REST or GraphQL? A Performance Comparative Study

Matheus Seabra
Universidade Federal do Pará
Belém, PA, Brasil
matheusvieiracoelho@gmail.com

Marcos Felipe Nazário
Universidade Federal do Pará
Instituto Evandro Chagas
Belém, PA, Brasil
marcosnazario@iec.gov.br

Gustavo Pinto
Universidade Federal do Pará
Belém, PA, Brasil
gpinto@ufpa.br

RESUMO

Given the variety of architectural models that can be used, a frequent questioning among software development practitioners is: which architectural model to use? To respond this question regarding performance issues, three target applications have been studied, each written using two models web services architectures: REST and GraphQL. Through research of performance metrics of response time and the average transfer rate between the requests, it was possible to deduce the particularities of each architectural model in terms of performance metrics. It was observed that migrating to GraphQL. resulted in an increase in performance in two-thirds of the tested applications, with respect to average number of requests per second and transfer rate of data. However, it was noticed that services after migration for GraphQL performed below its REST counterpart for workloads above 3000 requests, ranging from 98 to 2159 Kbytes per second after the migration study. On the other hand, for more trivial workloads, services on both REST and GraphQL architectures presented similar performances, where values between REST and GraphQL services ranged from 6.34 to 7.68 requests per second for workloads of 100 requests.

KEYWORDS

Modelo arquitetural, REST, GraphQL, Teste de desempenho

ACM Reference Format:
Matheus Seabra, Marcos Felipe Nazário, and Gustavo Pinto. 2019. REST or GraphQL? A Performance Comparative Study. In *XIII Brazilian Symposium on Software Components, Architectures, and Reuse (SBCARS '19), September 23–27, 2019, Salvador, Brazil.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUÇÃO

Com a ascensão de dispositivos móveis reivindicando uma parcela cada vez maior do tráfego da Internet, um padrão eficiente para a recuperação de dados é fundamental. Para atender tal demanda, um dos modelos de arquitetura mais utilizados recentemente na implementação comunicação entre serviços web é o REST (Transferência de Estado Representativa, do Inglês, *Representational State Transfer*) [5, 22]. A especificação REST define um modelo arquitetural de software que determina um conjunto de restrições a serem usadas para criar serviços web. Tais serviços precisam estar em conformidade com a especificação arquitetural REST criada por *Fielding* [5, 6], e são comumente chamados pela comunidade de desenvolvimento de software de *RESTful web services* [16]. Tais serviços permitem que sistemas solicitantes acessem e manipulem recursos usando um conjunto uniforme e predefinido de operações sem estado, de forma a fornecer interoperabilidade entre sistemas na Internet.

No entanto, este modelo de arquitetura também impõe restrições na flexibilidade de criar APIs (do inglês, *Application Programming Interface*) que potencialmente apresentam um desempenho abaixo do ideal e dificuldades de implementação como alto acoplamento entre recursos. Uma solução proposta para aumentar a eficiência na busca de dados é através do uso de linguagens de consulta de *API* como GraphQL. Tome, por exemplo, uma empresa na indústria de *e-commerce* que queira implementar serviços web REST para seus clientes, a fim de obter informações detalhadas para seus produtos. Para isso, é necessário que seja implementado um serviço web que forneça dados para uma visualização de produto que deve conter apenas uma imagem, o nome do produto e um preço.

Uma solução recorrente implementada seria implementar um serviço web REST que acesse o modelo de produto por meio da camada de ORM (do Inglês, *Object Relational Mapping*), encarregada de recuperar os dados da base de dados, e retorná-los para o cliente com o intuito de mostrar as informações do produto. Entretanto, um possível problema é que dependendo dos campos adicionados à resposta de uma requisição REST ao cliente, iriam fornecer informações extras que podem ser desnecessárias no momento, como por exemplo o número de unidades em estoque, comentários de clientes que compraram o produto e uma descrição. Essa adição de dados cria uma sobrecarga extra ao transferir dados desnecessários junto com os dados de que a aplicação-cliente requisitando os dados precisa [2].

A solução mais viável em termos de valor agregado para a empresa, seria expor um serviço com múltiplos pontos de acesso (*endpoints*) que forneçam recursos que atendam à todos os seus clientes. No entanto, isso pode rapidamente se tornar computacionalmente custoso, pois clientes vêm em todas as formas e tamanhos, solicitando diferentes respostas e diferentes combinações de dados, que por sua vez têm que servir seus próprios clientes que provavelmente estão fazendo requisições de redes e dispositivos distintos.

