# Comparativo de comunicação de microsserviços desenvolvidos em múltiplas linguagens de programação

**Lucas Martins Chaves**[1], **Marcos José Brusso**[2]

[1]Instituto de Tecnologia – Universidade de Passo Fundo (UPF)
Caixa Postal 15.064 – 91.501-970 – Porto Alegre – RS – Brazil

{183165,brusso}@upf.br

***Abstract.*** *How microservices communicate is fundamental to the overall performance of a system. Therefore, there are various protocols and communication methods available in different programming languages. With this in mind, this work presents a testing model to identify performance behaviors among different communication methods applied to multiple languages. For this purpose, Elastic Stack observability resources are used to collect data and create visualizations of load tests generated with the K6 tool. To be subjected to the tests, three applications were developed in Go, JavaScript, and Python, implementing REST, gRPC, and GraphQL. The results show overall better performance in Go, superior performance of GraphQL for insertions, and similar behavior between GraphQL and REST, with a slight advantage for REST in handling large volumes of data.*

***Resumo.*** *Como os microsserviços se comunicam é fundamental para o desempenho geral de um sistema. Por isso, há diversos protocolos e formas disponíveis de comunicação em diferentes linguagens de programação. Com isso em mente, este trabalho apresenta um modelo de teste para identificar comportamentos de desempenho entre diferentes métodos de comunicação, aplicados a múltiplas linguagens. Para isso, utilizam-se recursos de observabilidade da Elastic Stack para coletar dados e criar visualizações dos testes de carga gerados com a ferramenta K6. Para serem submetidas aos testes, desenvolveram-se três aplicações em Go, JavaScript e Python, implementando REST, gRPC e GraphQL. Nos resultados, nota-se um maior desempenho geral no Go, do GraphQL nas inserções e um comportamento parecido entre GraphQL e REST, com leve vantagem do REST em grande volume de dados.*

## 1. Introdução

Microsserviços podem ser entendidos como aplicações que possuem apenas uma responsabilidade, como, por exemplo, a implementação de um único requisito funcional. Os principais benefícios dessa arquitetura são escalabilidade, melhor manutenção e liberdade técnica. Assim, ela se torna uma preferência em relação ao modelo monolítico popularmente usado [Thönes 2015].

Contudo, a forma de comunicação e a linguagem de programação podem interferir diretamente no desempenho geral da arquitetura. Por isso, há diversas opções de soluções para comunicação, como REST [Wilde & Pautasso 2011], gRPC [Yong 2023] e GraphQL

[Facebook 2012], além de linguagens como Go [Google 2024], JavaScript [Mozilla 2024] e Python [Foundation 2024].

Visando uma análise desse comportamento, utiliza-se o conceito de observabilidade, que se refere ao entendimento do estado interno do sistema com base no estudo de suas saídas externas. Nas aplicações, a observabilidade se resume a três pilares: logs, métricas e traces. Sua principal importância está na contribuição para a avaliação, monitoramento e melhoria de sistemas de tecnologia distribuídos [Elastic 2024].

Para dar suporte à observabilidade, existem diversas ferramentas disponíveis no mercado. Uma delas é o conjunto de soluções denominado Elastic Stack, que possibilita coletar dados de diferentes origens e manipulá-los para a criação de visualizações, alertas e outros recursos [Shaji & George 2024].

Com essas informações, este trabalho tem como objetivo criar um modelo de teste que permita comparativos de desempenho entre microsserviços desenvolvidos com diferentes linguagens de programação e métodos de comunicação. Para isso, serão criados três microsserviços em diferentes linguagens, sendo elas Go, JavaScript e Python, todos implementando três interfaces de comunicação: REST, gRPC e GraphQL. Além disso, para a coleta de dados e criação dos gráficos, serão usadas ferramentas da Elastic Stack, como Elasticsearch para observabilidade e Kibana para as visualizações. Para a execução dos testes, será utilizada a ferramenta K6 para a geração de cargas de teste para todas as aplicações.

Para tanto, o documento está organizado da seguinte forma: a Seção 2 apresenta diferentes análises e comparativos entre métodos de comunicação usando REST e gRPC em diversos contextos. A Seção 3 descreve as ferramentas utilizadas, o desenvolvimento do código, a estruturação do ambiente e o fluxo de teste. A Seção 4 detalha os testes realizados e seus resultados, seguido por uma análise dos mesmos. Por fim, a Seção 5 expõe as conclusões do trabalho.

## 2. Trabalhos Relacionados

Nesta seção, são apresentados, de forma breve, alguns trabalhos com focos em comunicação de microsserviços.

### 2.1. Efficiency of REST and gRPC in Communication Tasks in Microservice-Based Ecosystems

O estudo de [Bolanowski et al. 2022] aborda um comparativo de comunicação entre microsserviços com interfaces REST e gRPC desenvolvidas em .NET. Foram realizados diversos cenários de testes, considerando diferentes tipos de dados, como números inteiros, textos longos e curtos, arrays de inteiros e arquivos de texto e PDF.

Como resultado, foi perceptível que o REST possui certa vantagem em recursos de pequeno tamanho e textuais. Por sua vez, o gRPC aparenta ser mais adequado em aplicações que necessitam de um fluxo muito frequente entre microsserviços. Além disso, em transferências de arquivos grandes e comunicação criptografada, o gRPC apresentou melhor desempenho.

### 2.2. Comparing REST, SOAP, Socket, and gRPC in Computation Offloading of Mobile Applications: An Energy Cost Analysis

Na abordagem de [Chamas et al. 2017], é proposta uma análise do consumo de energia de dispositivos móveis com base no método de comunicação utilizado, como gRPC, REST, SOAP e Socket. Para isso, foi desenvolvida uma aplicação em Java, na qual foram implementados algoritmos clássicos de ordenação com um número variado de entradas.

