COMPARAÇÃO DE DESEMPENHO ENTRE LINGUAGENS DE PROGRAMAÇÃO NA IMPLEMENTAÇÃO DE APIS

Felipe Trevisani Cardoso[1], Marlon de Matos de Oliveira[2]

Resumo: O crescimento da transformação digital nas organizações aumentou a necessidade de comunicação eficiente entre diferentes sistemas, aplicações e serviços. Nesse contexto, as Interfaces de Programação de Aplicações desempenham um papel fundamental ao viabilizar a integração entre sistemas por meio de interfaces de comunicação padronizadas. Ao mesmo tempo, diversas linguagens de programação e *frameworks* têm sido utilizados no desenvolvimento dessas APIs. Diante desse cenário, torna-se importante compreender como diferentes tecnologias se comportam quando aplicadas à implementação de serviços de integração. Este estudo tem como objetivo comparar o desempenho de diferentes linguagens de programação utilizadas no desenvolvimento de APIs. Para alcançar esse objetivo, são apresentados experimentos comparativos que avaliam o comportamento das linguagens analisadas em cenários controlados. Os resultados indicam que o gRPC apresentou menores tempos de resposta e maior vazão 80,95% dos cenários, enquanto o REST demonstrou maior estabilidade em situações de maior volume e concorrência. O GraphQL, por sua vez, apresentou maior flexibilidade na seleção dos dados, porém com maior custo computacional em consultas amplas. Também foi observado que a linguagem utilizada influencia o desempenho, o consumo de recursos e a estabilidade das APIs. Assim, a escolha tecnológica deve considerar conjuntamente, a abordagem de comunicação, a linguagem de programação, o volume de dados, o padrão de acesso, a estabilidade e o consumo de recursos.

Palavras-chave: integração de sistemas; APIs; linguagens de programação; desempenho de software; arquitetura de sistemas.

ABSTRACT: The growth of digital transformation in organizations has increased the need for efficient communication among different systems, ap-

[1]Curso de Ciência da Computação, Universidade do Extremo Sul Catarinense (Unesc), Criciúma - Santa Catarina - Brasil. felipet.cardosoo@unesc.net

[2]Orientador, Curso de Ciência da Computação, Universidade do Extremo Sul Catarinense (Unesc), Criciúma - Santa Catarina - Brasil. marlonoliveira@unesc.net

plications, and services. In this context, Application Programming Interfaces play a fundamental role in enabling system integration through standardized communication interfaces. At the same time, several programming languages and frameworks have been used in the development of these APIs. Given this scenario, it becomes important to understand how different technologies behave when applied to the implementation of integration services. This study aims to analyze and compare the performance of different programming languages used in API development. To achieve this objective, comparative experiments are presented to evaluate the behavior of the analyzed languages in controlled scenarios. The results indicate that gRPC presented lower response times and higher throughput in most scenarios, while REST demonstrated greater stability in situations involving higher data volume and concurrency. GraphQL, in turn, presented greater flexibility in data selection, but with higher computational cost in broad queries. It was also observed that the programming language used influences the performance, resource consumption, and stability of APIs. Thus, the technological choice should jointly consider the communication approach, the programming language, data volume, access pattern, stability, and resource consumption.

Keywords: system integration; APIs; programming languages; software performance; system architecture.

1 INTRODUÇÃO

Diante do avanço tecnológico que presenciamos, a diversidade crescente de sistemas, aplicações e serviços utilizados pelas organizações, que muitas vezes operam de forma independente, está criando ecossistemas complexos dentro das organizações. Estes ecossistemas exigem que as transmissões de dados entre as aplicações ocorram com segurança, agilidade e eficiência, atendendo as demandas de escalabilidade e flexibilidade que a organização necessita (Knoche; Hasselbring, 2021).

Segundo Goodwin (2024), As *Application Programming Interfaces* (APIs) são um conjunto de regras, padrões e protocolos que permitem a comunicação entre diferentes sistemas, possibilitando que as aplicações troquem dados, funcionalidades e serviços de forma organizada e padronizada por meio de interfaces. As APIs atuam como uma camada intermediária que permite a comunicação entre sistemas, abstraindo o código interno e facilitando o desenvolvimento e a integração.

Há diversas ferramentas que auxiliam na construção de APIs, incluindo linguagens de programação, frameworks e bibliotecas voltadas ao desenvolvimento de sistemas distribuídos. Nesse contexto, também se destacam diferentes abordagens de comunicação, como *Representational State Transfer* (REST), *Graph Query Language* (GraphQL) e *Google Remote Procedure Call* (gRPC), cada uma apresentando características distintas quanto à comunicação entre sistemas. Esse cenário possibilita ao desenvolvedor, de maneira estratégica, combinar as ferramentas e metodologias mais adequadas aos objetivos e requisitos técnicos do projeto (Pinto et al., 2022; Niswar et al., 2024).

Estudos como de Śliwa e Pańczyk (2021) têm investigado o desempenho e as características das diferentes arquiteturas utilizadas na construção de APIs, realizando comparações entre abordagens, analisando métricas relacionadas ao tempo de resposta, volume de dados transmitidos e número de requisições processadas por segundo. Os resultados indicam que cada tecnologia apresenta vantagens específicas, dependendo do cenário de uso, evidenciando que não existe uma solução universalmente superior para todos os contextos de aplicação.

Comumente utilizados no desenvolvimento de aplicações, os *frameworks* também devem ser considerados no desempenho das APIs, como analisado por Lubartowicz e Pańczyk (2020), onde foi realizada a comparação entre três opções diferentes e avaliado o tempo de resposta e o consumo de memória de cada um deles. Os resultados apresentaram métricas em que o *framework* Spring em Java obteve melhor desempenho do que o Symfony e o Ruby on Rails, considerando assim que, além da arquitetura, a linguagem de programação e as ferramentas utilizadas influenciam diretamente no desempenho final da aplicação

Pesquisas voltadas ao contexto de microsserviços, como a de Niswar et al. (2024), analisaram o comportamento de diferentes abordagens em ambientes distribuídos com múltiplos serviços e cargas simultâneas. Esses estudos indicam que o gRPC tende a apresentar melhor desempenho em cenários que exigem baixa latência e comunicação eficiente entre serviços, enquanto REST mantém um desempenho equilibrado e ampla compatibilidade, e GraphQL se destaca pela flexibilidade na recuperação de dados.

A falta de uma análise comparativa abrangente e atualizada sobre o impacto das diferentes linguagens de programação na construção de APIs tem resultado em escolhas pouco assertivas, que podem comprome-

ter o desempenho, a escalabilidade e a facilidade de manutenção dos sistemas. Observa-se que os estudos disponíveis focam principalmente nos protocolos e padrões de comunicação, sem explorar de forma prática como as linguagens influenciam o desenvolvimento e a eficiência das APIs.

Este trabalho tem como objetivo comparar o desempenho, atributo crítico de qualidade de software, associado ao tempo de resposta, à responsividade e ao consumo de recursos computacionais em tempo de execução conforme expressos por Zhao et al. (2023), de diferentes linguagens de programação na implementação de APIs utilizando distintas arquiteturas de comunicação. Para atingir esse objetivo, foram desenvolvidas aplicações equivalentes em C#, Go e Rust, implementando para cada uma delas as arquiteturas REST, GraphQL e gRPC, de modo a garantir condições comparáveis de avaliação. A partir dessas implementações, foram realizados testes com o intuito de coletar métricas que revelem o desempenho dessas tecnologias e abordagens. Com base nos resultados obtidos, busca-se identificar o comportamento de cada combinação entre linguagem e arquitetura, analisando suas vantagens e limitações.

As APIs como soluções estratégica para a comunicação entre aplicações, podem ser implementadas em diversos cenários, cada um com uma necessidade e recursos distintos. Para extrair o máximo do desempenho e eficiência dos recursos computacionais, é necessário pensar de forma estratégica quais ferramentas e abordagens serão usadas para alcançar os objetivos (Kamiński et al., 2023).

O processo de modernização de software, sendo eles migrados de outras arquiteturas ou construídos inicialmente em sistemas distribuídos, favorece o padrão de microsserviços. Nestas arquiteturas, as APIs são o mecanismo essencial que garante a comunicação entre os serviços, permitindo a troca de dados entre contêineres isolados. A orquestração e gestão desse ambiente devem garantir um fluxo transacional caracterizado por baixa latência e robustez, maximizando a eficiência operacional (Kazżanavičius; Mažeika, 2023).

Considerando a necessidade de fornecer subsídios técnicos que auxiliem desenvolvedores, arquitetos de software e organizações na tomada de decisões mais embasadas sobre a escolha da linguagem e das ferramentas mais adequadas, este trabalho justifica-se por realizar uma análise experimental, empírica e comparativa. Diferentemente dos estudos existentes que focam no desempenho de uma linguagem em diferentes abordagens, o presente estudo propõe a criação de APIs separadas e

independentes, utilizando três linguagens de programação distintas. Com isso, busca-se contribuir com evidências e subsídios técnicos claros sobre como cada linguagem opera em cada abordagem, permitindo a comparação de desempenho em todos os cenários e, assim, auxiliando na escolha que melhor atenda aos cenários de cada projeto.

