percorre milhares de registros de vendas e produtos, agrupa os dados por mês e categoria, soma o total vendido por categoria em cada mês e faz cálculos de projeção futura.GET /relatorio

4.3 Planejamento Experimental de Ambientes com APIs REST

O planejamento experimental seguiu a estrutura definida na metodologia, considerando exclusivamente os elementos utilizados neste estudo. Foram definidos três fatores principais: (i) a tecnologia empregada na implementação da API (NestJS/Node.js, Spring Boot/Java e FastAPI/Python), (ii) o tipo de operação avaliada (escrita de dados, leitura de dados, cálculo de juros compostos e geração de relatório) e (iii) a intensidade de carga aplicada, com 200, 500 e 1000 requisições por execução. Os fatores e seus respectivos níveis utilizados no experimento estão detalhados na Tabela 2, que apresenta a estrutura organizada das combinações consideradas no planejamento experimental.

As métricas coletadas foram: tempo médio de resposta, vazão (throughput), taxa de erros, utilização de CPU e utilização de memória. Para garantir estabilidade estatística, cada combinação entre tecnologia, cenário e nível de carga foi executada 30 vezes, permitindo a análise de variabilidade e a identificação de padrões de desempenho consistentes.

Tabela 2 - Planejamento de experimento

Fator	Níveis
Linguagem/API	Node.js (NestJS), Java (Spring Boot), Python (FastAPI)
Tipo de Carga	I/O-bound, CPU-bound, I/O-bound + CPU-bound
Volume de Carga	Baixa (200 req/s), Média (500 req/s), Alta (1000+ req/s)

4.4 Especificação das Cargas de Trabalho

As cargas de trabalho foram modeladas no Apache JMeter utilizando scripts desenvolvidos especificamente para os quatro cenários definidos neste estudo: (1) escrita de dados, (2) leitura de dados, (3) cálculo de juros compostos e (4) geração de relatório. Cada script reflete o comportamento real esperado de cada operação, incluindo seus respectivos payloads, parâmetros de entrada e estrutura de resposta.

Para todos os cenários, foram geradas cargas com 200, 500 e 1000 requisições, distribuídas conforme a configuração de threads definida no JMeter. A modelagem estabeleceu parâmetros de execução como número de usuários virtuais, ramp-up e número total de iterações, mantendo a mesma estrutura de carga para todas as linguagens avaliadas. Essa padronização permitiu que as comparações de desempenho refletissem o impacto das tecnologias e não diferenças de configuração da ferramenta de testes.

4.5 Configuração e Isolamento do Ambiente de Teste

A execução dos experimentos foi realizada no mesmo equipamento que hospedou tanto as APIs quanto a ferramenta de geração de carga, condição explicitamente registrada como limitação experimental. Para minimizar interferências entre execuções, foram adotados procedimentos de isolamento que incluíram: encerramento de processos não essenciais, reinicialização das APIs entre cenários, reconexão ao banco de dados quando necessário e um período de estabilização antes de iniciar novas rodadas de testes.

Para cada conjunto de execuções, foi aplicado um intervalo entre repetições de 30 segundos para reduzir o impacto de possíveis oscilações do sistema operacional. Além disso, os serviços do banco de dados permaneceram ativos e estáveis durante todo o experimento, garantindo que os resultados refletissem apenas as diferenças das tecnologias avaliadas. Toda a configuração do ambiente foi mantida constante ao longo dos testes, assegurando consistência entre as medições.

4.6 Procedimento de Medição e Instrumentação do Desempenho

A instrumentação combinou a coleta de métricas geradas pelo Apache JMeter (latência, throughput, erros) com monitoramento de recursos via Server Agent para CPU e memória; Foi utilizado a extensão do Perfmon para o Jmeter para coleta dos dados do Server Agent. Foram definidos intervalos de amostragem e agregação consistentes com amostragem de métricas de recursos a cada 1 segundo e estabelecido um procedimento de registro centralizado em planilhas CSV. Os logs de execução do JMeter, os relatórios do Server Agent e os logs de aplicação foram preservados integralmente para permitir auditoria e análises posteriores. Antes da consolidação, verificou-se a ausência de falhas de execução (timeouts, erros HTTP) que comprometesse uma execução.