Como potencial solução para esse tipo de problema, em 2015 o *Facebook* publicou a especificação do GraphQL, que define um modelo para consulta de *APIs* baseado em uma linguagem declarativa e um sistema de tipagem, que descreve as capacidades de requisitos dos modelos de dados, para operações entre cliente e servidor. Além disso, estudos apontam que GraphQL fornece ganhos de desempenho para os serviços nos quais implementam esse

SBCARS '19, September 23–27, 2019, Salvador, Brazil

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

modelo quando em comparação com o modelo de arquitetura REST [2]. No entanto, até o presente momento, poucos foram os estudos conduzidos para melhor entender se, de fato, os eventuais ganhos de desempenho acontecem. Portanto, essa análise tem como objetivo explorar as diferenças de desempenho entre os dois modelos arquiteturais. Para guiar esse trabalho, foram colocadas as seguintes questões de pesquisa:

- **QPe1**: Qual o número médio de requisições, por segundo, por modelo arquitetural de serviço web?
- **QPe2**: Qual o tempo médio por requisição, em milissegundos, por modelo arquitetural de serviço web?
- **QPe3**: Qual o tempo médio por requisição concorrente, em milissegundos, por modelo arquitetural de serviço web?
- **QPe4**: Qual a taxa média de transferência dos documentos, em *KBytes* por segundo, por modelo arquitetural de serviço web?

Como principal descoberta, foi observado que após a migração, GraphQL introduziu um aumento no número médio de requisições por segundo, entre mais de dois terços das cargas de trabalho postas em teste, e uma redução na taxa de transferência de documento, o que representa um aumento no desempenho para as questões de pesquisa QPe1 e QPe4, pois resultados levam a crer que o corpo da resposta enviada pelos serviços implementados em GraphQL sejam menores, de acordo com as características da linguagem a serem vistas na Seção 2. No entanto houveram algumas exceções, serviços em GraphQL tiveram um desempenho inferior aos serviços REST quando comparados para cargas de trabalho de 3000 requisições, variando de 298 para 2159 *Kbytes*, métrica referente a questão de pesquisa QPe4. Já para questões de pesquisa QPe2 e QPe3, resultados mostraram que o modelo arquitetural REST apresentou desempenho abaixo do esperado entre todas as aplicações testadas, com exceção de cargas de trabalho mais intensas. Já para cargas de trabalho mais triviais, serviços em ambas arquiteturas apresentaram desempenhos similares, onde valores entre serviços REST e GraphQL variaram de 6.34 para 7.68 requisições após o estudo de migração, para cargas de trabalho menores de até 100 requisições. É importante ressaltar também que para níveis de concorrência maiores ou iguais a 1000 usuários, nenhum dos serviços testados foram capazes de servir requisições devido aos serviços estarem indisponíveis. Em resumo, GraphQL não é uma solução para eliminar todos os problemas associados à implementação de *API*. Os gargalos de desempenho podem acontecer e as causas principais podem ser difíceis de identificar. Entretanto, pode ser menos custoso dar manutenção e evoluir aplicações e *APIs* em GraphQL.

2 FUNDAMENTAÇÃO TEÓRICA

Nesta seção, uma fundamentação teórica é brevemente apresentada, destacando os principais aspectos de cada modelo arquitetural neste estudo. As características mais importantes de cada modelo arquitetural são apresentadas para um melhor entendimento das motivações por de trás da análise, bem como as decisões de *design* adotadas pelos modelos arquiteturais utilizados como base do experimento final.

2.1 Arquitetura Orientada à Serviços

Fornecer a interoperabilidade contínua entre pilhas de tecnologia heterogêneas e promover baixo acoplamento entre o consumidor de serviços (solicitante, cliente) e o provedor de serviços são os principais objetivos de *design* da Arquitetura Orientada a Serviços (SOA) e modelos arquiteturais tais como serviços web. [10]

O conceito de agente de software ou serviço consiste em uma unidade de software que atua como provedor de serviço que é desenvolvido para servir os resultados requisitados por um consumidor de serviços. Tanto o provedor quanto consumidor são papéis desempenhados pelos agentes de software. Além disso, SOA é uma abordagem que ajuda os sistemas a permanecerem escaláveis e flexíveis enquanto crescem, ajudando também a reduzir a lacuna atualmente existente entre Negócios e TI.

De forma geral, a estrutura desta arquitetura consiste em três elementos principais:

1. **Serviços**: representam funcionalidades de negócio independentes, que podem fazer parte de um ou mais processos e podem ser implementados em qualquer tecnologia e em qualquer plataforma.
2. **Infraestrutura que suporte interoperabilidade**: conhecida como *enterprise service bus*, (ESB) é a infraestrutura que permite alta interoperabilidade entre sistemas distribuídos para serviços, facilitando a distribuição de processos de negócios em vários sistemas usando diferentes plataformas e tecnologias.
3. **Baixo acoplamento**: é o conceito de reduzir as dependências de um sistema. Considerando que processos de negócio sejam distribuídos em vários *backends*, é importante minimizar os efeitos de modificações e falhas inesperadas. Caso contrário, as modificações se tornam muito arriscadas e as falhas do sistema podem quebrar o status geral do sistema. Há de se notar, no entanto, que há um preço para alcançar baixo acoplamento: complexidade. Sistemas distribuídos baixamente acoplados requerem maior esforço para se desenvolver, manter e depurar.