Este trabalho está estruturado em quatro seções. A primeira apresenta a introdução, com uma visão geral das APIs e suas exigências de desempenho. A segunda seção aborda os materiais e métodos utilizados neste estudo. Na terceira seção, apresentam-se os resultados obtidos nos testes realizados e discussões em relação aos demais trabalhos existentes. A quarta seção descreve as conclusões obtidas com base nas métricas dos cenários de teste.

2 MATERIAIS E MÉTODOS

O presente estudo caracteriza-se como uma pesquisa experimental, na qual foram desenvolvidas e analisadas aplicações utilizando diferentes linguagens e arquiteturas de comunicação, com o objetivo de comparar seu desempenho em cenários controlados. A proposta técnica adotada foi a construção de serviços equivalentes em diferentes protocolos de comunicação. Dessa forma, o repositório foi organizado em uma matriz de implementações composta por nove serviços, distribuídos por linguagem e protocolo, permitindo a comparação técnica entre abordagens distintas de desenvolvimento de APIs.

2.1 AMBIENTE DE DESENVOLVIMENTO

O ambiente de desenvolvimento foi configurado para garantir estabilidade e isolamento das variáveis. Todos os testes foram executados em uma única estação de trabalho com sistema operacional Linux e distribuição Ubuntu 22.04.4 LTS, 8 GB de memória RAM, processador Intel Core i5-1135G7, pertencente à 11ª geração da linha Intel Core, com frequência base de 2,40 GHz e unidade de armazenamento SSD.

Esta configuração foi selecionada pela compatibilidade do sistema operacional com o Docker Engine nativo no sistema operacional Linux, facilitando a execução de aplicações em contêineres, visto que cada implementação foi isolada, reduzindo variações relacionadas aos recursos de hardware do sistema e permitindo dados mais consistentes sobre consumo destes recursos.

Definiu-se o limite de 512 MB de memória para cada contêiner,

com o objetivo de controlar o uso de recursos durante os testes.

As ferramentas de software, suas versões e respectivas finalidades na pesquisa são detalhadas na Tabela 1.

Tabela 1 - Tecnologias utilizadas no ambiente de desenvolvimento

Ferramenta	**Versão**	**Finalidade**
C#	8	Linguagem
Golang	1.22.2	Linguagem
Rust	2024 edition	Linguagem
ASP.NET Core	8.0	Framework Web
EF Core	8.0.13	ORM
Npgsql	8.0.4	Driver de Banco
HotChocolate	15.1.11	Framework GraphQL
gRPC (ASP.NET)	2.65.0	Framework RPC
lib/pq	1.10.9	Driver de Banco
GraphQL-Go	0.8.1	Framework GraphQL
gRPC-Go	1.64.0	Framework RPC
Protobuf-Go	1.36.5	Serialização
Axum	0.8.1	Framework Web
SQLx	0.8.6	ORM
Tokio	1.47.1	Runtime
Async-GraphQL Axum	7.0.17	Framework GraphQL
Tonic	0.12.3	Framework RPC
Prost	0.13.5	Serialização
Docker	29.4.0	Containerização
.NET	8.0	Runtime
Go (base)	1.22-alpine	Imagem Base
Rust (base)	1.89-bookworm	Imagem Base
k6	1.7.1	Teste de Carga

Fonte: Elaborado pelo autor.

Utilizou-se a ferramenta k6 para realização dos testes, por ser uma ferramenta utilizada nos estudos de Stępień e Skublewska-Paszkowska (2025), no qual seu uso é mencionado por conta das funcionalidades que a ferramenta oferece, como cenários de carga, usuários simultâneos e métricas como tempo de resposta, throughput e volume de dados.

2.2 PADRÃO ARQUITETURAL

A adoção de uma arquitetura monolítica em camadas, teve como objetivo de garantir a uniformidade estrutural entre as implementações e não criar complexidades arquiteturais desnecessárias.

Optou-se por essa estrutura por se basear em princípios de separação de responsabilidades, amplamente adotados na engenharia de

software, facilitando a compreensão das responsabilidades de cada camada como pode-se verificar na Figura 1.

Figura 1 - Fluxograma da arquitetura

Fonte: Elaborado pelo autor.

A primeira camada da aplicação é a camada de interface. Esta camada é responsável por definir os *endpoints* que foram utilizados nas requisições, assim como passar a requisição para a camada de aplicação e devolver a resposta para o solicitante assim que todo o processo for executado.

A camada de aplicação é executada logo em seguida, onde estão presentes as regras de negócio, as validações e o tratamento dos dados, respeitando a lógica implementada no sistema e chamando a próxima camada, a de persistência.

A ultima camada da arquitetura é a camada de persistência, que possui a função de comunicar com o banco de dados, executando comandos de leitura, escrita, atualização e exclusão conforme solicitado na requisição feita pelo solicitante.

Todas as implementações possuem a mesma estrutura para que a arquitetura não seja um fator relevante nos dados obtidos em teste, havendo alterações mínimas a nível de código, consequentes da linguagem e da abordagem utilizada.

2.3 MODELAGEM DA APLICAÇÃO

A aplicação foi construída de forma minimalista, baseada em um domínio simplificado de gerenciamento de produtos, com objetivo de representar cenários comuns em sistemas corporativos, focando em funcionalidades básicas de criação, leitura, atualização e exclusão (CRUD). As funcionalidades implementadas contemplam a inserção de produtos, de forma unitária ou em lote. Consulta, que pode ser individual, geral ou por id. Atualização dos dados do produto, incluindo controle de saldo e estoque; e a exclusão, unitária ou geral.

A entidade principal, denominada “Produto”, foi projetada contendo atributos heterogêneos, incluindo dados numéricos, textuais e biná-

rios, tais como identificador, nome, preço, categoria, descrição e saldo. Essa composição foi adotada com o objetivo de representar um cenário mais próximo de sistemas corporativos de gerenciamento de produtos.

Todos os projetos utilizaram o mesmo banco de dados, PostgreSql, contemplando os mesmos parâmetros, a mesma configuração e uma bateria de teste (linguagem e abordagem) após a outra, para que este não seja um ponto de interferência em relação aos testes.

2.4 ESTRATÉGIA DE IMPLEMENTAÇÃO

Para assegurar a validade dos resultados, todas as aplicações estão isoladas por containers, utilizam a mesma estrutura de dados, utilizam o mesmo volume de dados, as mesmas operações e regras de negócio, assim como se comunicam com um banco em comum que é reiniciado ao inicio e fim de cada execução dos testes.

Os teste foram executados no mesmo ambiente computacional, em dois cenários com mesmo volume de dados. Os primeiros cenários são de requisições sequenciais, o segundo cenário possui requisições simultâneas. Ambos os cenários possuem volumes de produtos, separados em unitário, pequeno porte (mil produtos), médio porte (cinco mil produtos) e grande porte (vinte e cinco mil produtos). Este valores foram escolhidos por conta de serem uma crescente geométrica aceitável, visto que os testes com cinquenta mil produtos, realizados no inicio da implementação, em ambos os cenários geravam uma sobrecarga no sistema operacional, impossibilitando a progressão dos testes.

Como cada teste executou todos os *endpoints* criados, estes geraram uma quantidade de requisições considerável, como pode ser visto na Tabela 2.

Tabela 2 - Resultados de requisições por perfil de carga

Perfil	**Produtos**	**Sequencial**	**Simultâneo**
Unitário	1	8 requisições	Não realizado
Pequeno	1.000	1.043 requisições	1.207 requisições
Médio	5.000	5.412 requisições	7.052 requisições
Grande	25.000	27.052 requisições	37.252 requisições

Fonte: Elaborado pelo autor.

Sendo que cada uma das rotas que são disponibilizadas foram divididas para executar em uma quantidade diferente de vezes, onde seguem uma escala 1:5:25. A única bateria de testes que foi realizada e que esteve fora dessa escala foi a inserção individual de produtos, pois a inser-

ção de produtos em lote cria uma grande quantidade de produtos. Pode-se ver detalhadamente na Tabela 3 a distribuição das requisições sequenciais, e na Tabela 4 a distribuição das requisições com usuários virtuais para execução simultânea em cada bateria de testes.

Tabela 3 - Quantidade de requisições por endpoint em cenário sequencial

Endpoint	**unit**	**small**	**medium**	**high**
DELETE /Product/all – limpeza inicial	1	1	1	1
POST /Product/batch	0	1	10	50
POST /Product	1	990	4.900	24.500
GET /Product/all	1	10	100	500
GET /Product?id={id}	1	10	100	500
PUT /Product/{id}	1	10	100	500
GET /Product?id={id} após update	1	10	100	500
DELETE /Product/{id}	1	10	100	500
DELETE /Product/all – limpeza final	1	1	1	1

Fonte: Elaborado pelo autor.