4.7 Validação, Tratamento e Consistência dos Dados Coletados

Os dados coletados foram submetidos a um processo sistemático de validação e tratamento antes da análise estatística. Inicialmente, validou-se a integridade dos arquivos gerados (logs JMeter e exportes CSV do monitoramento) e verificou-se a ocorrência de falhas de execução que justificassem exclusão. Critérios de exclusão (definidos no planejamento) foram aplicados, por exemplo: execuções com >5% de erros HTTP, interrupções do sistema ou amostragens incompletas. Em seguida, foram identificados e documentados outliers por meio de análises exploratórias (boxplots e inspeção de percentis); outliers explicáveis por falhas externas foram excluídos, enquanto variações no tempo de resposta foram mantidas.

4.8 Análise Estatística e Interpretação dos Resultados

A análise dos dados consistiu na consolidação e interpretação estatística das métricas coletadas ao longo das 30 execuções realizadas para cada combinação de tecnologia, cenário e nível de carga. Para cada métrica, tempo médio de resposta, vazão, utilização de CPU e utilização de memória, foram calculadas a média das execuções e o desvio padrão.

5 Resultados

O experimento foi dividido em quatro cenários distintos, Escrita de dados, Leitura de dados, Cálculo de Juros Compostos e Relatório, cada um representando um tipo de operação comum em sistemas baseados em APIs REST.

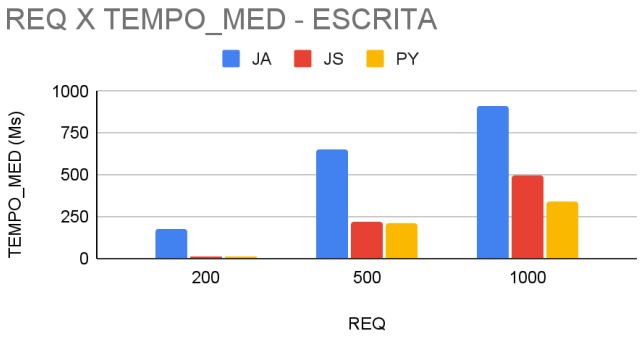
5.1 Cenário Escrita de Dados

O endpoint de escrita recebeu requisições para inserir registros de produtos em banco de dados, simulando operações de cadastro em sistemas de e-commerce. Os testes variaram de 200, 500 e 1000 requisições simultâneas, avaliando a escalabilidade das linguagens.

Nos testes de escrita, Python (FastAPI) apresentou os menores tempos médios de resposta, variando entre 8,36 ms (200 requisições) e 336,86 ms (1000 requisições), o que demonstra alta eficiência na inserção de dados sob cargas moderadas.

O Node.js (NestJS) manteve desempenho consistente, mas com tempos ligeiramente maiores, entre 14,72 ms e 496,53 ms, enquanto Java (Spring Boot) apresentou maiores latências, chegando a 915,26 ms em 1000 requisições. A relação entre o número de requisições e o tempo médio para cada tecnologia pode ser visualizada na Figura 2.

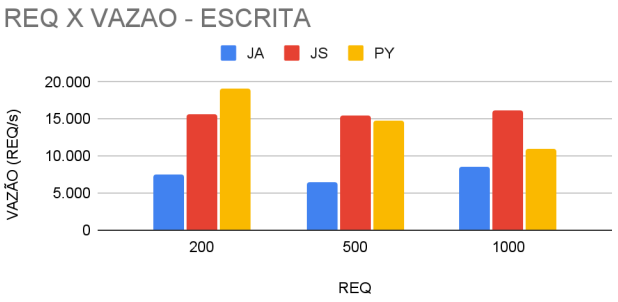
Figura 2 - Gráfico requisição x tempo médio - escrita.



Em relação à vazão, o FastAPI (Python) iniciou o teste com a maior taxa de requisições processadas por segundo com 19000 req/s, mas teve queda gradual conforme o aumento da carga indo para 10900 req/s, indicando perda de desempenho sob estresse. Já o Node.js (NestJS) manteve vazão com pouca variação de 15600 a 16000 req/s ao longo de todos os níveis de requisição, evidenciando melhor escalabilidade. O Spring Boot (Java) apresentou as menores taxas de throughput variando de 6400 a 8500 req/s, compatíveis com seus tempos de resposta mais elevados.