Um manifesto foi publicado para a arquitetura orientada a serviços em outubro de 2009[10]. Vários dos grandes autores sobre *SOA* participaram, incluindo aqui *Thomas Erl*, *Joe McKendrick*, entre tantos outros. Esse manifesto é um guia de princípios a serem seguidos quando se trata de Arquitetura Orientada a Serviços. Tal manifesto resultou em seis valores principais, listados a seguir:

1. Valor de negócio (*business value*) é mais importante do que a estratégia técnica.
2. Metas estratégicas recebem mais importância do que os benefícios específicos do projeto.
3. Interoperabilidade recebe mais importância do que integrações customizadas.
4. Serviços compartilhados ao invés de implementações com propósitos específicos.
5. Flexibilidade é dada mais importância do que otimização.
6. Refinamento ao longo do tempo ao invés da busca pela perfeição

SOA pode ser visto como uma continuidade que vai desde o conceito mais antigo de computação distribuída e programação

modular, até as práticas atuais de *mashups*, *SaaS* e computação em nuvem (que alguns veem como descendentes de *SOA*).

Em uma aplicação que adota tal arquitetura, protocolos determinam modelo de verificação e encaminhamento de mensagens por meio de metadados. Tais dados descrevem as características funcionais e de qualidade do serviço. A arquitetura também permite que blocos de funcionalidades possam ser criadas exclusivamente a partir de serviços já existentes, que são combinados de maneira plugável para formar uma ou mais aplicações. Um serviço representa uma interface para o cliente, abstraindo a complexidade e agindo como uma "caixa preta". Além disso, outros usuários também podem acessar esses serviços de forma independente, sem qualquer conhecimento de sua implementação interna.[10]

2.2 Serviços Web

Um serviço web é considerado uma coleção de protocolos e padrões abertos usados para trocar dados entre sistemas web. [11] Aplicações de software escritas em várias linguagens de programação e em execução em várias plataformas podem usar serviços web para trocar dados em redes de computadores, como a Internet, de maneira semelhante à comunicação entre processos em um único computador. Tal interoperabilidade foi facilitada devido ao uso de padrões abertos.

Geralmente, o termo serviço web pode englobar um significado mais geral, como por exemplo um serviço oferecido por um dispositivo eletrônico para outro dispositivo eletrônico, que se comunicam através da *World Wide Web*, ou ter um significado mais específico, como um serviço web implementado em uma tecnologia específica (e.g. SOAP web service).

Tais serviços fornecem interoperabilidade entre a comunicação de sistemas,[13] permitindo assim novas oportunidades de negócio. Porém, o conceito de interoperabilidade de software não é novo. Já houve várias implementações que fornecem soluções para esse conceito. A RPC (Chamada de Procedimento Remoto), OSI (Open System Interconnection), CORBA (Common Object Request Broker Architecture) e RMI (Remote Method Invocation) estão entre algumas dessas soluções.[1]

Cada serviço web pode desempenhar o papel de um provedor de serviços que pode ser chamado por um cliente. Além disso, um serviço web pode fornecer flexibilidade para estabelecer comunicação entre sistemas ou dispositivos separados geográficamente.

Esses serviços podem estar disponíveis como arquivos WSDL (Web Service Description Language) ou, às vezes, os serviços podem estar disponíveis diretamente. A crescente popularidade dos serviços web pode ser atribuída a um movimento em direção à Arquitetura Orientada à Serviços (SOA)[10]. Na prática, um serviço web geralmente fornece uma interface orientada a objetos para um servidor de banco de dados, utilizado, por exemplo, por outro serviço ou por um aplicativo móvel, que fornece a interface para o usuário final.

Muitas organizações fornecem seus dados em páginas *HTML* também irão fornecem os mesmos dados em seus servidores como *XML* ou *JSON*, geralmente por meio de um serviço web. Outra caso de uso oferecida ao usuário final pode ser um *mashup*, onde um servidor web consome vários serviços web em máquinas diferentes, compilando os dados consumidos de vários serviços em uma interface do usuário.

"Recursos da Web"foram definidos pela primeira vez na *World Wide Web* como documentos ou arquivos identificados por suas *URLs*. No entanto, hoje eles têm uma definição muito mais genérica e abstrata que engloba qualquer entidade que pode ser identificada, nomeada, endereçada ou manipulada, de qualquer forma, na Web. Portanto, a principal diferença entre um serviço web e um website está em quem o acessa. Websites geralmente são acessados por pessoas, e serviços web são acessados por outros sistemas de software.

2.3 REST

REST (Transferência de estado representativa, do Inglês, *Representational State Transfer*) é uma especificação de modelo de arquitetura de software que define um conjunto de restrições a serem usadas para criação de serviços web. O termo transferência de estado representativa foi introduzido e definido no ano de 2000 por Roy Fielding em sua dissertação de doutorado [5] e posteriormente em um artigo científico [6]. O trabalho de Fielding explicou os princípios REST que eram conhecidos como "modelo de objeto HTTP" a partir de 1994 e foram usados no design dos padrões HTTP 1.1 e URI [6]

O termo REST destina-se a evocar a imagem de como um aplicativo da Web bem projetado se comporta: é uma rede de recursos (máquina de estados virtuais) na qual o usuário avança pela aplicação selecionando identificadores de recursos.