Tabela 4 - Quantidade de requisições por endpoint em cenário simultâneo

Endpoint	**small (5 VUs)**	**medium (5 VUs)**	**high (5 VUs)**
DELETE /Product/all	2	2	2
POST /Product/batch	5	50	250
POST /Product	950	4.500	22.500
GET /Product/all	50	500	2.500
GET /Product?id={id}	100	1000	5.000
PUT /Product/{id}	50	500	2.500
DELETE /Product/{id}	50	500	2.500

Fonte: Elaborado pelo autor.

2.5 PROCEDIMENTO EXPERIMENTAL

Inicialmente, realizou-se a preparação da máquina de execução, com instalação das dependências básicas, Docker, Docker Compose e k6. Em seguida, o repositório do projeto foi clonado e os serviços foram construídos e inicializados por meio do comando `docker compose up -d -build`. Após a inicialização, a disponibilidade dos containers foi verificada com `docker compose ps`, enquanto a conectividade do banco de dados foi validada por meio do comando `pg_isready`.

Para reduzir efeitos de inicialização, foi prevista uma etapa de aquecimento antes das medições oficiais, contando com inserção de dados e consultas no banco de dados, para que este não fosse uma variável significativa nos primeiros testes, assim como foi percebido nas primeiras execuções de testes.

Após a validação da base, foram executados 3 baterias de testes funcionais em cada API, contemplando operações de criação, consulta e listagem de produtos com a ferramenta k6. Os scripts de teste foram organizados por abordagem de integração e linguagem de programação, mantendo os mesmos parâmetros de carga para todas as implementações.

Para cada cenário, foram coletados dados como tempo de resposta, taxa de erro, e quantidade por requisição executadas além de consumo de memória, consumo de CPU e *throughput* em cada cenário de teste. Os resultados foram exportados em arquivos JSON por meio do parâmetro `-summary-export`, permitindo posteriormente a consolidação e análise comparativa.

Um teste foi considerado válido quando a API avaliada respondeu corretamente às requisições, mesmo registrando uma resposta de erro explícito como *timeout*, visto que é um dado relevante para este trabalho. Entretanto, foram desconsiderados a bateria de teste que conteve cenários onde o consumo dos recursos foi tão elevado que derrubou o container, pois estes não retornavam um valor consistente para análise. Dessa forma, buscou-se garantir que os resultados de desempenho refletissem o comportamento das implementações sob condições equivalentes de execução.

2.6 COLETA, MONITORAMENTO E MÉTRICAS

Para embasar o estudo com métricas quantitativas que expressam pontos a serem avaliados no cotidiano das organizações, as métricas utilizadas são tempo de resposta, *throughput*, uso de CPU, consumo de memória e escalabilidade que podem ser interpretadas na Tabela 5

Tabela 5 - Métricas utilizadas

Métrica	**Definição**	**Ferramenta**
Tempo de resposta	Intervalo entre o envio da requisição e o recebimento da resposta pelo cliente	k6
Throughput	Número de requisições processadas por segundo (RPS)	k6
Uso de CPU	Percentual de processamento utilizado durante a execução da aplicação	Docker Stats
Consumo de memória	Quantidade de memória RAM utilizada pela aplicação em execução	Docker Stats

Escalabilidade	Comportamento do sistema diante do aumento do número de requisições simultâneas	k6 / Docker Stats

Fonte: Elaborado pelo autor.

Estas métricas foram monitoras em tempo de execução dos testes e utilizadas comparativamente em cada linguagem e abordagem definidas para que fossem identificados os pontos fortes e fracos em cada um dos cenários.

2.7 TRATAMENTO E ANÁLISE DOS DADOS

Os dados obtidos nos testes de desempenho foram tratados a partir dos dados gerados pela ferramenta K6 e pelo docker status. O tratamento dos dados constituiu na organização, consolidação e comparação das métricas coletadas para uma analise de cada combinação entre linguagem abordagem e perfil de carga.

Os experimentos foram utilizados como base técnica para observar o comportamento das implementações sob condições controladas. Desta forma, os resultados foram interpretados de forma comparativa, considerando os valores obtidos em teste.

As principais métricas utilizadas na análise foram o tempo médio de resposta, a mediana, o *throughput*, a quantidade total de requisições executadas, a taxa de falha e os *checks* funcionais definidos nos *scripts* de teste. Essas métricas permitiram avaliar não apenas o tempo médio de atendimento das requisições, mas também a estabilidade das APIs em diferentes níveis de carga, uma vez que o desvio padrão evidencia variações de desempenho que podem não ser percebidas apenas pela média. Os códigos fonte deste trabalho podem ser encontrados no GitHub[3].

3 RESULTADOS E DISCUSSÃO

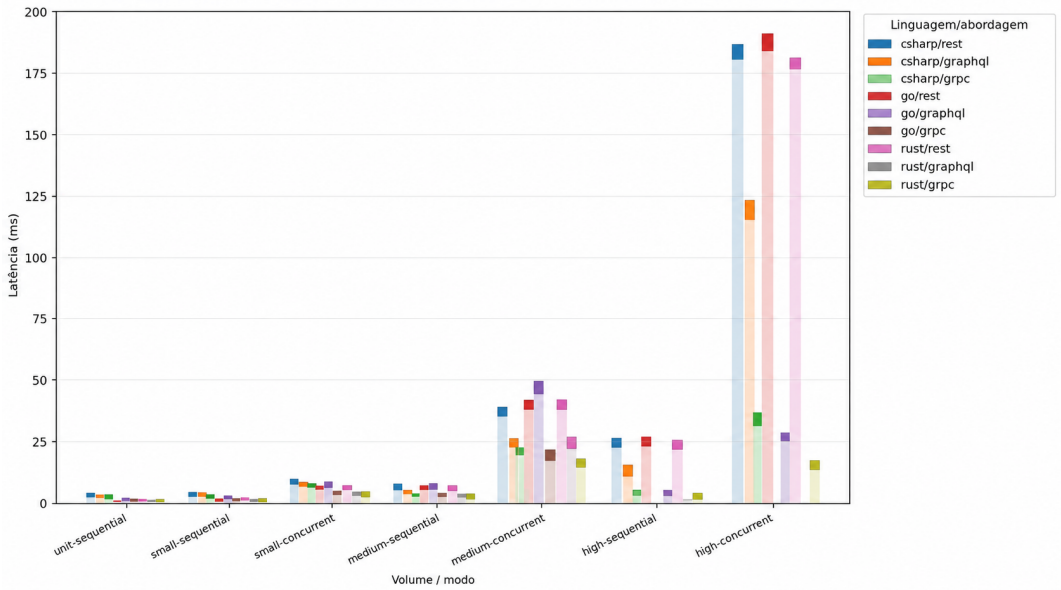
Os resultados obtidos evidenciam que o desempenho das APIs variou conforme a combinação entre linguagem de programação, abordagem de comunicação, volume de dados e forma de execução das requisições. A análise das métricas coletadas demonstrou diferenças relevantes em latência, *throughput*, consumo de CPU e uso de memória, principalmente nos cenários de maior carga, nos quais as características internas de cada abordagem se tornaram mais perceptíveis.

Para fins de síntese e objetividade na apresentação dos resultados, esta seção concentra-se nos dados obtidos no volume médio de requisições. Esse volume foi selecionado por representar uma carga suficientemente expressiva para evidenciar diferenças de desempenho entre as linguagens e abordagens analisa-

[3]https://github.com/felipetrevisanic/TrabalhoPratico

das, sem os efeitos extremos observados nos cenários de maior carga conforme pode-se visualizar na Figura 2. Ainda assim, todos os demais cenários executados, incluindo os volumes unitário, pequeno e alto, encontram-se disponíveis no apêndice, permitindo a consulta completa dos dados experimentais.

Figura 2 - Gráfico exibindo os resultados da latência nos testes



Fonte: Elaborado pelo autor.

Os resultados de latência indicam que o gRPC apresentou um menor tempo médio de resposta em 80,95% dos cenários, possivelmente em razão do uso de HTTP/2 e serialização binária. Contudo, no cenário de maior volume com concorrência, as implementações gRPC apresentaram falhas associadas à operação que busca todas as entidades em uma única requisição, *get_all*, devido ao retorno massivo de dados.

Essas falhas não caracterizam, necessariamente, uma limitação estrutural do gRPC, mas indicam que a recuperação integral dos registros não foi adequada para cargas elevadas e simultâneas. Estratégias como paginação, limitação de registros, streaming ou divisão das consultas em lotes poderiam reduzir o tamanho das respostas e aumentar a estabilidade.

Nesse cenário crítico, o REST foi a abordagem mais estável para fins comparativos, enquanto o GraphQL apresentou maior consumo de recursos, ocasionando interrupções no container e impedindo a conclusão completa dos testes. Assim, embora o GraphQL ofereça flexibilidade na seleção dos dados, sua execução tende a ser mais custosa em consultas amplas e concorrentes como executadas nos testes.