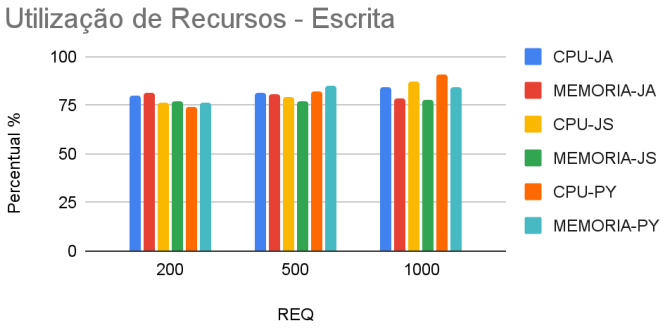
Esse comportamento indica que Python é altamente eficiente em cargas leves e moderadas, mas Node.js mantém desempenho mais consistente em cenários de alta concorrência. Essa tendência é ilustrada na Figura 3.

Figura 3 - Gráfico requisição x vazão - escrita.



A utilização de CPU e memória foram parecidas entre as linguagens, embora o Java tenha apresentado leve sobrecarga de CPU nas cargas mais altas. Uma representação visual desses resultados é mostrada na Figura 4.

Figura 4 - Gráfico utilização de recursos- escrita.

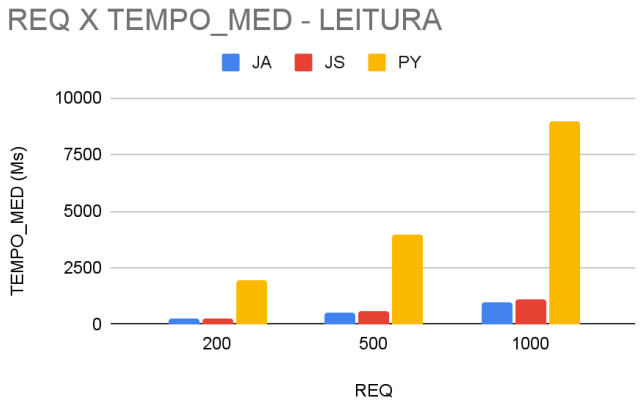


5.2 Cenário Leitura de Dados

Este cenário avaliou consultas simultâneas a 500 registros previamente inseridos, medindo tempo de resposta e vazão sob diferentes níveis de carga.
Nas operações de leitura o Node.js (NestJS) apresentou o melhor desempenho sob cargas leves, com tempo médio de 268,2 ms em 200 requisições, seguido de Java (Spring Boot) com valor próximo, com 271,04 ms. À medida que a carga aumentou, Java manteve-se mais eficiente, registrando 499,33 ms em 500 requisições e 981 ms em 1000, enquanto o NestJS atingiu 554,9 ms e 1.126,67 ms nas mesmas requisições.

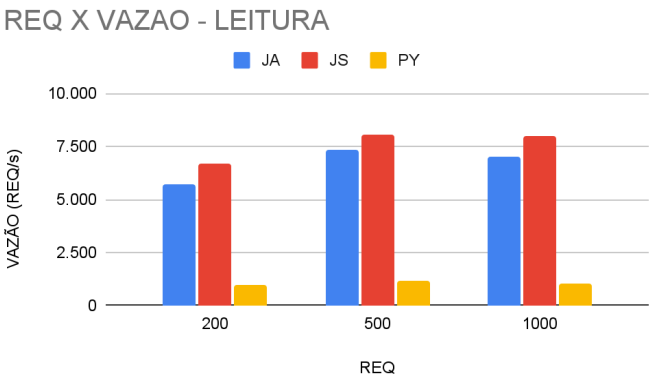
O Python (FastAPI) apresentou o pior desempenho geral, com tempos médios de 1.921 ms, 3.959,8 ms e 8.982,8 ms, evidenciando significativa degradação de tempo de resposta sob maior concorrência. A Figura 5 evidencia o comportamento do tempo médio frente ao aumento das requisições.

Figura 5 - Gráfico requisição x tempo médio - leitura.