Serviços web implementados no modelo arquitetural REST permitem que clientes acessem e manipulem representações textuais de recursos da Web usando um conjunto uniforme e predefinido de operações sem estado. Já outros tipos de serviços web, como serviços SOAP (do Inglês, *Simple Object Access Protocol*), expõem seus próprios conjuntos de operações arbitrários.

Objetos em REST são definidos como URIs endereçáveis. Quando o protocolo HTTP é usado, como é mais comum, a interação com os recursos é feita através dos verbos internos do HTTP, especificamente: GET, PUT, DELETE, POST. Em particular:

- **GET**: recupera um recurso.
- **PUT**: atualiza um recurso existente ou cria um recurso.
- **POST**: cria um novo recurso.
- **DELETE**: remove um recurso.

Mais concretamente, um acesso via método **GET** a um acesso ao um recurso disponível no site do Twitter, poderia ser feito através da URI `https://www.twitter.com/user/bruno/tweet/16`, assumindo que 16 é identificador o recurso (tweet) desejado.

Operações para manipular recursos, como *GET* (para consulta) ou *POST* (para inserção), resultam na próxima representação do recurso (o próximo estado da aplicação) sendo transferida ao usuário final para seu uso.

Em um serviço web REST, as solicitações feitas a uma URI de um recurso geram uma resposta com formatos HTML, XML, JSON ou outro formato de serialização de dados.[5] A resposta a uma URI (recebida como documento JSON, por exemplo) pode confirmar que alguma alteração foi feita no recurso armazenado, além de eventualmente fornecer links de hipertexto para outros recursos ou conjuntos de recursos relacionados.

Utilizando-se do protocolo HTTP e operações padronizadas, as aplicações que utilizam serviços web REST buscam desempenho, confiabilidade e capacidade de crescimento, reutilizando componentes que podem ser gerenciados e atualizados sem afetar o sistema como um todo, mesmo enquanto estiver em execução.[5]

2.4 GraphQL

GraphQL [4] é uma linguagem de consulta para *APIs*. A linguagem fornece uma descrição do modelo de dados, oferecendo aos clientes a capacidade de solicitar exatamente os dados que precisam. Utilizada para consultar servidores que possuem recursos e dados definidos de acordo com o sistema de tipos que a acompanha a linguagem, onde para cada recurso existe um tipo. Um conjunto de tipos formam um *schema*.

Uma *API* GraphQL é criada definindo tipos e campos nesses tipos e, em seguida, fornecer funções para cada campo de cada tipo. Tais funções são chamadas de *resolvers*. Por exemplo, na Figura 1,um serviço que fornece dados sobre artigos de blog poderia ser implementado com o seguinte *schema*:

```
type Query {
  topStories(limit: Int): [Story]
}

type Story {
    title: String
    text: String
    upvotes: [String]
    url: String
    createdAt: Date
}
```

Figura 1: Formato de uma schema de dados em GraphQL

Feita a modelagem e relacionamento da aplicação, pode ser feita a implementação de suas respectiva função *resolver*:

```
function topStories() {
  return Story.find({}).sort('-createdAt');
}
```

Figura 2: Exemplo de função *resolver* em *JavaScript*.

Uma vez que um serviço GraphQL está sendo executado (geralmente em uma URL em um serviço web), clientes podem enviar consultas GraphQL. Uma consulta recebida é primeiramente verificada para garantir que ela se refere apenas aos tipos e campos definidos e, em seguida, executa as funções *resolver* para responder aos clientes. Para ilustrar um exemplo, a consulta GraphQL mostrada na Figura 3 solicita os títulos dos artigos mais populares do blog *Hacker News*:

A consulta na Figura acima irá produzir a seguinte resposta mostrada na Figura 4:

Atualmente, na maioria das aplicações há a necessidade de buscar dados de um servidor, e esses dados por sua vez estão armazenados em um banco de dados. Portanto, é responsabilidade da *API*

```
{
    hn {
        topStories(limit: 3) {
            title
        }
    }
}
```

Figura 3: Formato de uma consulta em GraphQL

```
{
  "data": {
    "hn": {
      "topStories": [
        {
          "title": "Monte Carlo methods  Why it's a bad
          idea to go to the casino"
        },
        {
          "title": "Public Domain Movies"
        },
        {
          "title": "Agent 355"
        }
      ]
    }
  }
}
```

Figura 4: Exemplo de uma resposta HTTP no formato *JSON* respondida por um serviço web GraphQL

fornecer uma interface para os dados armazenados que atenda às necessidades da aplicação.

GraphQL é frequentemente confundido com um modelo de banco de dados. Isso é um equívoco, GraphQL é uma linguagem de consulta para *APIs*. Nesse sentido, a especificação é agnóstica de banco de dados e, efetivamente, pode ser implementada em qualquer contexto em que uma *API* é usada.