Conforme pode-se visualizar na Tabela 6, o desvio padrão das linguagens é consideravelmente alto. Esse comportamento ocorre em razão da diferença entre os tempos de execução das operações avaliadas. Requisições de inserção

de um único produto são expressamente mais rápidas, enquanto as requisições de consulta que retornam toda a base de dados de produtos demandam maior processamento, gerando resultados com variabilidade alta.

Tabela 6 - Latência média no volume médio

Ling.	Abord.	Volume	Modo	Média(ms)	Desvio padrão
C#	GraphQL	médio	concorrente	24.90	± 77.33
C#	gRPC	médio	concorrente	20.12	± 61.19
C#	REST	médio	concorrente	37.63	± 124.12
C#	GraphQL	médio	sequencial	4.53	± 16.79
C#	gRPC	médio	sequencial	3.90	± 13.48
C#	REST	médio	sequencial	6.63	± 32.31
Go	GraphQL	médio	concorrente	46.53	± 159.32
Go	gRPC	médio	concorrente	19.69	± 64.46
Go	REST	médio	concorrente	39.50	± 136.93
Go	GraphQL	médio	sequencial	6.38	± 33.42
Go	gRPC	médio	sequencial	3.36	± 14.21
Go	REST	médio	sequencial	6.05	± 34.42
Rust	GraphQL	médio	concorrente	24.79	± 83.96
Rust	gRPC	médio	concorrente	16.46	± 52.68
Rust	REST	médio	concorrente	38.72	± 134.39
Rust	GraphQL	médio	sequencial	3.86	± 18.70
Rust	gRPC	médio	sequencial	2.85	± 12.17
Rust	REST	médio	sequencial	6.14	± 32.96

Fonte: Elaborado pelo autor.

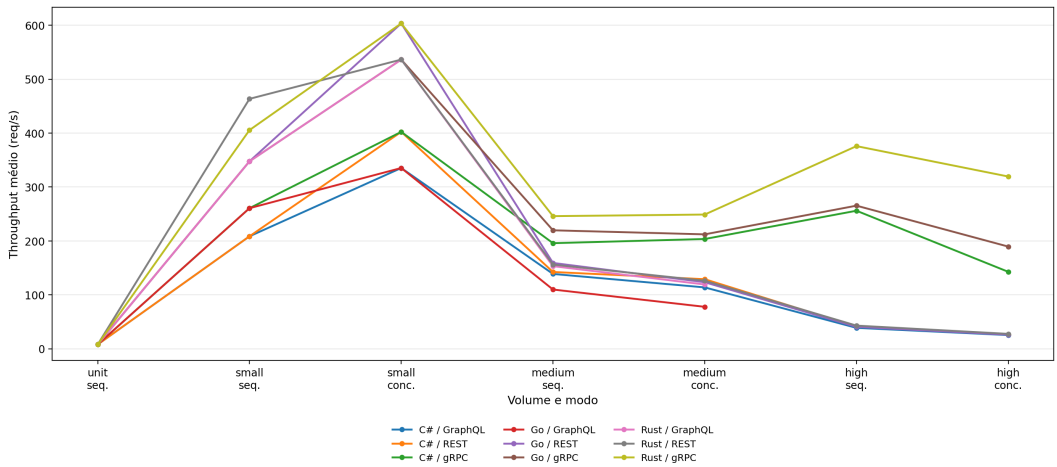
Em relação ao *throughput*, o gRPC acompanhou a tendência observada na latência, indicando maior capacidade potencial de processamento nos cenários em que os testes foram concluídos, sintetizado na Figura 3. Contudo, as falhas registradas no cenário *high* concorrente impedem classificá-lo como a abordagem mais eficiente de forma absoluta para esse volume. Nesse caso, o REST apresentou maior estabilidade operacional, enquanto o GraphQL, embora tenha obtido boa latência em algumas combinações, não concluiu os testes em Rust e Go devido à interrupção dos containers. Desta forma, a análise de vazão evidencia que o desempenho bruto do gRPC foi superior, detalhado na Tabela 7, mas a sustentação da carga dependeu diretamente da estabilidade da implementação e do controle do volume de dados retornado.

Tabela 7 - *throughput* no volume médio

Ling.	Abord.	Volume	Modo	Média(req/s)	Desvio padrão
C#	GraphQL	médio	concorrente	115.88	± 3.40
C#	gRPC	médio	concorrente	209.11	± 8.15
C#	REST	médio	concorrente	131.02	± 1.09
C#	GraphQL	médio	sequencial	141.74	± 0.23
C#	gRPC	médio	sequencial	201.48	± 7.88
C#	REST	médio	sequencial	148.98	± 0.62
Go	GraphQL	médio	concorrente	78.10	± 5.65
Go	gRPC	médio	concorrente	216.91	± 12.72
Go	REST	médio	concorrente	125.95	± 8.99
Go	GraphQL	médio	sequencial	112.35	± 4.65
Go	gRPC	médio	sequencial	226.36	± 10.06
Go	REST	médio	sequencial	163.50	± 0.33
Rust	GraphQL	médio	concorrente	121.55	± 8.25
Rust	gRPC	médio	concorrente	256.10	± 5.99
Rust	REST	médio	concorrente	128.07	± 6.89
Rust	GraphQL	médio	sequencial	156.62	± 0.98
Rust	gRPC	médio	sequencial	257.26	± 3.07
Rust	REST	médio	sequencial	161.34	± 4.10

Fonte: Elaborado pelo autor.

Figura 3 - Gráfico exibindo os resultados de *throughput* nos testes

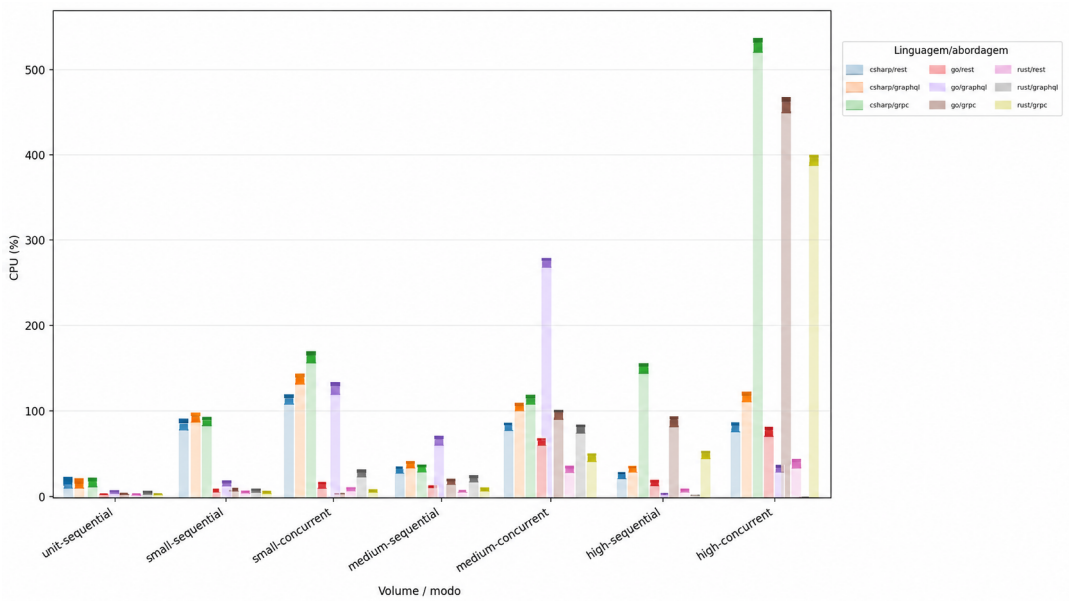


Fonte: Elaborado pelo autor.

Quanto ao uso de CPU e memória, os resultados reforçam que a eficiência deve ser analisada em conjunto com o desempenho entregue. Entre as linguagens, Rust apresentou a melhor relação geral, combinando baixa latência e bom aproveitamento dos recursos nos cenários avaliados. Entre as abordagens,

o REST se destacou pela estabilidade no cenário concorrente, enquanto o gRPC apresentou maior desempenho, mas com limitação decorrente das falhas nessa carga. O GraphQL, apesar de gerar resultados de latência inferior ao REST em algumas combinações no volume, este não concluiu os testes em Rust e Go, o que limita sua avaliação quanto à robustez sob maior consumo de recursos. Assim, Rust se destacou entre as linguagens evidenciado nas Figuras 4 e 5 , e REST apresentou o comportamento mais consistente entre as abordagens no cenário de maior carga, detalhados na Tabela 8 e na Tabela 9.

Figura 4 - Gráfico exibindo o consumo de CPU nos testes



Fonte: Elaborado pelo autor.