Os resultados obtidos demonstram que o REST teve, na maioria dos casos, menor consumo de energia. Por outro lado, o gRPC não apresentou nenhuma economia significativa em comparação com os demais.

### 2.3. Performance Comparison of Programming Interfaces: REST API, GraphQL, and gRPC

[Śliwa & Pańczyk 2021] conduz uma análise comparativa de desempenho entre REST, GraphQL e gRPC. Para isso, foram desenvolvidas aplicações para todas as interfaces em .NET, utilizando a ferramenta k6 para testes de carga.

Através dessa análise, constatou-se que o gRPC não teve um bom desempenho na realização de consultas simples, em comparação com as demais interfaces. No entanto, demonstrou uma forte vantagem ao baixar informações do servidor.

Tendo em vista esses estudos, nota-se que há uma busca por um melhor entendimento do desempenho das aplicações, mas geralmente com foco em uma única linguagem específica. Além disso, os trabalhos não apresentam formas de manipular e criar visualizações com os resultados gerados.

Com base nessas informações, este estudo propõe um modelo de teste que possibilite a integração de várias aplicações e facilite a criação de visualizações do sistema. Além disso, desenvolvem-se e implementam-se microsserviços em diferentes linguagens de programação e interfaces de comunicação, as quais serão comparadas entre si.

## 3. Materiais e Métodos

Nesta seção, são explicados de forma breve as ferramentas e metodologias utilizadas na realização do projeto.

### 3.1. Ferramentas

Nestas subseções, são descritas as ferramentas usadas para desenvolver os microsserviços, geração de métricas e criação dos gráficos.

### 3.1.1. Arquitetura

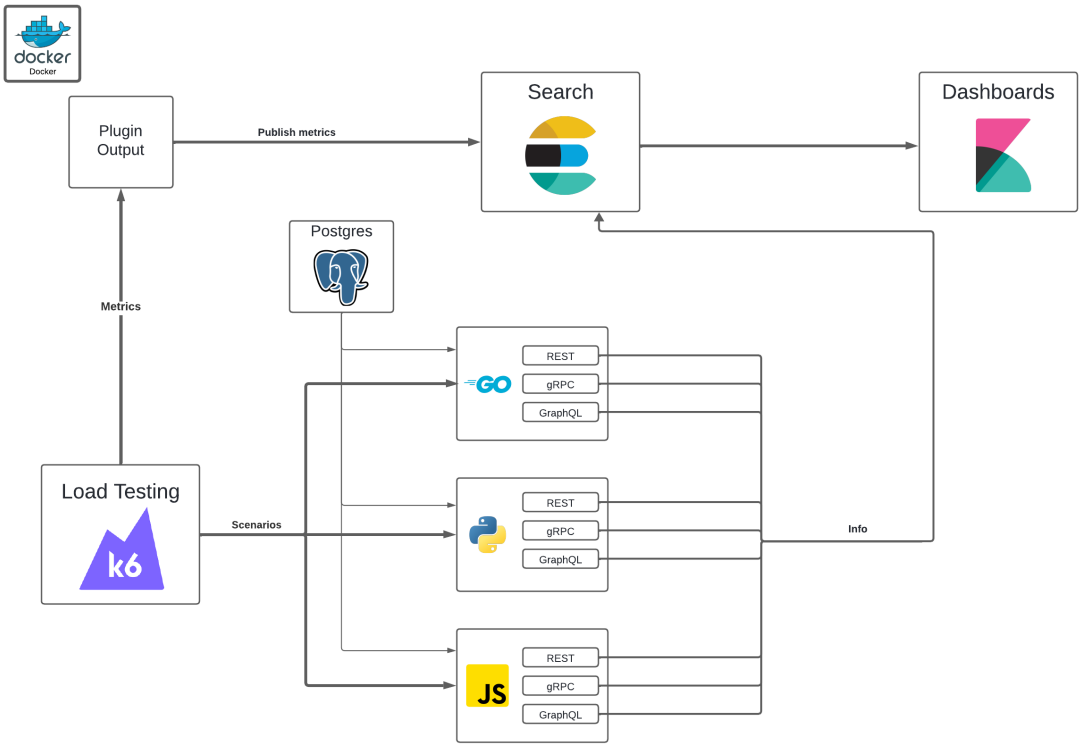


**Figura 1. Diagrama da arquitetura da solução proposta**

A Figura 1 apresenta a arquitetura proposta e a integração de seus componentes, desde os testes até as visualizações. Cada ferramenta e fluxo será detalhado a seguir neste trabalho.

### 3.1.2. K6

Para criação e execução de testes de carga, utilizou-se a ferramenta K6, desenvolvida pela Grafana Labs, que, por sua vez, oferece a criação de cenários de teste por meio de scripts JavaScript. Além disso, disponibiliza a opção de testar diferentes tipos de protocolos, dentre eles HTTP, WebSockets, gRPC, entre outros [Labs 2007].

Neste projeto, foi escolhido o K6 devido à sua versatilidade com todos os métodos de comunicação propostos, além de promover facilidade no desenvolvimento dos cenários de teste.

### 3.1.3. Elasticsearch

Para a manipulação dos dados, utilizou-se o Elasticsearch, um mecanismo de pesquisa e análise baseado no Apache Lucene. Além de ser open source, o Elasticsearch oferece diversas ferramentas adicionais para visualizações, segurança, logs, entre outros. Esse conjunto de ferramentas é conhecido como Elastic Stack [Shaji George, 2024].