GraphQL é frequentemente explicado como uma "interface unificada para acessar dados de diferentes fontes". Embora essa explicação seja precisa, ela não revela as ideias subjacentes ou a motivação por trás da linguagem. GraphQL permite que clientes possam consultar um banco de dados representado por um *schema*. Tal escolha de *design* representa uma grande mudança de paradigma para serviços REST: em REST, servidores implementam uma lista de pontos de acesso (recursos) que podem ser requisitados pelos clientes; em contraste, serviços GraphQL mapeiam um banco de dados como um grafo de dados, que pode ser consultado pelos clientes. Dessa forma, a complexidade de requisitar os dados é passada do servidor para o cliente.

A estrutura de um banco de dados é definida por um *schema*, que basicamente representa um grafo. Nós contidos no grafo são objetos, que definem um tipo e incluem uma lista de campos; cada campo tem um nome e também um tipo. As arestas aparecem quando os objetos definem campos cujos tipos são outros tipos de objetos.

Uma vez que um serviço GraphQL está sendo executado (geralmente em uma URL em um serviço da web), ele pode enviar consultas GraphQL para validar e executar. Uma consulta recebida é primeiro verificada para garantir que ela se refira apenas aos tipos e campos definidos e, em seguida, executa as funções fornecidas para produzir um resultado.

3 MÉTODO DE PESQUISA

O objetivo desse trabalho é explorar o comportamento, em termos de desempenho, entre os modelos arquiteturais de serviços web REST e GraphQL. Nesta seção são determinadas as questões de pesquisa, as aplicações estudadas, e o projeto do experimento final.

3.1 Questões de Pesquisa

De acordo com os conceitos e fundamentação na Seção 2, foram selecionadas quatro questões de pesquisa que buscam apresentar um retrato inicial do comportamento de desempenho de arquiteturas de serviços web. Portanto, a questão de pesquisa (QP) geral que esta análise e seus experimentos propõem a investigar é:

QP: Dentre os modelos arquiteturais de serviços web REST e GraphQL, qual apresenta melhor desempenho?

Para responder essa questão de pesquisa geral, foram criadas quatro outras questões de pesquisa específicas (QP). São elas:

- **QP1**: Qual o número médio de requisições, em segundos, por modelo arquitetural de serviço web?
- **QP2**: Qual o tempo médio por requisição, em milissegundos, por modelo arquitetural de serviço web?
- **QP3**: Qual o tempo médio por requisição concorrente, em milissegundos, por modelo arquitetural de serviço web?
- **QP4**: Qual é a taxa média de transferência dos documentos, em *KBytes* por segundo, por modelo arquitetural de serviço web?

A QP1 tem como objetivo analisar o número médio de requisições por segundo que o sistema é capaz de atender. A QP2 analisa a média de tempo, em milissegundos, para uma requisição apenas. A QP3, por outro lado, visa mensurar qual a média de tempo, em milissegundos, para requisições onde há concorrência. E por fim, a QP4, procura responder qual é a taxa de transferência das respostas enviadas pelo serviços, em *Kbytes* por segundo.

As questões de pesquisa são específicas, e têm como objetivo avaliar dois dos principais aspectos que distinguem esses modelos arquiteturais: latência e volume. Ou seja, o tempo de entrega (latência) e a taxa de transferência dos documentos (volume) entregues ao cliente. Para fornecer respostas para estas questões de pesquisa, foi seguido um protocolo experimental rigoroso, como apresentado na Seção 3, aplicado em um conjunto de aplicações-alvo, como apresentado na Seção 3.2.

3.2 Aplicações Alvo

Para realizar este estudo, foram utilizadas três aplicações que contam com tanto versões escritas em REST, quanto em GraphQL. Dado uma consulta inicial em repositório de projetos de software como *GitHub* e *Bitbucket*, não foi identificado nenhum sistema de *software* que tivesse duas versões da mesma aplicação desenvolvida utilizando os dois modelos arquiteturais. Dessa forma, foi decidido implementar três aplicações de software usando ambos modelos arquiteturais. As aplicações implementadas são descritas a seguir:

- **App1**: Aplicação chamada *RocketBox*, que permite usuários gerenciar pastas e fazer *upload* de seus arquivos como imagens, audio video e documentos. O intuito de implementar esse tipo de aplicação foi de validar como os modelos de arquiteturas em questão se comportam quando há muitas operações com o disco rígido da máquina, como é o caso da funcionalidade de *upload* de arquivos. Essa aplicação foi escrita em *Node.js* utilizando o *microframework* web *Express*, e possui 356 linhas de código no modelo REST e 271 ao ser reescrita em GraphQL.
- **App2**: Aplicação chamada *Tech Teams*, desenvolvida com o objetivo de gerência de projetos para pequenos times de tecnologia. Talvez essa seja a aplicação mais genérica em termos de funcionalidades dentre as aplicações selecionadas. A aplicação faz operações básicas de gerência para projetos e tarefas. Tarefas por sua vez podem ser compostas de diversas sub-tarefas. Essa aplicação foi escrita em *Node.js* utilizando o *microframework* web *Express*, e possui 471 linhas de código no modelo REST e 388 ao ser reescrita em GraphQL.
- **App3**: Aplicação chamada *TwitterC*, que utiliza o protocolo de *WebSocket* através da biblioteca socket.io[18] para realizar conexões persistentes de tempo real com o servidor, a fim de simular as funcionalidades primárias do *website Twitter*. Esta aplicação foi escrita em *JavaScript*, e tem 171 linhas de código no modelo REST e 265 no modelo GraphQL.