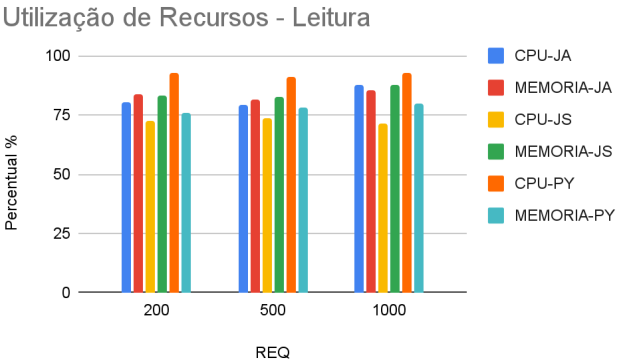
Em relação à vazão o Node.js e o Java mantiveram desempenho com valores entre 6500 e 8000 req/s, enquanto o FastAPI apresentou 1000 req/s nas cargas mais altas. Esse comportamento está sintetizado graficamente na Figura 6.

Figura 6 - Gráfico requisição x vazão -leitura.



Na utilização de recursos, o Node.js demonstrou a menor utilização de CPU, enquanto Python apresentou valores mais elevados, próximos de 93%, com a utilização de memória levemente superior, possivelmente em razão do gerenciamento dinâmico do interpretador. Uma visão comparativa desses resultados encontra-se na Figura 7.

Figura 7 - Gráfico utilização de recursos - leitura.

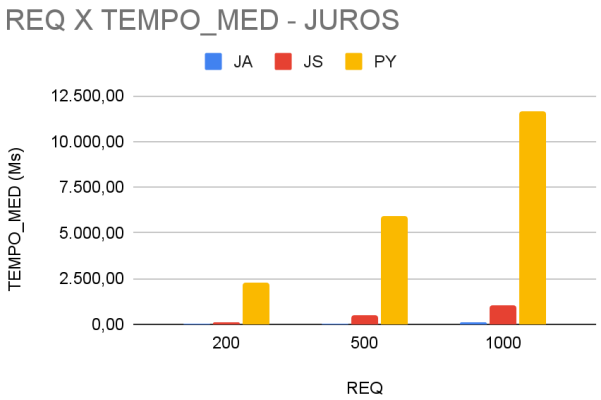


5.3 Cenário Cálculo de Juros Compostos

Este cenário simulou operações intensivas de CPU, com 10.000 iterações por requisição, avaliando a eficiência em tarefas de alto processamento.

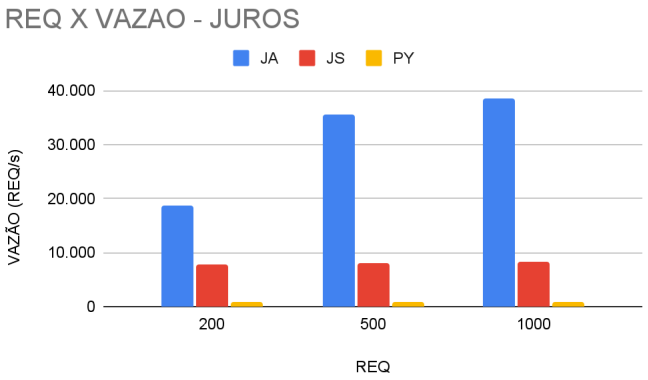
No cálculo de juros compostos, o Java (Spring Boot) apresentou o melhor desempenho em todas as cargas, com tempos médios de 13,9 ms em 200 requisições, 41,93 ms em 500 e 139,26 ms em 1000 requisições, evidenciando a eficiência do compilador JIT da JVM para operações numéricas intensivas. O Node.js (NestJS) apresentou tempo de resposta variando de 148,2 ms a 1.011,3 ms, enquanto Python (FastAPI) apresentou o pior tempo de resposta, variando de 2.286,6 ms para 11.690,3 ms conforme o aumento da carga. A Figura 8 reforça graficamente a diferença entre os tempos médios obtidos.

Figura 8 - Gráfico requisição x tempo médio - juros.