Tabela 8 - Consumo médio de CPU no volume medium

Ling.	Abord.	Volume	Modo	Média(%)	Desvio padrão
C#	GraphQL	médio	concorrente	105.48	± 1.26
C#	gRPC	médio	concorrente	115.13	± 2.26
C#	REST	médio	concorrente	80.66	± 2.82
C#	GraphQL	médio	sequencial	41.16	± 0.91
C#	gRPC	médio	sequencial	36.83	± 1.09
C#	REST	médio	sequencial	33.34	± 1.90
Go	GraphQL	médio	concorrente	273.64	± 2.30
Go	gRPC	médio	concorrente	96.71	± 3.01
Go	REST	médio	concorrente	65.59	± 2.43
Go	GraphQL	médio	sequencial	66.91	± 1.50
Go	gRPC	médio	sequencial	18.42	± 1.44

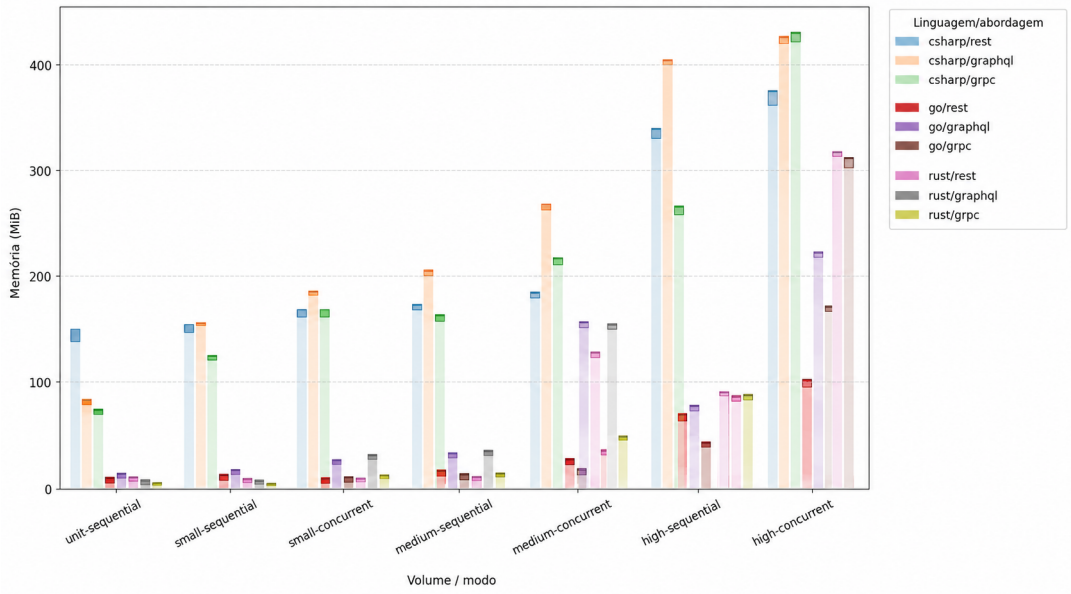
Continua na próxima página

Tabela 8 – Continuação

Ling.	Abord.	Volume	Modo	Média	Desvio padrão
Go	REST	médio	sequencial	13.70	± 0.33
Rust	GraphQL	médio	concorrente	79.78	± 3.59
Rust	gRPC	médio	concorrente	49.49	± 3.51
Rust	REST	médio	concorrente	37.09	± 1.53
Rust	GraphQL	médio	sequencial	15.37	± 0.17
Rust	gRPC	médio	sequencial	9.28	± 0.36
Rust	REST	médio	sequencial	7.00	± 0.81

Fonte: Elaborado pelo autor.

Figura 5 - Gráfico exibindo o consumo de memória RAM nos testes



Fonte: Elaborado pelo autor.

Tabela 9 - Consumo médio de memória no volume medium

Ling.	Abord.	Volume	Modo	Média(MB)	Desvio padrão
C#	GraphQL	médio	concorrente	265.85	± 1.67
C#	gRPC	médio	concorrente	208.31	± 4.93
C#	REST	médio	concorrente	177.99	± 4.54
C#	GraphQL	médio	sequencial	199.55	± 4.14
C#	gRPC	médio	sequencial	158.84	± 3.22
C#	REST	médio	sequencial	168.66	± 4.01
Go	GraphQL	médio	concorrente	149.63	± 1.95
Go	gRPC	médio	concorrente	20.05	± 0.97

Continua na próxima página

Tabela 9 – Continuação

Ling.	Abord.	Volume	Modo	Média	Desvio padrão
Go	REST	médio	concorrente	25.42	± 0.11
Go	GraphQL	médio	sequencial	33.45	± 0.01
Go	gRPC	médio	sequencial	13.62	± 0.64
Go	REST	médio	sequencial	17.04	± 1.14
Rust	GraphQL	médio	concorrente	127.27	± 15.62
Rust	REST	médio	concorrente	30.95	± 3.73
Rust	gRPC	médio	concorrente	44.12	± 1.28
Rust	GraphQL	médio	sequencial	29.16	± 7.15
Rust	REST	médio	sequencial	8.72	± 0.87
Rust	gRPC	médio	sequencial	18.00	± 0.19

Fonte: Elaborado pelo autor.

A análise de escalabilidade evidencia que o comportamento das linguagens e abordagens variou conforme o aumento do volume de dados e da concorrência. Nos volumes menores, as diferenças entre as abordagens foram menos críticas, permitindo que gRPC, REST e GraphQL concluíssem os testes com maior previsibilidade. À medida que o volume aumentou, as diferenças tornaram-se mais evidentes.

O Rust apresentou melhor comportamento geral de escalabilidade, mantendo tempos reduzidos e melhor estabilidade relativa entre os cenários. O Go apresentou desempenho competitivo em alguns testes, mas sofreu maior impacto em cenários de carga elevada, especialmente quando combinado ao GraphQL. O C# manteve comportamento intermediário, com resultados estáveis em parte dos cenários.

Quanto às abordagens, o gRPC escalou melhor em termos de velocidade até o limite em que a consulta massiva comprometeu a estabilidade; o REST escalou de forma mais conservadora, porém mais estável; e o GraphQL apresentou maior dificuldade de escalabilidade quando submetido a alto volume de dados e concorrência. Portanto, os resultados indicam que a escalabilidade não depende apenas da linguagem ou do protocolo isoladamente, mas da combinação entre tecnologia, estratégia de consulta e volume de dados trafegado.

Os resultados obtidos apresentam convergência com os trabalhos correlatos. Niswar et al. (2024) também observaram menor tempo de resposta no gRPC em comparação ao REST e ao GraphQL, além de maior uso de CPU pelo GraphQL. Esse comportamento foi semelhante ao encontrado neste trabalho, em que o gRPC apresentou melhor desempenho

em latência, enquanto o GraphQL foi mais impactado nos cenários de maior carga. A explicação apontada pelos autores, relacionada ao uso de HTTP/2 pelo gRPC e seus mecanismos de multiplexação, também ajuda a justificar o desempenho superior observado nos testes realizados.

De forma semelhante, Kamiński et al. (2023) indicam o gRPC como uma alternativa eficiente e confiável para transferência de dados, enquanto o GraphQL aparece entre as abordagens mais lentas nos cenários avaliados. Neste estudo, essa tendência se manifestou principalmente nos testes de maior volume, nos quais o GraphQL apresentou maior degradação e falha no cenário *high* concorrente. No entanto, este trabalho amplia a comparação ao incluir também diferentes linguagens de programação, permitindo observar que o desempenho final não depende apenas do protocolo adotado.

Essa relação também é reforçada por Lubartowicz e Pańczyk (2020), que demonstra que linguagem, *framework* e ambiente de execução influenciam diretamente o tempo de resposta e o consumo de memória em serviços. Os autores observaram que o Spring apresentou melhor tempo de execução, mas com maior consumo de memória, enquanto o Symfony consumiu menos memória, porém com tempos de resposta superiores. Essa relação entre desempenho e uso de recursos também foi observada neste trabalho, especialmente no comportamento do Rust, que apresentou melhor relação entre latência e escalabilidade em parte dos cenários avaliados, mesmo que tenha apresentado um consumo de CPU maior que os outros, enquanto Go e C# variaram conforme o volume de dados e a abordagem utilizada.

Os resultados reforçam que não há uma abordagem de comunicação universalmente superior para todos os cenários. O gRPC apresentou melhor desempenho bruto em latência e *throughput* nos testes concluídos, mas demonstrou limitações sob carga concorrente elevada em operações com grande volume de dados. O REST, embora nem sempre tenha obtido os menores tempos de resposta, mostrou maior estabilidade no cenário crítico. O GraphQL, por sua vez, manteve a vantagem da flexibilidade na seleção dos dados, mas apresentou maior custo computacional em consultas amplas.