3.2.1 Processo de implementação e migração. O processo de desenvolvimento ocorreu da seguinte forma. Inicialmente, todas as aplicações foram desenvolvidas no modelo REST. Durante o estudo de migração, as aplicações-alvo foram refatoradas do modelo arquitetural REST para GraphQL. Essa refatoração envolveu mapear (*endpoints*) típicos da arquitetura REST, para tipos em GraphQL. Essa foi uma etapa importante para o primeiro autor da análise ao projetar o experimento, como também a implementação e execução dos mesmos, tido que o primeiro autor já havia o conhecimento básico dos modelos de arquitetura.

Durante a migração das aplicações também foi possível observar que não é uma tarefa trivial refatorar os serviços REST para utilizar consultas aninhadas usando GraphQL. Para cada recurso existente e exposto como ponto de acesso um serviço REST, é necessário um tipo em GraphQL. Um mapeamento precisa ser feito, o que nem sempre é uma operação simples a ser feita pois muitas vezes, há relacionamentos entre tais recursos, resultando em um *schema* GraphQL que irá gerar consultas altamente aninhadas. Portanto, refatorar tais sistemas para servir uma larga estrutura de dados em grafo pode se tornar rapidamente uma tarefa complexa de reengenharia dependendo do tamanho do sistema sendo migrado.

Tanto a implementação quanto o processo de migração foi conduzido pelo autor do artigo e é, portanto, propensa a erros. Para minimizar essa ameaça, testes manuais foram realizados, após o desenvolvimento das aplicações, para garantir que todas as aplicações apresentavam o comportamento esperado. Também foi disponibilizado o código fonte das aplicações publicamente, de forma a

permitir inspeção, replicação e testes por outros pesquisadores e praticantes[17].

3.3 Experimento

A fase de desenho do experimento começou com a pesquisa de métricas de desempenho adequadas e como tais métricas podiam ser medidas. As aplicações em teste foram implementadas, experimentos de latência e tempo foram projetados, teste de carga foram propostos e conjuntos de dados para uso nos experimentos foram determinados. Após a implementação e migração das aplicações-alvo, foram executadas baterias de testes de desempenho em cada uma das aplicações.

3.3.1 Métricas de Desempenho. Com base nesses indicadores, foram empregadas as seguintes métricas para a análise de desempenho das aplicações-alvo:

- **(RPs) Request per second:** Número médio de requisições por segundo.
- **(TPR) Time per request:** Tempo médio da duração de requisições em milisegundos.
- **(TPCR) Time per concurrent request:** Tempo da duração de requisições concorrentes em milisegundos.
- **(TR) Transfer Rate:** Taxa de transferência de dados em Kbytes por segundo.

Para medir estas métricas, a ferramenta *Apache Benchmark*(AB)[1] foi utilizada. Esta ferramenta realiza testes de carga ao enviar um número arbitrário de requisições simultâneas para servidores. De acordo com o seu website, o AB é uma ferramenta para pontuar servidores HTTP, projetada para dar ao usuário um relatório do desempenho de um servidor HTTP. A ferramenta também exibe quantas requisições por segundo um servidor HTTP é capaz de servir. Ademais, a ferramenta AB tem sido foco de recentes estudos de desempenho, em outros contextos [**?**].

As métricas descritas foram exercitadas utilizando diferentes cargas de trabalho. A Tabela 1 apresenta os detalhes das cargas de trabalho.

Tabela 1: Descrição da carga de trabalho empregada nas aplicações-alvo.

Requisições	Concorrência
100	1, 10, 100 usuários
1000	1, 500, 1000 usuários
3000	1, 1000, 3000 usuários

Como é possível perceber na tabela 1, há três grupos de requisições (100, 1.000 e 3.000 requisições) e, para cada grupo de requisição, há três níveis de concorrência. Os níveis de concorrência foram implementados através da ferramenta AB, que permite definir um nível de concorrência para as requisições.

Por exemplo, ao experimentar uma aplicação alvo com 100 requisições, os níveis de concorrência sugerem o grau de concorrência dessas requisições. O nível 1 indica que as 100 requisições serão feitas de forma sequencial (uma após a outra) enquanto que o nível 100 indica que as requisições serão feitas simulando um alto número de usuários simultâneos (respeitando o número de processadores disponíveis). Ao final desse processo, foram realizados um total de 162 execuções (6 aplicações × 3 níveis de requisições × 9 níveis de concorrência).

No entanto, como pontuado na Seção 4, ao executar serviços em ambas arquiteturas para as cargas de trabalho 1000/1 e 3000/1 (requisições/concorrência), não foi possível concluir o experimento, devido a erros HTTP com código de status 408, ou seja, o tempo de requisição foi esgotado, mostrando que foi uma característica do servidor que hospeda a aplicação, ao invés de ser um erro do serviço que realizou as requisições.