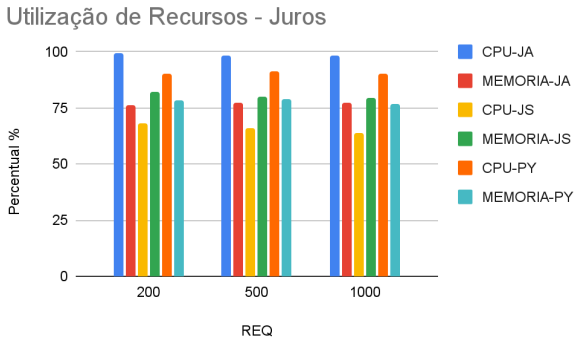
O Java apresentou a vazão de 38500 req/s, seguido de Node.js, que manteve desempenho entre 7800 e 8200 req/s. O Python apresentou baixa vazão, próxima de 800 req/s, confirmando sua limitação em cenários de alto processamento. Esse padrão de desempenho fica mais claro ao observar a Figura 9.

Figura 9 - Gráfico requisição x vazão - juros.



A utilização de CPU foi elevada para todas as linguagens, com Java utilizando cerca de 98 % de forma estável, já o Python oscilando entre 90 e 91 %, e o Node.js tem valores de utilização menores de 65 %. A utilização de memória manteve-se equilibrado entre 75% a 80%, com pequenas variações e o Java apresentando a menor utilização média ficando com 75% de utilização. Essa tendência é ilustrada graficamente na Figura 10.

Figura 10 - Gráfico utilização de recursos- juros.

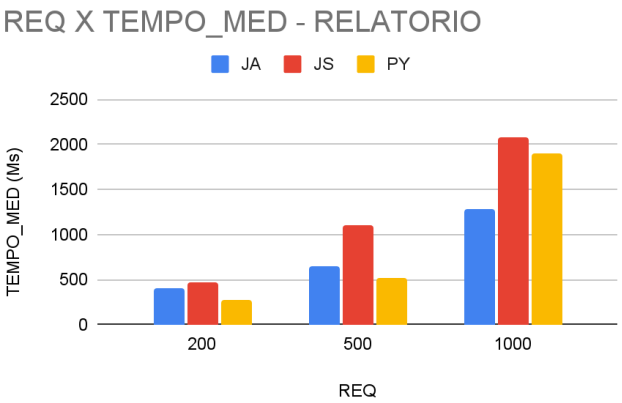


5.4 Cenário Geração de Relatório

Esse cenário combinou leitura e agregação, representando uma carga mista (I/O e CPU).

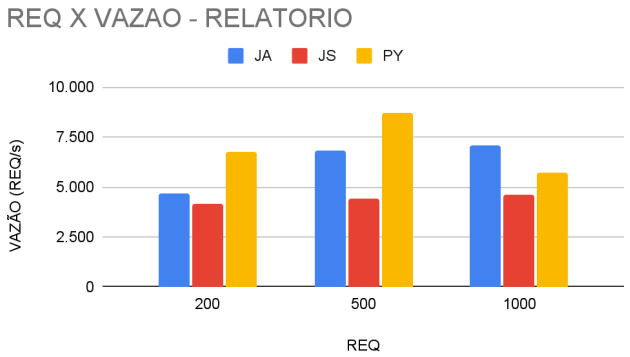
Na geração de relatórios, o Python (FastAPI) apresentou o melhor desempenho geral, com tempos médios de 267,85 ms para 200 requisições, 519,6 ms para 500 e 1.898,67 ms para 1000 requisições, demonstrando eficiência tanto em operações de leitura quanto operações de agregações. Já o Java (Spring Boot) apresentou tempos entre 413,5 ms e 1.286,2 ms. O Node.js (NestJS) apresentou desempenho inferior nesse cenário, com tempo de resposta mais altos (466,53 ms a 2.085,86 ms), possivelmente devido à sobrecarga em operações mistas de I/O e CPU. A Figura 11 mostra de forma gráfica como o tempo médio se comporta diante das requisições.

Figura 11 - Gráfico requisição x tempo médio - relatório.