Neste projeto, o Elasticsearch foi escolhido por ser open source e por seus recursos de observabilidade, que facilitam as análises propostas.

### 3.1.4. Kibana

Para a criação de gráficos, utilizou-se o Kibana, que possibilita a criação de visualizações das informações indexadas para o Elasticsearch de diversas formas [Shaji & George 2024]. Um dos principais recursos é chamado de Lens; nele, a construção das visualizações é feita arrastando e soltando informações, além de sugerir gráficos diferentes.

Neste projeto, optou-se pelo Kibana devido à sua integração com o Elasticsearch e pelas ferramentas que abstraem a complexidade da criação dos dashboards.

## 3.2. Código

As aplicações foram desenvolvidas em três linguagens de programação: Go, JavaScript e Python. Para isso, cada implementação segue o padrão arquitetônico de camadas Clean Architecture, criado por [Martin 2012]. Uma das principais características dessa arquitetura é a independência de frameworks, o que possibilitou o desenvolvimento de um único código para os diferentes protocolos de comunicação, preservando o núcleo lógico da aplicação.

Os dados utilizados para as requisições têm como base duas estruturas denominadas *Pet* e *Person*, geradas pela biblioteca Gofakeit [Voelker 2023]. O intuito principal consiste em uma estrutura com um grande volume de informações, sendo a *Person* composta por um campo com textos mais extensos, e a *Pet* com um volume menor, para avaliar a existência de divergências no comportamento dos endpoints. Cada estrutura possui os seguintes campos:

- **Person**
  - ID - (Inteiro)
  - CreatedAt - (Data)
  - UpdatedAt - (Data)
  - DeletedAt - (Data)
  - FirstName - (String)
  - LastName - (String)
  - Email - (String)
  - Age - (String)
  - Phone - (String)
  - Gender - (String)
  - Biography - (String, consiste em um lorem ipsum de cinco parágrafos, cada um com cinco sentenças e cada uma com sete palavras)
- **Pet**
  - ID - (Inteiro)
  - CreatedAt - (Data)
  - UpdatedAt - (Data)
  - DeletedAt - (Data)
  - Name - (String)

- Age - (Inteiro)
- Type - (String)

Cada estrutura terá dois endpoints, que serão implementados para cada protocolo, sendo um para criação e outro para listagem.

Para acessar o banco de dados, decidiu-se como padrão a utilização de ORMs, sendo esses o GORM [GORM 2024] no Go, Sequelize [Sequelize 2024] no JavaScript e SQLAlchemy [SQLAlchemy 2024] no Python.

### 3.3. Ambiente

Buscando facilitar questões de ambiente e isolar as dependências de cada microsserviço, foram estruturados arquivos para a construção de containers Docker. Cada linguagem possui um arquivo chamado Dockerfile e um docker-compose.yml [Reis et al. 2022]. Com o Compose, uma imagem será montada para cada protocolo, que, por sua vez, é identificado pelo argumento passado, como mostra a Figura 2.

```yaml
grpc-go-tcc:
  container_name: grpc-go-tcc
  build: ./go
  command:
    - "grpc"
  env_file:
    - ./go/.env
  environment:
    - ELASTIC_APM_SERVICE_NAME=TCC_GO_GRPC
    - ELASTIC_APM_SERVER_URL=http://apm:8200
    - ELASTIC_APM_ENVIRONMENT=development
    - ELASTIC_APM_LOG_LEVEL=debug
    - ELASTIC_APM_LOG_FILE=stderr
  depends_on:
    - postgres
  ports:
    - "50051:50051"
  networks:
    - observability
```

```yaml
graph-go-tcc:
  container_name: graph-go-tcc
  build: ./go
  command:
    - "graphql"
  env_file:
    - ./go/.env
  environment:
    - ELASTIC_APM_SERVICE_NAME=TCC_GO_GRAPH
    - ELASTIC_APM_SERVER_URL=http://apm:8200
    - ELASTIC_APM_ENVIRONMENT=development
    - ELASTIC_APM_LOG_LEVEL=debug
    - ELASTIC_APM_LOG_FILE=stderr
  depends_on:
    - postgres
  ports:
    - "4000:4000"
  networks:
    - observability
```

```yaml
rest-go-tcc:
  container_name: rest-go-tcc
  build: ./go
  command:
    - "rest"
  env_file:
    - ./go/.env
  environment:
    - ELASTIC_APM_SERVICE_NAME=TCC_GO_REST
    - ELASTIC_APM_SERVER_URL=http://apm:8200
    - ELASTIC_APM_ENVIRONMENT=development
    - ELASTIC_APM_LOG_LEVEL=debug
    - ELASTIC_APM_LOG_FILE=stderr
  ports:
    - "8080:8080"
  networks:
    - observability
```

**Figura 2. Docker-compose.yml dos microsserviços em Go**

Além disso, um banco de dados SQL Postgres [Group 2024] também é criado por imagem Docker para persistência dos dados em um único local para todas as implementações.

Para suportar o ambiente, foram alocadas quatro instâncias EC2 da AWS, sendo três t3.small e uma t2.medium [AWS 2024], com as seguintes especificações:

- **t3.small**
  - CPU: Intel(R) Xeon(R) Platinum 8259CL CPU @ 2.50GHz
  - vCPU: 2
  - Memória: 2 GiB
  - Armazenamento: 8 GiB EBS
- **t2.medium**
  - CPU: Intel(R) Xeon(R) CPU E5-2686 v4 @ 2.30GHz
  - vCPU: 2
  - Memória: 4 GiB
  - Armazenamento: 16 GiB EBS

As instâncias t3.small comportam as aplicações, uma para cada linguagem. Enquanto a t2.medium fica responsável por todas as tecnologias do Elastic Stack, o K6 e o banco de dados Postgres.