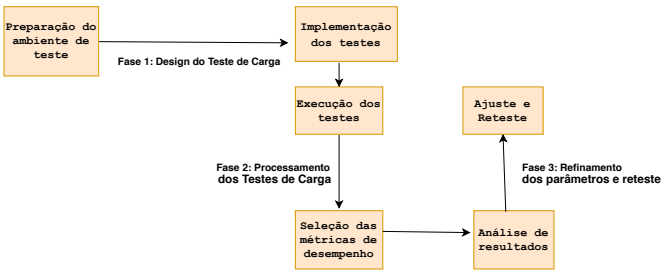

Figura 5: Uma visão geral do fluxo para de Teste de Carga utilizada na análise

3.3.2 Execução dos experimentos. A Figura 5 apresenta o fluxo geral de execução dos experimentos. Para cada uma das aplicações apresentadas, as seguintes etapas foram executadas:

(1) **Preparação do ambiente de teste:** Etapa de configuração da aplicação em teste. Consiste em fazer o *deploy* do serviço em execução no *host* local da máquina e listar os pontos de acesso (*endpoints*) para o serviço sendo testado.
(2) **Implementação do teste:** Etapa de execução de um *script* customizado com as chamadas da ferramenta AB. Esta etapa principal do teste, consiste em montar as chamadas com os pontos de acesso e cargas de trabalho determinadas na Seção 3.3.1.
(3) **Execução do teste:** Feito o mapeamento das chamadas, a próxima etapa do teste consiste em monitorar a execução do *script* com as respectivas chamadas de teste de carga para as aplicações, assim como monitorar *logs*.
(4) **Seleção das métricas de desempenho:** Após a execução dos testes ser finalizada, o *script* seleciona as métricas relevantes para análise e serializa tais métricas em arquivos no formato CSV para cada uma das métricas selecionadas.
(5) **Análise de resultados:** Esta etapa consiste na geração dos gráficos a partir dos dados extraídos no passo 4, por meio da biblioteca *Gnuplot* para visualização de dados.
(6) **Ajuste e reteste:** Por fim, a etapa de ajuste e reteste consiste na análise dos gráficos gerados pelo Gnuplot[21] a partir das métricas geradas pela ferramenta Apache Ab[7].

3.4 Ambiente de Execução

Para a execução dos experimentos, foi utilizado um computador Intel NUC modelo D54250WYKH, com processador Intel® Core

[1] https://httpd.apache.org/docs/2.4/programs/ab.html

i7-2670QM, uma CPU de 2.20GHz com 8 núcleos físicos e memória de modelo DDR2-667 e tamanho 16GB, e sistema operacional Ubuntu 18.04.02 LTS (*kernel* 4.4.0-112-generic) e a linguagem de programação *JavaScript*, usando *Node.js* como *runtime environment*, na sua versão 10.15.3. Por fim, experimentos foram realizados sem nenhuma outra carga de trabalho em execução simultânea no computador que realizou os experimentos (exceto processos do próprio sistema operacional). Para cada *benchmark*, os resultados dos testes de carga são referente a média de quinze iterações ao longo do mesmo processo em execução.

4 RESULTADOS

Nesta seção os resultados apresentados são organizados em questões de pesquisa específicas. Como afirmado na Seção 1, a motivação do trabalho vem da possibilidade da linguagem GraphQL fornecer ganhos de desempenho para os serviços nos quais implementam esse modelo quando em comparação com o modelo de arquitetura REST. Para cada questão de pesquisa, primeiro é apresentado uma tabela com os resultados. Em seguida, são descritas observações gerais sobre os resultados obtidos.

4.1 QP1: Qual o número médio de requisições, por segundo, por modelo arquitetural de serviço web?

A tabela 2 apresenta os resultados para a questão de pesquisa QP1. Após a execução e análise dos testes para a aplicação *RocketBox*, pode-se observar que para o modelo arquitetural REST, serviços apresentaram uma média de 9.92, 33.42, 60.40 requisições para as cargas de trabalho 100/1, 100/10, e 100/100, respectivamente. Após a migração dos serviços em GraphQL, resultados apresentaram em média 34.83, 33.72, e 64.56 requisições, respectivamente para a mesma carga de trabalho. É notável observar resultados para a primeira carga, onde há uma discrepância de 3,92 para 34,83 requisições, ao contrário das demais cargas de trabalho onde há maioria das diferenças entre os resultados foi mais sútil. Já para a aplicação *TwitterC*, nota-se que houve um pequeno aumento de desempenho entre todas as cargas de trabalho testadas, após a migração de serviços web em REST para GraphQL. Com exceção da carga de trabalho 1000/1, onde houve uma redução de aproximadamente 125 requisições para 7 requisições por segundo. Por fim, a média de requisição por segundo para a aplicação *Tech Teams* mostra que que em geral, serviços GraphQL obtiveram maior desempenho quando comparado com a sua primeira implementação no modelo arquitetural REST. A tabela 2 omite os resultados quando executado com cargas de trabalho 1000/1000, 3000/1000 e 3000/3000, uma vez que a execução não foi bem sucedida devido à erros do tipo *SystemError*, onde o processo *node* é finalizado com *exit code* 1 nas aplicações em ambas arquiteturas.