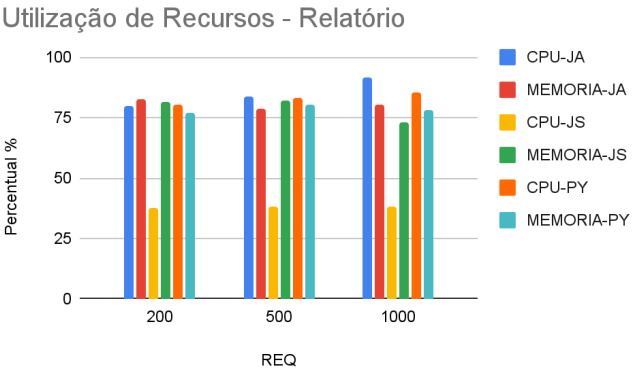
Em relação à vazão, o FastAPI destacou-se sob cargas moderadas, alcançando aproximadamente 8700 req/s, seguido de Java (≈7100 req/s) e o Node.js (≈4600 req/s). Uma representação visual desses resultados é mostrada na Figura 12.

Figura 12 - Gráfico requisição x vazão - relatório.



A utilização de CPU foi mais baixa no Node.js (≈38 %), enquanto Java e Python operaram entre 80 e 85 %, comportamento esperado em tarefas com alto processamento.
A utilização de memória manteve-se equilibrada entre as três linguagens ficando em torno de 78% de utilização, sem variações significativas. Os valores obtidos podem ser analisados visualmente por meio da Figura 13.

Figura 13 - Gráfico utilização de recursos - relatório.



6 Análise dos Resultados

De forma geral, os resultados demonstram que cada linguagem apresentou vantagens específicas conforme o tipo de carga e a natureza das operações. O Node.js (NestJS) destacou-se em cenários I/O-bound, oferecendo tempos de resposta baixos e boa estabilidade sob alta concorrência, o que o torna ideal para aplicações que demandam alta taxa de requisições simultâneas e comunicação intensiva com o banco de dados.

O Java (Spring Boot) obteve o melhor desempenho em operações CPU-bound, evidenciando eficiência em cálculos e tarefas computacionalmente pesadas, com alta consistência e aproveitamento dos recursos de processamento da JVM. Esse comportamento reforça sua adequação para sistemas corporativos e aplicações que exigem grande capacidade de processamento.

O Python (FastAPI) demonstrou excelente desempenho em cargas leves e moderadas, especialmente em escrita e geração de relatórios, porém apresentou limitações de escalabilidade sob altas cargas, reflexo de seu modelo interpretado e do custo de gerenciamento de recursos.

De maneira consolidada, observa-se que o Node.js é mais indicado para operações assíncronas e de alta concorrência, o Java para processamento intensivo e aplicações robustas, e o Python para prototipagem rápida e cenários de menor concorrência. Assim, a escolha da linguagem deve estar alinhada ao perfil de carga e às demandas de desempenho da aplicação.

7 Conclusão

Este trabalho apresentou uma análise comparativa do desempenho de três APIs REST desenvolvidas em Node.js (NestJS), Java (Spring Boot) e Python (FastAPI), avaliadas sob diferentes tipos de carga e cenários de operação. A execução dos experimentos seguiu rigorosamente a metodologia proposta, que inclui etapas sistemáticas de mapeamento do ambiente, padronização das interfaces, planejamento experimental, especificação das cargas de trabalho, instrumentação das medições, validação dos dados e análise estatística. Essa estrutura metodológica garantiu reprodutibilidade, controle das variáveis e confiabilidade dos resultados obtidos.

Os experimentos revelaram que o desempenho das três tecnologias varia conforme o tipo de carga aplicada, evidenciando comportamentos distintos em cenários I/O-bound, CPU-bound e cargas híbridas. O Node.js apresentou melhor desempenho em cenários I/O-bound, caracterizados por elevada concorrência e grande volume de requisições simultâneas. Já o Java demonstrou desempenho superior em operações CPU-bound, graças à otimização da JVM e ao compilador JIT, que proporcionaram tempos de resposta mais baixos e maior vazão em cenários de processamento intensivo. Por fim, o Python destacou-se em operações combinadas de leitura e agregações, apresentando performance em cargas leves e médias, embora tenha apresentado limitações de escalabilidade em cenários de maior concorrência.

De maneira geral, os resultados reforçam que não existe uma linguagem universalmente superior, mas sim tecnologias mais adequadas a determinados perfis de aplicação. Assim, a escolha da linguagem deve considerar o comportamento esperado da API, o tipo de demanda, as restrições de desempenho e o custo de desenvolvimento e manutenção.