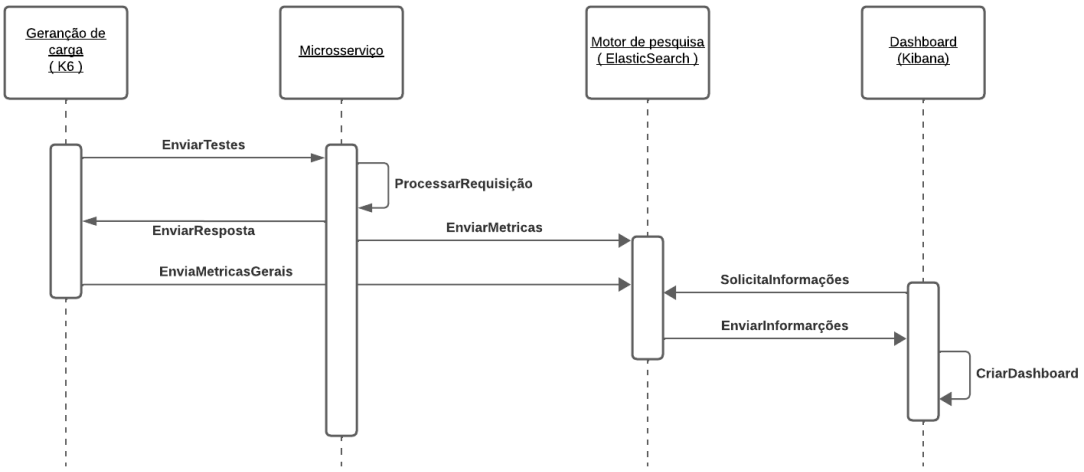


**Figura 3. Diagrama de sequência dos testes**

Nesse fluxo, o K6 envia os testes para os microsserviços, que executam sua lógica. Em seguida, com o auxílio de um plugin, o K6 e os microsserviços inserem os resultados no Elasticsearch. Por fim, o Kibana acessa as informações armazenadas no Elasticsearch para estruturar os gráficos e apresentar os resultados. Na sequência será aplicado cada ferramenta utilizada, como demonstra o fluxo de teste na Figura 3.

### 3.4. Testes

Os testes contemplam um cenário específico, no qual o comportamento proposto será de dez usuários virtuais realizando uma requisição por segundo durante sessenta segundos. Ademais, duas tags personalizadas fazem referência a qual aplicação e qual endpoint estão recebendo o teste. Esses parâmetros podem ser passados para o K6, permitindo a filtragem nas visualizações do Kibana. A Figura 4 demonstra um exemplo de como é a passagem dos parâmetros dentro do arquivo de teste.

```
export const options :{…} = {
    vus: 10,
    duration: '1m',
    tags : {
        app: "GO_REST",
        route: "ListPets"
    }
}
```

**Figura 4. Parâmetros de testes do K6**

A Figura 5 mostra como são realizados os filtros da visualização no Kibana.

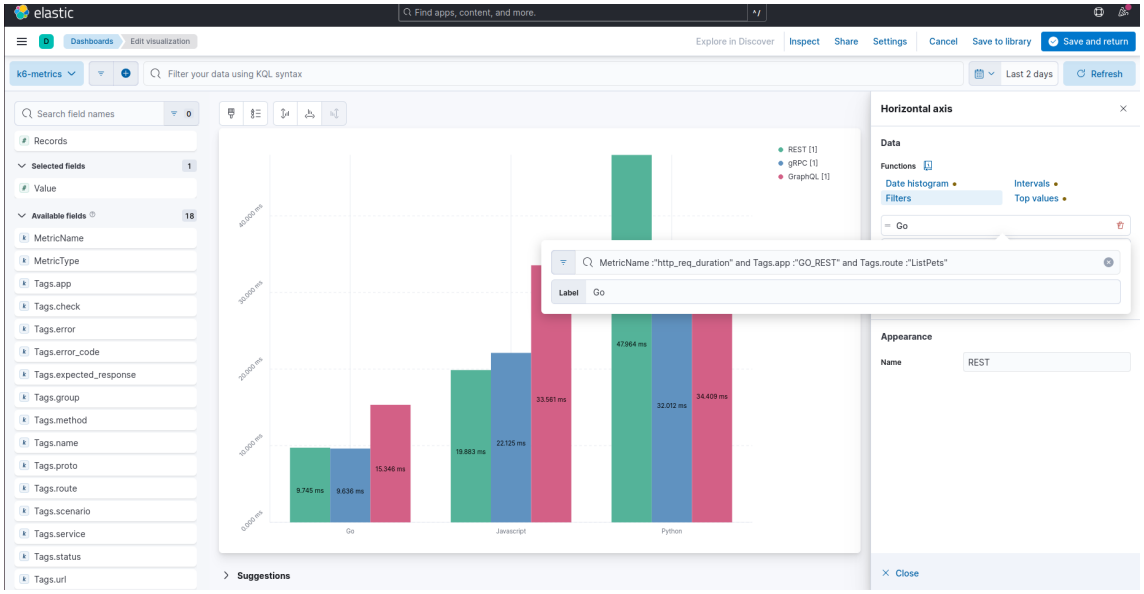


**Figura 5. Criação da visualização com Kibana**

## 4. Resultados e Discussões

Esta seção demonstra os resultados obtidos dos testes para todas as aplicações e propõe uma discussão sobre possíveis conclusões baseadas nos gráficos.