Também se observou que a linguagem de programação influencia o comportamento das APIs, aspecto ainda pouco abordado na literatura, que normalmente concentra a análise nas abordagens de comunicação. Neste trabalho, essa influência foi mais evidente no GraphQL com Go, en-

quanto nas demais combinações as diferenças entre C#, Go e Rust não alteraram de forma significativa a tendência geral dos resultados. Ainda assim, os dados demonstram que a linguagem utilizada pode impactar os resultados obtidos, especialmente quando associada às características da abordagem de comunicação adotada. Assim, a escolha tecnológica deve considerar não apenas a abordagem de comunicação, mas também a linguagem, a estabilidade, o consumo de recursos, o volume de dados e o padrão de acesso da aplicação.

4 CONCLUSÃO

Este trabalho realizou uma análise experimental do desempenho de APIs desenvolvidas em C#, Go e Rust, utilizando as abordagens REST, GraphQL e gRPC. Os testes foram executados em ambiente controlado, com diferentes volumes de dados e modos de execução, permitindo avaliar latência, *throughput*, consumo de recursos e estabilidade. Com isso, o estudo buscou identificar como a linguagem e a abordagem de comunicação influenciam o desempenho das APIs, fornecendo subsídios técnicos para decisões de desenvolvimento.

A principal contribuição deste estudo está na comparação entre linguagens de programação. Nesse ponto, os testes indicam que Rust apresentou a melhor relação geral entre desempenho e escalabilidade, sugerindo que a linguagem utilizada na implementação da API também exerce influência relevante sobre o resultado final, especialmente quando combinada com protocolos eficientes e estratégias adequadas de controle de carga.

O Rust apresentou os resultados mais eficientes entre as linguagens avaliadas, especialmente em relação à latência. Entre as abordagens de comunicação, o REST demonstrou maior estabilidade nos diferentes cenários, enquanto o gRPC apresentou menores tempos de resposta na maior parte dos testes avaliados, embora tenha demonstrado limitações no cenário de maior carga. O GraphQL, por sua vez, foi mais afetado pelo aumento do volume de dados, apresentando maior degradação em cenários mais exigentes.

Com isso, conclui-se que não há uma alternativa única que atenda melhor a todos os contextos. A escolha entre linguagem de programação e abordagem de comunicação deve considerar os requisitos da aplicação, principalmente em relação ao volume de dados, à necessidade de baixa latência, e à capacidade de escalabilidade. Nos cenários avaliados, Rust

se mostrou a linguagem mais eficiente, REST a abordagem mais estável, gRPC a alternativa com maior potencial de desempenho e GraphQL a opção mais sensível ao aumento da carga.

Como continuidade deste estudo, sugere-se ampliar os testes para ambientes distribuídos, aplicar estratégias de paginação e controle de resposta em cenários de maior volume, além de aprofundar na diversidade dos dados, utilizando arquivos (pdf, xlsx, csv), imagens e videos.

DECLARAÇÃO DE USO DE INTELIGÊNCIA ARTIFICIAL

Os autores declaram que ferramentas de Inteligência Artificial generativa foram utilizadas exclusivamente como apoio auxiliar à elaboração deste trabalho, incluindo assistência na revisão linguística, estruturação textual e organização de ideias. Nenhuma ferramenta de IA foi utilizada para substituição da autoria intelectual, análise científica, interpretação dos resultados ou tomada de decisões acadêmicas. A responsabilidade integral pelo conteúdo, originalidade, veracidade das informações e conformidade com os princípios de integridade científica permanece integralmente atribuída aos autores.

REFERÊNCIAS

GOODWIN, M. **What is an API (application programming interface)?** 2024. <https://www.ibm.com/think/topics/api>. Acesso em: 11 Jun. 2025.

KAMIńSKI Łukasz et al. **Comparative review of selected internet communication protocols**. Miscellaneous, v. 48, n. 1, p. 39. 2023.

KAZżANAVIČIUS, J.; MAŽEIKA, D. **The evaluation of microservice communication while decomposing monoliths**. Computing and Informatics, v. 42, n. 1, p. 1–36. 2023.

KNOCHE, H.; HASSELBRING, W. **Continuous api evolution in heterogenous enterprise software systems**. In: **2021 IEEE 18th International Conference on Software Architecture (ICSA)**. [S.l.: s.n.], 2021. 2021. p. 58–68.

LUBARTOWICZ, P.; PAńCZYK, B. **Performance comparison of web services using symfony, spring, and rails examples**. Journal of Computer Sciences Institute, v. 17, p. 384–389. 2020.

NISWAR, M. et al. **Performance evaluation of microservices communication with rest, graphql, and grpc**. International Journal of Electronics and Telecommunications, v. 70, n. 2, p. 429–436. 2024.

PINTO, P. et al. **Pegasus: Performance engineering for software applications targeting HPC systems**. IEEE Transactions on Software Engineering, v. 48, n. 3, p. 732–754. 2022.

STęPIEń, K.; SKUBLEWSKA-PASZKOWSKA, M. **Performance evaluation of rest and graphql api aproaches in data retrieval scenarios using nestjs**. Journal of Computer Sciences Institute, v. 36, p. 350–356. Acesso em: 07 jul. 2026. 2025, Disponível em: <https://ph.pollub.pl/index.php/jcsi/article/view/7794>.

ZHAO, Y. et al. **A large-scale empirical study of real-life performance issues in open source projects**. IEEE Transactions on Software Engineering, v. 49, n. 2, p. 924–946. 2023.

ŚLIWA, M.; PAńCZYK, B. **Performance comparison of programming interfaces on the example of rest api, graphql and grpc**. Journal of Computer Sciences Institute, v. 21, p. 356–361. 2021.

APÊNDICE A - RESULTADOS COMPLETOS DOS TESTES DE LATÊNCIA

Este apêndice apresenta os resultados completos da métrica de latência obtidos nos testes de desempenho. Enquanto a seção de resultados e discussão prioriza o volume médio para manter a objetividade da análise, as tabelas a seguir reúnem os demais cenários avaliados, permitindo a consulta integral dos dados por linguagem, abordagem de comunicação, volume de dados e modo de execução.

A latência foi medida em milissegundos e representa o intervalo entre o envio da requisição pelo cliente e o recebimento da resposta da API. Os valores são apresentados como média acompanhada do desvio padrão, permitindo observar tanto o desempenho médio quanto a variação dos tempos de resposta em cada cenário

Tabela 10 - Latência média por volume e modo de execução

Ling.	Abord.	Volume	Modo	Média (ms)	DP
C#	GraphQL	high	concorrente	104.53	± 380.55
C#	GraphQL	high	sequencial	13.57	± 85.71
C#	GraphQL	médio	concorrente	24.90	± 77.33
C#	GraphQL	médio	sequencial	4.53	± 16.79
C#	GraphQL	small	concorrente	7.76	± 16.74
C#	GraphQL	small	sequencial	3.65	± 5.92
C#	GraphQL	unit	sequencial	3.29	± 1.43
C#	gRPC	high	concorrente	34.52	± 115.45
C#	gRPC	high	sequencial	3.78	± 17.29
C#	gRPC	médio	concorrente	20.12	± 61.19
C#	gRPC	médio	sequencial	3.90	± 13.48
C#	gRPC	small	concorrente	6.55	± 12.23
C#	gRPC	small	sequencial	3.26	± 4.13
C#	gRPC	unit	sequencial	3.17	± 1.24
C#	REST	high	concorrente	186.86	± 668.11
C#	REST	high	sequencial	24.15	± 162.08
C#	REST	médio	concorrente	37.63	± 124.12
C#	REST	médio	sequencial	6.63	± 32.31
C#	REST	small	concorrente	8.40	± 20.96
C#	REST	small	sequencial	3.72	± 7.02
C#	REST	unit	sequencial	3.50	± 1.93
Go	GraphQL	high	concorrente	2.88	± 2.17

