Relatório Final: Análise Comparativa de Performance entre GraphQL e REST

Carlos Henrique Neimar - João Victor Temponi
Laboratório de Experimentação de Software
Engenharia de Software - PUC Minas

4 de dezembro de 2025

Resumo

Este relatório apresenta os resultados de um experimento controlado realizado para comparar as arquiteturas de API REST e GraphQL. O estudo avaliou o desempenho de ambas as tecnologias em cenários de leitura simples, complexa e listagem massiva (N+1), utilizando implementações em Node.js e Python. As métricas analisadas foram o tempo de resposta e o tamanho do payload. Os resultados demonstram uma vantagem significativa do GraphQL na redução de tráfego de rede, enquanto o desempenho de tempo variou conforme a linguagem de programação e a complexidade da consulta.

1 Introdução

A arquitetura REST tem sido o padrão da indústria para APIs web por muitos anos. No entanto, o surgimento do GraphQL, proposto pelo Facebook, prometeu resolver problemas clássicos do REST, como o *over-fetching* (buscar dados desnecessários) e o *under-fetching* (necessidade de múltiplas requisições).

O objetivo deste laboratório é validar quantitativamente essas promessas através de um experimento controlado. Para guiar o estudo, foram definidas duas Questões de Pesquisa (RQs):

- **RQ 01:** Respostas às consultas GraphQL são mais rápidas que respostas às consultas REST?
- **RQ 02:** Respostas às consultas GraphQL têm tamanho menor que respostas às consultas REST?

2 Desenho do Experimento

O experimento foi desenhado seguindo os princípios de experimentação de software. Abaixo são detalhados os componentes fundamentais do planejamento.

A. Hipóteses Nula e Alternativa

	RQ 01 (Tempo)	**RQ 02 (Tamanho)**
H_0	Não há diferença significativa no tempo de resposta.	Não há diferença significativa no tamanho das respostas.
H_1	GraphQL apresenta tempos de resposta menores.	GraphQL apresenta respostas com tamanho menor.

Tabela 1: Definição das Hipóteses

B. Variáveis

- **Variáveis Dependentes:**
  1. Tempo de resposta (em milissegundos) medido do lado do cliente.
  2. Tamanho do *response body* (em bytes).
- **Variáveis Independentes:**
  1. Tipo de Arquitetura de API (Níveis: `REST`, `GraphQL`).
  2. Linguagem de Programação (Níveis: `JavaScript/Node.js`, `Python`).

C. Tratamentos

Foram aplicados quatro tratamentos distintos para cruzar as variáveis independentes:

1. API implementada em GraphQL com JavaScript (Node.js).
2. API implementada em REST com JavaScript (Node.js).
3. API implementada em GraphQL com Python (Flask).
4. API implementada em REST com Python (Flask).

D. Objetos Experimentais

Os objetos consistem em duas aplicações back-end que implementam as duas arquiteturas e acessam uma base de dados comum. A massa de dados foi gerada sinteticamente utilizando a biblioteca *Faker*, totalizando **100.000 usuários** registrados, onde cada usuário possui múltiplos posts associados, garantindo volume de dados suficiente para validar os testes de carga e desempenho.

E. Tipo de Projeto Experimental

O estudo caracteriza-se como um **experimento controlado** *in vitro* com desenho fatorial. O objetivo é analisar o efeito principal de cada variável independente (Tipo de API e Linguagem) e a interação entre elas nas métricas de desempenho.

F. Quantidade de Medições

Para garantir a significância estatística e mitigar flutuações momentâneas do sistema operacional, foi definida uma bateria de testes com:

- **100 repetições** para cada cenário de teste em cada um dos 4 tratamentos.
- Execução de *warm-up* (aquecimento) prévio para eliminar a latência inicial de conexão e cache.

G. Ameaças à Validade

- **Validade Interna:** A execução de processos concorrentes no sistema operacional pode afetar o tempo de resposta. *Mitigação:* O ambiente foi isolado e os testes executados sequencialmente.
- **Validade de Construto:** A qualidade da implementação do código pode beneficiar uma tecnologia. *Mitigação:* Utilizou-se bibliotecas padrão de mercado (Express/Apollo para JS e Flask/Graphene para Python) com lógica de negócio idêntica.
- **Validade Externa:** Os resultados em ambiente local (*localhost*) podem não refletir latências de rede reais (Internet). *Mitigação:* O foco do estudo é a eficiência do protocolo e processamento, sendo a latência de rede uma constante externa neste modelo.