### 4.1. Teste e Gráficos

Cada gráfico representa o teste de um endpoint em todas as linguagens. O Go é o primeiro conjunto de barras verticais, seguido pelo JavaScript e, por último, o Python. Cada barra possui uma cor específica que representa um método de comunicação: verde para REST, azul para gRPC e rosa para GraphQL. Os números apresentados correspondem ao tempo de resposta das requisições, medido em milissegundos. Quanto menor o número, melhor o desempenho.

Os testes de carga foram feitos inicialmente para a listagem dos ”Pets”, sendo executados para dez, cem e mil objetos, com os parâmetros previamente estabelecidos. As Figuras 6, 7 e 8 representam esses testes, respectivamente.

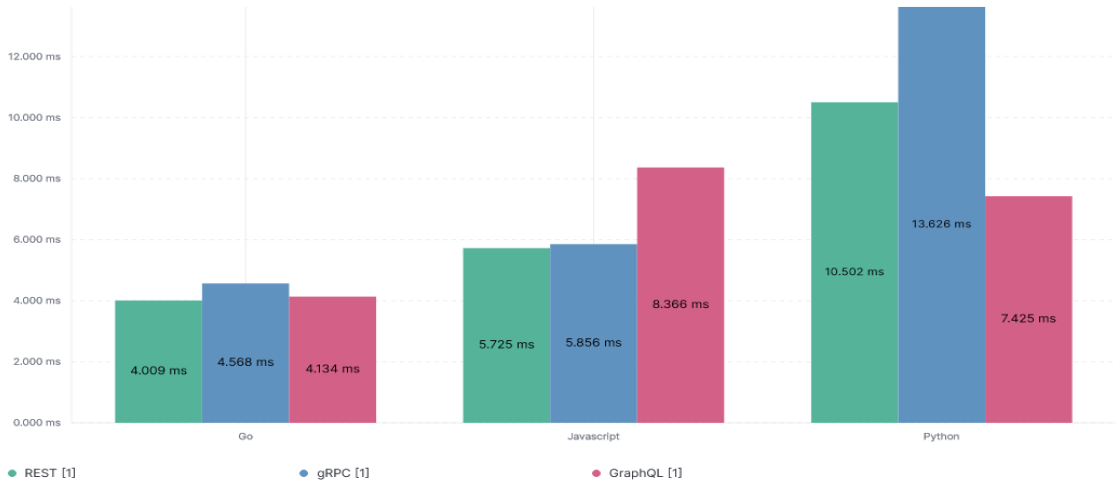


**Figura 6. Gráfico sobre listagem de 10 ”Pets”**

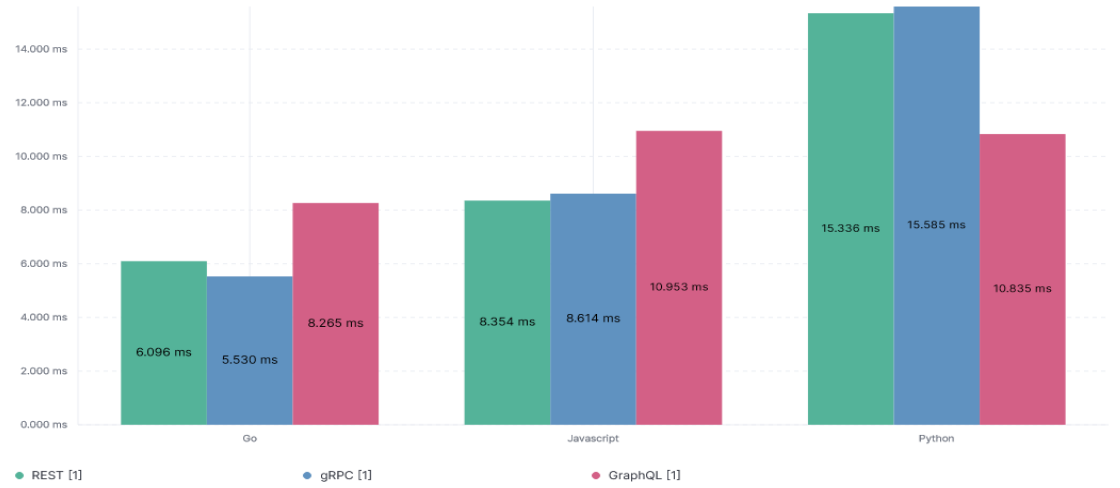


**Figura 7. Gráfico sobre listagem de 100 ”Pets”**

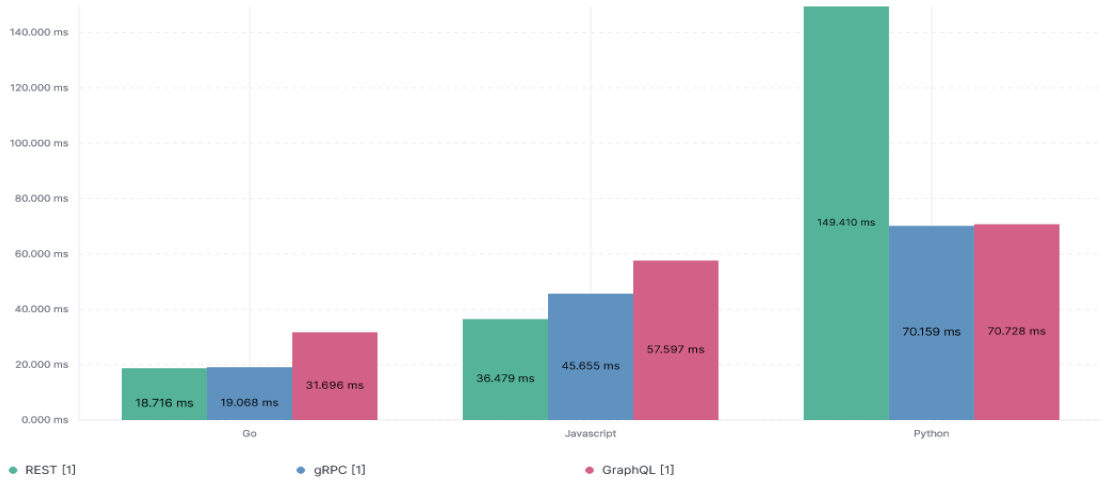


**Figura 8. Gráfico sobre listagem de 1000 ”Pets”**

Para um maior fluxo de dados, os testes dos endpoints para a listagem de ”People”foram executados, assim como os ”Pets”, possuindo os mesmos parâmetros citados anteriormente. Os testes foram executados com dez, cem e mil objetos no retorno, como demonstram as Figuras 9, 10 e 11, respectivamente.