Além disso, o estudo evidenciou que a metodologia proposta é adequada para análises experimentais de desempenho, podendo ser aplicada em outros contextos e com outras tecnologias, desde que sigam as mesmas etapas estruturais.

Conclui-se que o trabalho atingiu seu objetivo ao fornecer evidências experimentais que auxiliam desenvolvedores, arquitetos e pesquisadores na tomada de decisão sobre a linguagem mais apropriada para diferentes perfis de APIs REST, contribuindo para o avanço de estudos empíricos na área de desempenho de software.

Como perspectivas futuras, sugere-se ampliar o estudo incluindo: (a) diferentes frameworks em cada linguagem; (b) ambientes distribuídos ou em nuvem; (c) bancos de dados variados (NoSQL, in-memory, distribuídos); (d) análise de

eficiência energética; (e) avaliação sob múltiplos níveis de concorrência em máquinas com configurações mais robustas.

Referências

Ivanov, N. (2022) *Is Rust C++-fast? Benchmarking system languages on everyday routines.* East Lansing: Michigan State University. DOI: https://doi.org/10.48550/arXiv.2209.09127

Crestani, A. de R. (2024) *Uma comparação empírica em velocidade de processamento entre C++, Go, Rust, Python e JavaScript.* Bachelor's Thesis – Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre. Available at: http://hdl.handle.net/10183/272878.

Cavalcante, A. R. da S. (2022) *Análise de desempenho entre as linguagens Java e Scala.* Undergraduate Thesis – Centro de Tecnologia, Universidade Federal do Rio Grande do Norte, Natal. Available at: https://repositorio.ufrn.br/items/507f02ab-3709-4b4e-b6cd-39e89a9b9295

Zimmer, N. R. (2023) *Análise comparativa de desempenho de um banco de dados relacional em diferentes linguagens de programação.* Bachelor's Thesis – Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre. Available at: http://hdl.handle.net/10183/267656

Türkmen, G., Sezen, A. and Şengül, G. (2024) "Comparative analysis of programming languages utilized in artificial intelligence applications: features, performance, and suitability." *International Journal of Computational and Experimental Science and Engineering (IJCESEN)*, Ankara, vol. 10, no. 3, pp. 461–469. DOI: https://doi.org/10.22399/ijcesen.342.

Beierleib, L., Bauer, A., Leppich, R., Iffländer, L. and Kounev, S. (2023) "Efficient data processing: assessing the performance of different programming languages." In: *Companion of the 2023 ACM/SPEC International Conference on Performance Engineering (ICPE '23 Companion)*, Coimbra. New York: ACM, pp. 1–5. DOI: https://doi.org/10.1145/3578245.3584691

Costanzo, M., Rucci, E., Naiouf, M. and De Giusti, A. (2021) "Performance vs programming effort between Rust and C on multicore architectures: case study in N-body." *Latin American Computing Conference (CLEI)*. DOI: https://doi.org/10.48550/arXiv.2107.11912

Dymora, P. and Paszkiewicz, A. (2020) "Performance analysis of selected programming languages in the context of supporting decision-making processes for Industry 4.0." *Applied Sciences*, Basel, vol. 10, no. 23, pp. 1–17. DOI: https://doi.org/10.3390/app10238521

Pereira, R., et al. (2022) "Ranking programming languages by energy efficiency." *Science of Computer Programming*, vol. 213, p. 102609. DOI: https://doi.org/10.1016/j.scico.2021.102609

Souza, R. D. C., Florian, F. and Dallilo, F. D. (2023) "Uma análise de performance das linguagens de programação no processamento paralelo." *Revista Científica Multidisciplinar Núcleo do Conhecimento*, year 08, ed. 12, vol. 04, pp. 05–37, December. DOI:

https://doi.org/10.32749/nucleodoconhecimento.com.br/engenharia-da-computacao/linguagens-de-programacao

Di Meglio, S. and Starace, L. L. L. (2024) “Evaluating Performance and Resource Consumption of REST Frameworks and Execution Environments: Insights and Guidelines for Developers and Companies.” *IEEE Access.* DOI: https://doi.org/10.35877/454RI.jinav3041