Continua na próxima página

Tabela 10 – Continuação

Ling.	Abord.	Volume	Modo	Média (ms)	DP
Go	GraphQL	high	sequencial	23.72	± 160.57
Go	GraphQL	médio	concorrente	46.53	± 159.32
Go	GraphQL	médio	sequencial	6.38	± 33.42
Go	GraphQL	small	concorrente	7.17	± 21.28
Go	GraphQL	small	sequencial	2.32	± 5.18
Go	GraphQL	unit	sequencial	1.54	± 0.83
Go	gRPC	high	concorrente	26.11	± 86.97
Go	gRPC	high	sequencial	3.66	± 16.30
Go	gRPC	médio	concorrente	19.69	± 64.46
Go	gRPC	médio	sequencial	3.36	± 14.21
Go	gRPC	small	concorrente	3.84	± 8.55
Go	gRPC	small	sequencial	1.59	± 2.18
Go	gRPC	unit	sequencial	1.21	± 0.66
Go	REST	high	concorrente	191.02	± 687.60
Go	REST	high	sequencial	24.55	± 169.05
Go	REST	médio	concorrente	39.50	± 136.93
Go	REST	médio	sequencial	6.05	± 34.42
Go	REST	small	concorrente	5.76	± 18.15
Go	REST	small	sequencial	1.80	± 5.20
Go	REST	unit	sequencial	0.92	± 0.88
Rust	GraphQL	high	concorrente	2.64	± 28.89
Rust	GraphQL	high	sequencial	13.93	± 92.43
Rust	GraphQL	médio	concorrente	24.79	± 83.96
Rust	GraphQL	médio	sequencial	3.86	± 18.70
Rust	GraphQL	small	concorrente	4.54	± 12.27
Rust	GraphQL	small	sequencial	1.61	± 3.04
Rust	GraphQL	unit	sequencial	1.25	± 0.79
Rust	gRPC	high	concorrente	15.37	± 49.45
Rust	gRPC	high	sequencial	2.55	± 9.33
Rust	gRPC	médio	concorrente	16.46	± 52.68
Rust	gRPC	médio	sequencial	2.85	± 12.17
Rust	gRPC	small	concorrente	3.63	± 7.89
Rust	gRPC	small	sequencial	1.42	± 1.93
Rust	gRPC	unit	sequencial	1.38	± 0.77
Rust	REST	high	concorrente	180.40	± 648.17
Rust	REST	high	sequencial	23.21	± 160.80
Rust	REST	médio	concorrente	38.72	± 134.39

Continua na próxima página

Tabela 10 – Continuação

Ling.	Abord.	Volume	Modo	Média (ms)	DP
Rust	REST	médio	sequencial	6.14	± 32.96
Rust	REST	small	concorrente	5.58	± 17.30
Rust	REST	small	sequencial	1.66	± 5.00
Rust	REST	unit	sequencial	1.17	± 0.82

Fonte: Elaborado pelo autor.

APÊNDICE B - RESULTADOS COMPLETOS DOS TESTES DE THROUGHPUT

Este apêndice apresenta os resultados completos da métrica de throughput. Essa métrica representa a quantidade média de requisições processadas por segundo em cada combinação de linguagem de programação, abordagem de comunicação, volume de dados e modo de execução.

A inclusão desses dados complementares permite verificar o comportamento das APIs em diferentes perfis de carga, especialmente nos cenários unitário, pequeno e alto, que foram omitidos da análise principal para evitar excesso de tabelas no corpo do trabalho. Assim como nas demais métricas, os valores são apresentados como média e desvio padrão.

Tabela 11 - Throughput médio por volume e modo de execução

Ling.	Abord.	Volume	Modo	Média (req/s)	DP
C#	GraphQL	unit	sequencial	287.71	± 9.63
C#	REST	unit	sequencial	271.15	± 18.09
C#	gRPC	unit	sequencial	239.60	± 14.80
Go	GraphQL	unit	sequencial	595.45	± 12.55
Go	REST	unit	sequencial	793.53	± 9.09
Go	gRPC	unit	sequencial	555.40	± 23.17
Rust	GraphQL	unit	sequencial	653.55	± 5.14
Rust	REST	unit	sequencial	720.85	± 85.00
Rust	gRPC	unit	sequencial	537.22	± 25.88
C#	GraphQL	small	sequencial	247.44	± 1.43
C#	REST	small	sequencial	264.07	± 6.56
C#	gRPC	small	sequencial	286.84	± 0.12
Go	GraphQL	small	sequencial	371.84	± 4.03
Go	REST	small	sequencial	531.26	± 26.26
Go	gRPC	small	sequencial	555.89	± 14.10

Continua na próxima página

Tabela 11 – Continuação

Ling.	Abord.	Volume	Modo	Média (req/s)	DP
Rust	GraphQL	small	sequencial	510.56	± 16.25
Rust	REST	small	sequencial	578.29	± 24.27
Rust	gRPC	small	sequencial	614.35	± 20.89
C#	GraphQL	small	concorrente	492.99	± 6.17
C#	REST	small	concorrente	556.93	± 9.53
C#	gRPC	small	concorrente	661.00	± 17.58
Go	GraphQL	small	concorrente	543.65	± 2.22
Go	REST	small	concorrente	830.60	± 40.41
Go	gRPC	small	concorrente	1126.13	± 61.66
Rust	GraphQL	small	concorrente	764.45	± 31.78
Rust	REST	small	concorrente	858.60	± 34.95
Rust	gRPC	small	concorrente	1197.68	± 94.36
C#	GraphQL	médio	sequencial	141.74	± 0.23
C#	REST	médio	sequencial	148.98	± 0.62
C#	gRPC	médio	sequencial	201.48	± 7.88
Go	GraphQL	médio	sequencial	112.35	± 4.65
Go	REST	médio	sequencial	163.50	± 0.33
Go	gRPC	médio	sequencial	226.36	± 10.06
Rust	GraphQL	médio	sequencial	156.62	± 0.98
Rust	REST	médio	sequencial	161.34	± 4.10
Rust	gRPC	médio	sequencial	257.26	± 3.07
C#	GraphQL	médio	concorrente	115.88	± 3.40
C#	REST	médio	concorrente	131.02	± 1.09
C#	gRPC	médio	concorrente	209.11	± 8.15
Go	GraphQL	médio	concorrente	78.10	± 5.65
Go	REST	médio	concorrente	125.95	± 8.99
Go	gRPC	médio	concorrente	216.91	± 12.72
Rust	GraphQL	médio	concorrente	121.55	± 8.25
Rust	REST	médio	concorrente	128.07	± 6.89
Rust	gRPC	médio	concorrente	256.10	± 5.99
C#	GraphQL	high	sequencial	38.78	± 0.57
C#	REST	high	sequencial	41.28	± 0.51
C#	gRPC	high	sequencial	257.75	± 0.53
Go	GraphQL	high	sequencial	28.09	± 0.11
Go	REST	high	sequencial	40.64	± 0.44
Go	gRPC	high	sequencial	267.45	± 10.20
Rust	GraphQL	high	sequencial	38.28	± 0.10

Continua na próxima página

Tabela 11 – Continuação

Ling.	Abord.	Volume	Modo	Média (req/s)	DP
Rust	REST	high	sequencial	42.98	± 0.07
Rust	gRPC	high	sequencial	378.76	± 7.62
C#	GraphQL	high	concorrente	25.37	± 2.00
C#	REST	high	concorrente	26.66	± 0.04
C#	gRPC	high	concorrente	143.16	± 1.61
Go	REST	high	concorrente	26.12	± 0.14
Go	gRPC	high	concorrente	189.92	± 1.81
Rust	REST	high	concorrente	27.62	± 0.15
Rust	gRPC	high	concorrente	321.00	± 0.04

Fonte: Elaborado pelo autor.

APÊNDICE C - RESULTADOS COMPLETOS DOS TESTES DE CONSUMO DE CPU

Este apêndice apresenta os resultados completos relacionados ao consumo de CPU durante a execução dos testes. O consumo de CPU foi coletado com o Docker Stats e expressa o percentual médio de processamento utilizado pelas aplicações durante cada cenário experimental.

Esses dados complementam a análise de desempenho ao demonstrar o custo computacional exigido por cada combinação entre linguagem e abordagem de comunicação. Dessa forma, é possível comparar não apenas o tempo de resposta e a vazão das APIs, mas também o esforço de processamento necessário para sustentar cada carga.

Tabela 12 - Consumo médio de CPU por volume e modo de execução

Ling.	Abord.	Volume	Modo	Média (%)	DP
C#	GraphQL	unit	sequencial	0.02	± 0.00
C#	REST	unit	sequencial	5.73	± 9.86
C#	gRPC	unit	sequencial	17.23	± 1.41
Go	GraphQL	unit	sequencial	0.08	± 0.14
Go	REST	unit	sequencial	0.12	± 0.01
Go	gRPC	unit	sequencial	0.01	± 0.00
Rust	GraphQL	unit	sequencial	0.05	± 0.09
Rust	REST	unit	sequencial	0.00	± 0.00
Rust	gRPC	unit	sequencial	0.00	± 0.00
C#	GraphQL	small	sequencial	85.74	± 1.18
C#	REST	small	sequencial	76.11	± 3.30