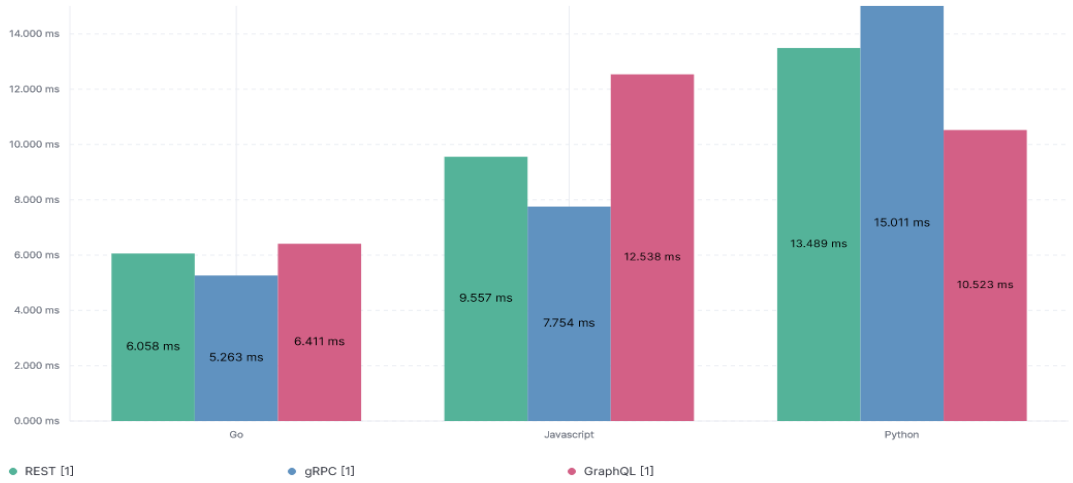


**Figura 9. Gráfico sobre listagem de 10 ”People”**

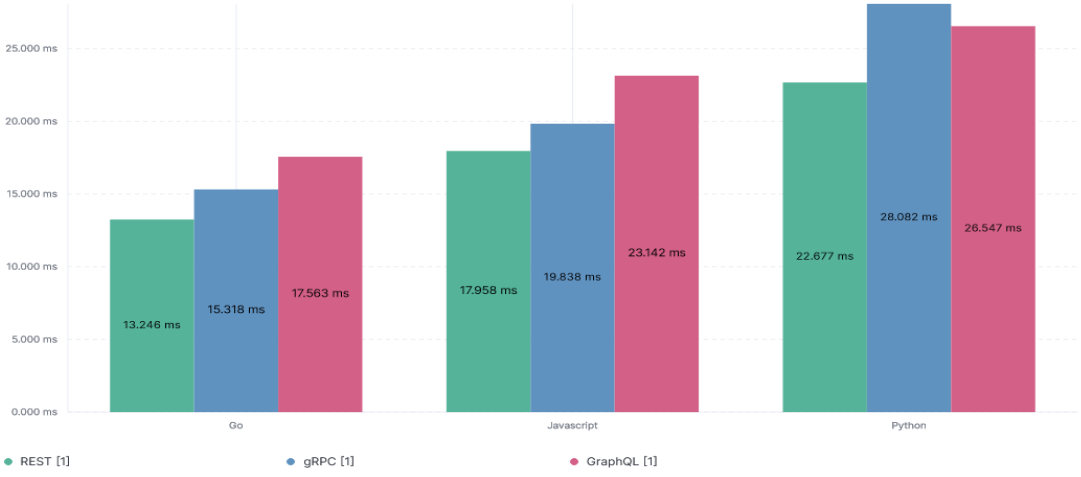


**Figura 10. Gráfico sobre listagem de 100 ”People”**

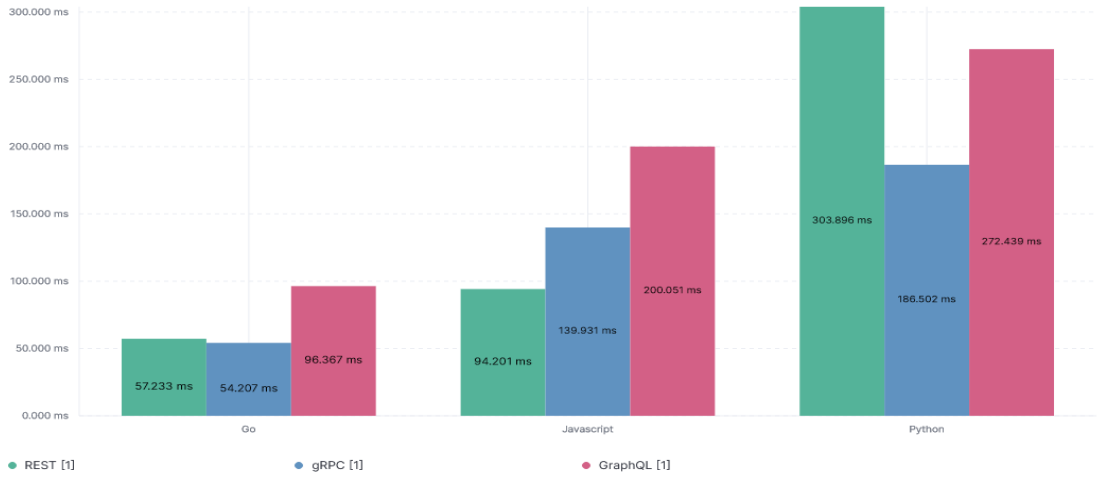


**Figura 11. Gráfico sobre listagem de 1000 ”People”**

Por fim, os testes dos endpoints de criação foram gerados com os mesmos parâmetros dos testes anteriores, porém com o princípio de inserir apenas um objeto, ou seja, de forma unitária, e receber como retorno o objeto inserido no banco. As Figuras 12 e 13 demonstram os resultados da criação de ”Pets”e ”People”, respectivamente.

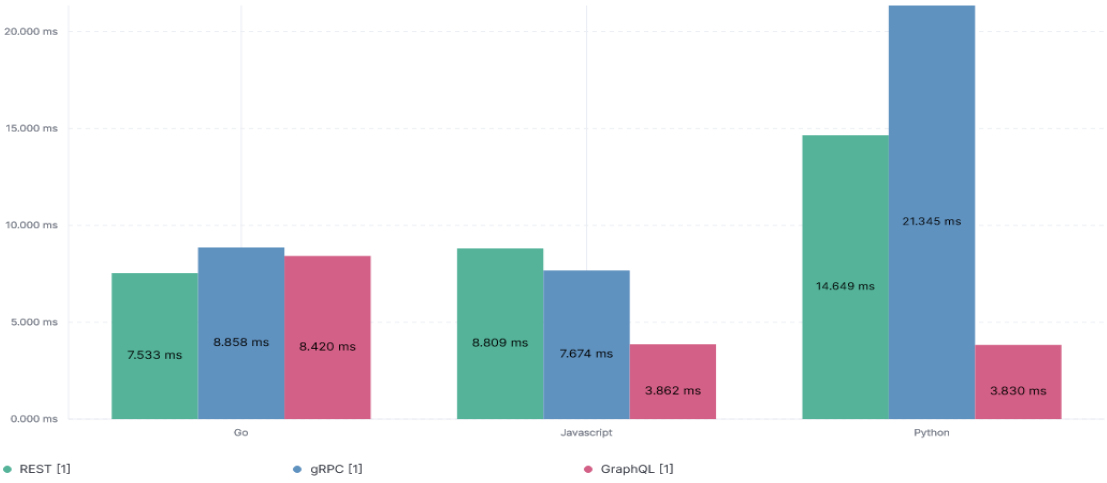


**Figura 12. Gráfico sobre criação de ”Pets”**

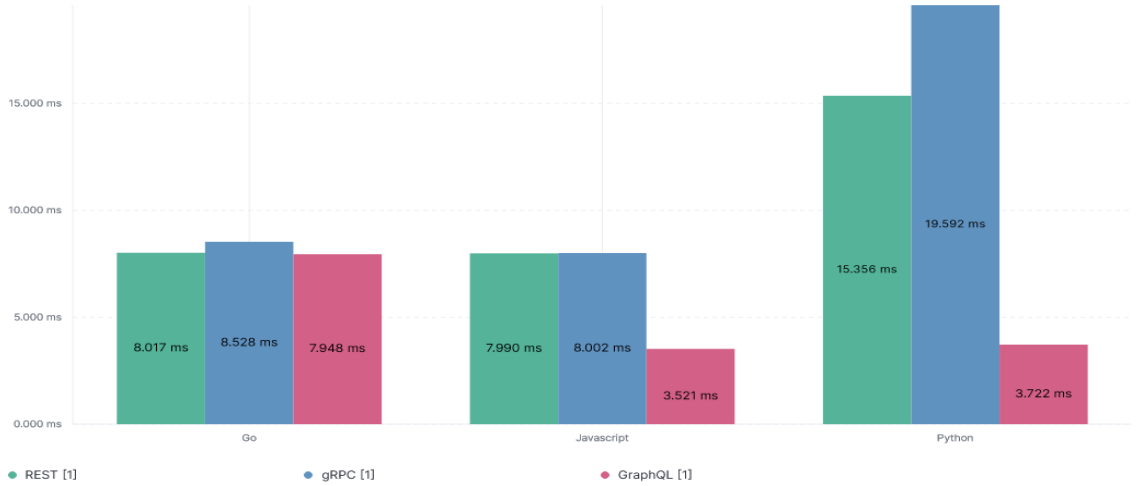


**Figura 13. Gráfico sobre criação de ”People”**

## 4.2. Avaliações

Com a apresentação dos gráficos, pode-se destacar, nas listagens, a influência das linguagens, tendo Go com o melhor desempenho, seguido por Javascript e, por último, Python. Além disso, pode-se concluir que, apesar das diferentes formas de comunicação, a linguagem de programação é o fator que mais afeta o desempenho do microsserviço.

Nas inserções, nota-se um desempenho muito superior do GraphQL, tanto em pequenos quanto em grandes volumes de dados, no Javascript e Python.

Com o aumento da carga, a implementação de gRPC em Python apresenta um aumento drástico de desempenho nas cargas com mil objetos. Enquanto isso, REST e gRPC têm comportamentos parecidos em cargas de baixo volume, como mostrado em ”Pet”. Contudo, em volumes maiores, REST tem melhores resultados à medida que a quantidade de objetos aumenta.