3 Metodologia de Execução

3.1 Ambiente Experimental

Os testes foram executados em ambiente controlado local com as seguintes especificações:

- **Hardware:** Processador Ryzen 95900xt 16 núcleos e 64 MB de thread , 48GB de memória RAM, Sistema operacional Linux.
- **Banco de Dados:** SQLite (arquivo local para minimizar latência de conexão TCP).
- **Massa de Dados:** Gerada via biblioteca *Faker*, composta por **100.000 usuários** e seus respectivos posts, criando um cenário de alto volume de dados para estressar as consultas de listagem.

3.2 Cenários de Teste

Foram definidos três cenários principais para estressar diferentes aspectos das APIs:

1. **Simple Fetch (Overfetch):** Busca de um único usuário. O REST retorna todos os campos do registro; o GraphQL solicita apenas campos específicos (Ex: id e nome).
2. **Real World Complex:** Busca de usuário com dados aninhados (Posts e Comentários), exigindo *joins* ou múltiplas resoluções no banco.
3. **N+1 List (Listagem):** Listagem de 10 usuários. Cenário crítico onde o endpoint REST padrão tende a trafegar a coleção inteira ou dados excessivos, enquanto o GraphQL aplica paginação e filtros nativos.

4 Resultados e Análise

Os dados foram coletados e processados gerando as estatísticas abaixo. A análise foca em responder às RQs propostas.

4.1 Análise da RQ 02: Eficiência de Payload

Pergunta: Respostas às consultas GraphQL têm tamanho menor?

Resposta: Sim, consistentemente. Os dados obtidos (Visualizáveis no Dashboard gerado) mostram uma redução drástica no tamanho das respostas.

Cenário	Média GraphQL (Bytes)	Média REST (Bytes)
Cenário 1: Leitura Simples	≈ 76	≈ 1.433
Cenário 2: Leitura Complexa	≈ 716	≈ 2.091
Cenário 3: Listagem (N+1)	**2.247**	**15.633.228**

Tabela 2: Comparativo de Tamanho de Payload (Dados aproximados das medições)

No **Cenário 3**, a diferença é expressiva. O REST trafegou aproximadamente **15 MB**, enquanto o GraphQL trafegou apenas **2 KB**. Isso ocorre porque o REST, sem endpoints específicos de filtragem, trafegou todo o banco de dados para que o cliente realizasse a paginação, enquanto o GraphQL aplicou o filtro `limit` e seleção de campos diretamente no servidor.

4.2 Análise da RQ 01: Performance de Tempo

Pergunta: Respostas às consultas GraphQL são mais rápidas?

Resposta: Depende da linguagem e do cenário.

- **JavaScript (Node.js):** O GraphQL mostrou-se extremamente performático, sendo mais rápido que o REST na maioria dos cenários. O *overhead* de processamento da query foi compensado pela menor quantidade de dados trafegados e serializados.
- **Python (Flask):** Observou-se um fenômeno de "Custo de CPU". Em cenários complexos, o REST foi mais rápido (em tempo de processamento) do que o GraphQL. Isso sugere que a biblioteca de GraphQL utilizada em Python possui um custo computacional alto para resolver os *resolvers* aninhados, penalizando o tempo total apesar da economia de banda.

5 Discussão

5.1 O Problema do Over-fetching

O experimento confirmou que o REST sofre severamente de *over-fetching*. No cenário simples, o REST trafegou cerca de 20 vezes mais dados que o necessário. Em dispositivos móveis ou redes limitadas, essa diferença impacta diretamente a experiência do usuário e o consumo de dados.

5.2 Viés de Implementação (Language Bias)

Foi notada uma discrepância entre as linguagens. O ambiente Node.js lidou melhor com a natureza assíncrona da resolução de grafos do GraphQL. Já em Python, a serialização e resolução do grafo adicionaram uma latência significativa, indicando que a escolha da tecnologia de servidor é crucial ao adotar GraphQL.

6 Conclusão

Este experimento controlado permitiu concluir que:

1. **Rejeita-se a Hipótese Nula para RQ 02:** O GraphQL é estatisticamente superior em eficiência de tamanho de mensagem, eliminando *over-fetching* e reduzindo o uso de banda em mais de 99% em cenários de listagem crítica.

2. **Resultado Misto para RQ 01:** O GraphQL não é inerentemente "mais rápido"em tempo de processamento bruto. Ele ganha velocidade ao reduzir o tráfego de rede, mas impõe um custo de CPU ao servidor. Em implementações otimizadas (como Node.js), ele supera o REST; em implementações com maior overhead (como Python/Graphene), o REST pode ser mais veloz em latência local.