Continua na próxima página

Tabela 12 – Continuação

Ling.	Abord.	Volume	Modo	Média (%)	DP
C#	gRPC	small	sequencial	83.75	± 7.56
Go	GraphQL	small	sequencial	12.66	± 0.07
Go	REST	small	sequencial	6.89	± 0.06
Go	gRPC	small	sequencial	9.11	± 0.22
Rust	GraphQL	small	sequencial	8.51	± 1.09
Rust	REST	small	sequencial	4.88	± 0.50
Rust	gRPC	small	sequencial	5.77	± 1.13
C#	GraphQL	small	concorrente	122.85	± 26.61
C#	REST	small	concorrente	111.03	± 6.95
C#	gRPC	small	concorrente	129.45	± 28.69
Go	GraphQL	small	concorrente	105.61	± 27.54
Go	REST	small	concorrente	15.60	± 2.53
Go	gRPC	small	concorrente	3.39	± 2.26
Rust	GraphQL	small	concorrente	23.32	± 3.18
Rust	REST	small	concorrente	7.80	± 1.45
Rust	gRPC	small	concorrente	1.33	± 2.19
C#	GraphQL	médio	sequencial	41.16	± 0.91
C#	REST	médio	sequencial	33.34	± 1.90
C#	gRPC	médio	sequencial	36.83	± 1.09
Go	GraphQL	médio	sequencial	66.91	± 1.50
Go	REST	médio	sequencial	13.70	± 0.33
Go	gRPC	médio	sequencial	18.42	± 1.44
Rust	GraphQL	médio	sequencial	15.37	± 0.17
Rust	REST	médio	sequencial	7.00	± 0.81
Rust	gRPC	médio	sequencial	9.28	± 0.36
C#	GraphQL	médio	concorrente	105.48	± 1.26
C#	REST	médio	concorrente	80.66	± 2.82
C#	gRPC	médio	concorrente	115.13	± 2.26
Go	GraphQL	médio	concorrente	273.64	± 2.30
Go	REST	médio	concorrente	65.59	± 2.43
Go	gRPC	médio	concorrente	96.71	± 3.01
Rust	GraphQL	médio	concorrente	79.78	± 3.59
Rust	REST	médio	concorrente	37.09	± 1.53
Rust	gRPC	médio	concorrente	49.49	± 3.51
C#	GraphQL	high	sequencial	32.74	± 0.30
C#	REST	high	sequencial	28.20	± 0.02
C#	gRPC	high	sequencial	150.04	± 3.00

Continua na próxima página

Tabela 12 – Continuação

Ling.	Abord.	Volume	Modo	Média (%)	DP
Go	GraphQL	high	sequencial	75.99	± 1.12
Go	REST	high	sequencial	17.68	± 0.45
Go	gRPC	high	sequencial	91.24	± 1.59
Rust	GraphQL	high	sequencial	18.48	± 0.73
Rust	REST	high	sequencial	8.07	± 0.08
Rust	gRPC	high	sequencial	51.50	± 1.35
C#	GraphQL	high	concorrente	113.70	± 6.86
C#	REST	high	concorrente	80.75	± 1.12
C#	gRPC	high	concorrente	523.62	± 9.63
Go	REST	high	concorrente	74.54	± 2.86
Go	gRPC	high	concorrente	462.01	± 3.07
Rust	REST	high	concorrente	41.12	± 1.20
Rust	gRPC	high	concorrente	395.29	± 4.64

Fonte: Elaborado pelo autor.

APÊNDICE D - RESULTADOS COMPLETOS DOS TESTES DE CONSUMO DE MEMÓRIA

Este apêndice apresenta os resultados completos referentes ao consumo de memória RAM das aplicações avaliadas. A métrica foi coletada em tempo de execução por meio do Docker Stats e indica a quantidade média de memória utilizada por cada aplicação durante os testes.

A análise dessa métrica é relevante porque permite observar a eficiência no uso de recursos computacionais, principalmente em cenários com maior volume de dados e concorrência. Com isso, os resultados de memória auxiliam na interpretação do comportamento geral das APIs, em conjunto com latência, throughput e uso de CPU.

Tabela 13 - Consumo médio de memória

Ling.	Abord.	Volume	Modo	Média	Desvio padrão
C#	GraphQL	unit	sequencial	77.05	$\pm 4.97 / -4.97$
C#	REST	unit	sequencial	105.21	$\pm 36.00 / -36.00$
C#	gRPC	unit	sequencial	65.58	$\pm 4.69 / -4.69$
Go	GraphQL	unit	sequencial	9.60	$\pm 4.36 / -4.36$
Go	REST	unit	sequencial	6.49	$\pm 3.26 / -3.26$
Go	gRPC	unit	sequencial	10.46	$\pm 5.21 / -5.21$
Rust	GraphQL	unit	sequencial	5.88	$\pm 3.14 / -3.14$

Continua na próxima página

Tabela 13 – Continuação

Ling.	Abord.	Volume	Modo	Média	Desvio padrão
Rust	REST	unit	sequencial	5.02	$\pm 2.68 / -2.68$
Rust	gRPC	unit	sequencial	5.04	$\pm 2.95 / -2.95$
C#	GraphQL	small	sequencial	152.13	$\pm 2.25 / -2.25$
C#	REST	small	sequencial	140.94	$\pm 3.97 / -3.97$
C#	gRPC	small	sequencial	120.04	$\pm 1.76 / -1.76$
Go	GraphQL	small	sequencial	13.28	$\pm 2.69 / -2.69$
Go	REST	small	sequencial	8.34	$\pm 0.63 / -0.63$
Go	gRPC	small	sequencial	8.56	$\pm 0.17 / -0.17$
Rust	GraphQL	small	sequencial	6.64	$\pm 1.47 / -1.47$
Rust	REST	small	sequencial	3.30	$\pm 0.05 / -0.05$
Rust	gRPC	small	sequencial	4.05	$\pm 1.00 / -1.00$
C#	GraphQL	small	concorrente	184.40	$\pm 0.78 / -0.78$
C#	REST	small	concorrente	164.75	$\pm 0.95 / -0.95$
C#	gRPC	small	concorrente	163.23	$\pm 9.84 / -9.84$
Go	GraphQL	small	concorrente	21.26	$\pm 0.95 / -0.95$
Go	REST	small	concorrente	8.96	$\pm 1.37 / -1.37$
Go	gRPC	small	concorrente	9.38	$\pm 0.92 / -0.92$
Rust	GraphQL	small	concorrente	21.62	$\pm 4.03 / -4.03$
Rust	REST	small	concorrente	9.98	$\pm 0.03 / -0.03$
Rust	gRPC	small	concorrente	10.77	$\pm 0.28 / -0.28$
C#	GraphQL	médio	sequencial	199.55	$\pm 4.14 / -4.14$
C#	REST	médio	sequencial	168.66	$\pm 4.01 / -4.01$
C#	gRPC	médio	sequencial	158.84	$\pm 3.22 / -3.22$
Go	GraphQL	médio	sequencial	33.45	$\pm 0.01 / -0.01$
Go	REST	médio	sequencial	17.04	$\pm 1.14 / -1.14$
Go	gRPC	médio	sequencial	13.62	$\pm 0.64 / -0.64$
Rust	GraphQL	médio	sequencial	29.16	$\pm 7.15 / -7.15$
Rust	REST	médio	sequencial	8.72	$\pm 0.87 / -0.87$
Rust	gRPC	médio	sequencial	18.00	$\pm 0.19 / -0.19$
C#	GraphQL	médio	concorrente	265.85	$\pm 1.67 / -1.67$
C#	REST	médio	concorrente	177.99	$\pm 4.54 / -4.54$
C#	gRPC	médio	concorrente	208.31	$\pm 4.93 / -4.93$
Go	GraphQL	médio	concorrente	149.63	$\pm 1.95 / -1.95$
Go	REST	médio	concorrente	25.42	$\pm 0.11 / -0.11$
Go	gRPC	médio	concorrente	20.05	$\pm 0.97 / -0.97$
Rust	GraphQL	médio	concorrente	127.27	$\pm 15.62 / -15.62$
Rust	REST	médio	concorrente	30.95	$\pm 3.73 / -3.73$

Continua na próxima página

Tabela 13 – Continuação

Ling.	Abord.	Volume	Modo	Média	Desvio padrão
Rust	gRPC	médio	concorrente	44.12	$\pm 1.28 / -1.28$
C#	GraphQL	high	sequencial	400.48	$\pm 3.24 / -3.24$
C#	REST	high	sequencial	332.46	$\pm 7.33 / -7.33$
C#	gRPC	high	sequencial	256.10	$\pm 4.00 / -4.00$
Go	GraphQL	high	sequencial	189.32	$\pm 3.04 / -3.04$
Go	REST	high	sequencial	62.98	$\pm 2.81 / -2.81$
Go	gRPC	high	sequencial	42.29	$\pm 3.01 / -3.01$
Rust	GraphQL	high	sequencial	342.12	$\pm 5.46 / -5.46$
Rust	REST	high	sequencial	69.00	$\pm 19.49 / -19.49$
Rust	gRPC	high	sequencial	85.24	$\pm 3.90 / -3.90$
C#	GraphQL	high	concorrente	428.83	$\pm 5.29 / -5.29$
C#	REST	high	concorrente	364.31	$\pm 16.87 / -16.87$
C#	gRPC	high	concorrente	431.69	$\pm 3.52 / -3.52$
Go	REST	high	concorrente	99.81	$\pm 0.40 / -0.40$
Go	gRPC	high	concorrente	167.96	$\pm 1.93 / -1.93$
Rust	REST	high	concorrente	225.36	$\pm 0.04 / -0.04$
Rust	gRPC	high	concorrente	314.77	$\pm 0.38 / -0.38$

Fonte: Elaborado pelo autor.