É importante notar que os comportamentos de uma forma de comunicação podem variar dependendo da linguagem. Uma das possíveis causas pode ser as diferentes bibliotecas e implementações, nas quais, com exceção do gRPC (que tem um grande suporte a linguagens), existem diversos autores e métodos diferentes entre si.

# 5. Conclusão

Este trabalho apresentou o desenvolvimento de três aplicações em linguagens diferentes, as quais possuem interfaces para três métodos de comunicação. Com isso, através de ferramentas de observabilidade, tornou-se possível integrar informações de desempenho das aplicações, filtrar e gerar gráficos, visando o entendimento dos resultados de forma visual e assertiva.

As aplicações foram submetidas a testes de carga, obedecendo a um cenário específico de teste, com diferentes quantidades de dados. Dessa forma, possibilitou-se avaliar os diferentes comportamentos em cada condição variável.

A principal contribuição deste trabalho é demonstrar um modelo de avaliação de desempenho de aplicações de forma mais prática, utilizando apenas ferramentas Open

Source e gratuitas. Através disso, evidencia-se a utilidade do modelo para empresas públicas ou privadas de qualquer tamanho, desenvolvedores, pesquisadores e estudantes que busquem testar e compreender melhor o desempenho geral das suas aplicações.

Como trabalhos futuros, pretende-se testar novos cenários, incluindo outras linguagens de programação como Java e Rust, além de outras formas de comunicação, por exemplo, o uso de streaming e WebSocket. Também pretende-se ampliar as ferramentas disponíveis na Elastic Stack, obtendo mais informações, como as do ambiente, e possibilitando o cruzamento com as da aplicação.

Além disso, sugere-se a exploração do modelo para outros testes, além dos comparativos entre métodos de comunicação. Ademais, pode ser exploradas outras métricas, como percentis do tempo de requisição, quantidade de dados enviados e recebidos, tempo para estabelecer a conexão, entre outros.

## 6. Agradecimentos

Agradeço aos que contribuíram de alguma forma para realização desse trabalho, em especial:

- Ao meu orientador, Professor Marcos Brusso pelo apoio e conhecimento;
- Ao Arthur Jesus, pelo companheirismo ao longo da formação;
- Ao Kauê Bortolotti, pelo conhecimento de observabilidade e auxílio na construção dos gráficos;
- As demais pessoas da minha rede de apoio, pelo cuidado e incentivo em toda a jornada.

## Referências

AWS (2024). Tipos de instâncias do amazon ec2. `https://www.elastic.co/pt/what-is/observability`. Accessed: 18-11-2024.

Bolanowski, M., Żak, K., Paszkiewicz, A., Ganzha, M., Paprzycki, M., Sowiński, P., Lacalle, I., & Palau, C. E. (2022). Eficiency of rest and grpc realizing communication tasks in microservice-based ecosystems. *arXiv preprint arXiv:2208.00682*.

Chamas, C. L., Cordeiro, D., & Eler, M. M. (2017). Comparing rest, soap, socket and grpc in computation offloading of mobile applications: An energy cost analysis. In *2017 IEEE 9th Latin-American Conference on Communications (LATINCOM)*, pages 1–6.

Elastic (2024). O que é a observabilidade? `https://www.elastic.co/pt/what-is/observability`. Accessed: 18-11-2024.

Facebook (2012). Introduction to graphql. `https://graphql.org/learn/`. Accessed: 18-11-2024.

Foundation, P. S. (2024). Python. `https://www.python.org`. Accessed: 18-11-2024.

Google (2024). Go. `https://go.dev`. Accessed: 18-11-2024.

GORM (2024). The fantastic orm library for golang. `https://gorm.io/index.html`. Accessed: 18-11-2024.

Group, T. P. G. D. (2024). Postgresql: The world's most advanced open source relational database. https://www.postgresql.org/. Accessed: 28-11-2024.

Labs, G. (2007). elasticsearch. https://github.com/grafana/k6. Accessed: 11-11-2024.

Martin, R. C. (2012). The clean architecture. https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html. Accessed: 11-11-2024.

Mozilla (2024). Javascript. https://developer.mozilla.org/pt-BR/docs/Web/JavaScript. Accessed: 18-11-2024.

Reis, D., Piedade, B., Correia, F. F., Dias, J. P., & Aguiar, A. (2022). Developing docker and docker-compose specifications: A developers' survey. *IEEE Access*, 10:2318–2329.

Sequelize (2024). Sequelize. https://sequelize.org. Accessed: 18-11-2024.

Shaji, A. S. & George, M. M. (2024). Elastic stack: A comprehensive overview. In *2024 IEEE Recent Advances in Intelligent Computational Systems (RAICS)*, pages 1–5.

Śliwa, M. & Pańczyk, B. (2021). Performance comparison of programming interfaces on the example of rest api, graphql and grpc. *Journal of Computer Sciences Institute*, 21:356–361.

SQLAlchemy (2024). Sqlalchemy orm. https://docs.sqlalchemy.org/en/20/orm. Accessed: 18-11-2024.

Thönes, J. (2015). Microservices. *IEEE software*, 32(1):116–116.

Voelker, B. (2023). Gofakeit. https://github.com/brianvoe/gofakeit. Accessed: 11-11-2024.

Wilde, E. & Pautasso, C. (2011). *REST: from research to practice*. Springer Science & Business Media.

Yong, T. Z. (2023). Introduction to grpc. https://grpc.io/docs/what-is-grpc/introduction. Accessed: 09-04-2024.