4.2 QP2: Qual o tempo médio por requisição, em milissegundos, por modelo arquitetural de serviço web?

A tabela 3 apresenta os resultados da QP2, onde foi perguntado qual é o tempo médio por requisição, em milissegundos, por modelo arquitetural de serviço web REST e GraphQL. E após a execução e análise dos resultados dos testes de desempenho para a aplicação *RocketBox*, pode-se observar que para o modelo arquitetural REST, o serviço apresentou uma média de 254.45, 292.27 e 1.5552 requisições para as cargas de trabalho 100/1, 100/10, e 100/100, respectivamente, de acordo com a 3. Após a migração do mesmo serviço para GraphQL, resultados apresentaram em média 249.18, 287.97, e 1.321 requisições, respectivamente para as mesmas cargas de trabalho. Para a aplicação *TwitterC*, nota-se que houve um aumento relativamente baixo de desempenho entre todas as cargas de trabalho testadas, após a migração do serviço web em REST para GraphQL. Em contraste, foi observado que para carga de trabalho mais intensas, o serviço REST testado obteve um melhor desempenho no que se refere à tempo médio por requisição em milissegundos. Por fim, para a aplicação *Tech Teams*, resultados mostram que após a migração dos serviços, resultados dos serviço GraphQL teve um desempenho abaixo da sua primeira implementação em REST. Ademais, a tabela 3 omite os resultados quando executado com cargas de trabalho 1000/1000, 3000/1000 e 3000/3000, uma vez que a execução não foi bem sucedida devido à erros *SystemError* onde o processo *node* é finalizado com *exit code* 1 nas aplicações em ambas arquiteturas.

4.3 QP3: Qual o tempo médio por requisição concorrente, em milissegundos, por modelo arquitetural de serviço web?

A tabela 4 apresenta os resultados da questão de pesquisa QP3. Os resultados revelam que para a aplicação *TwitterC*, houve uma redução no tempo médio de requisições entre todas as cargas de trabalho testadas, após a migração de serviços web na arquitetura REST para GraphQL. Já nos resultados aplicação *RocketBox*, serviços implementados no modelo arquitetural REST apresentaram uma média de 253.09, 277.26 e 1530.95 requisições para as cargas de trabalho 100/1, 100/10, e 100/1000, respectivamente.

Após a migração dos serviços em GraphQL, resultados apresentaram em média 249.42, 285.39 e 1323.59 requisições, respectivamente para a mesma carga de trabalho. Por fim, o tempo médio de requisição concorrente para a aplicação *Tech Teams* foi de aproximadamente 383, 277 e 744 para as cargas de trabalho 100/1, 100/10, e 100/100 respectivamente. Para carga de trabalho 1000/500 requisições, houve uma redução de 584 para 517 milissegundos após a migração das aplicações para GraphQL, no entanto, para cargas acima de 3000 requisições, resultados mostram que o serviço na arquitetura REST ainda se sobressai, tendo um tempo médio de requisição de 300.65 milissegundos para 517.20 milissegundos.

Após a execução e análise dos testes para a QP3, pode-se observar que houve uma redução do tempo médio por requisição concorrente em duas das três aplicações testadas após o estudo migração visto na seção 3. A tabela 4 omite os resultados quando executado com cargas de trabalho 1000/1000, 1000 e 3/3000, uma vez que a execução não foi bem sucedida devido à erros *SystemError* onde o processo node é finalizado com *exit code* 1 nas aplicações em ambas arquiteturas.

	RocketBox		TwitterC		Tech Teams	
	REST	**GraphQL**	**REST**	**GraphQL**	**REST**	**GraphQL**
100/1	3.92	34.83	6.34	7.68	2.57	**33.37**
100/10	33.42	33.72	119.04	119.78	33.34	133.74
100/100	60.04	64.56	248.85	244.28	135.06	128.36
1000/1	3.94	8.74	125.68	7.20	3.52	22.72
1000/500	50.84	66.85	248.85	105.80	105.73	157.56
3000/1	50.84	31.27	3.01	119.78	3.01	101.92

Tabela 2: Número médio de requisições, por segundo, por modelo arquitetural de serviço web REST e GraphQL

	RocketBox		TwitterC		Tech Teams	
	REST	**GraphQL**	**REST**	**GraphQL**	**REST**	**GraphQL**
100/1	254.45	249.18	156.10	156.14	383.72	291.19
100/10	292.27	287.97	174.94	176.94.00	733.95	377.77
100/100	1552.51	1321.79	377.07	379.47	177.10	735.44
1000/1	252.12	252.12	175.91	138.62	383.64	475.99
1000/500	1963.56	1963.56	377.07	104.36	266.78	619.83
3000/1	1963.56	1299.40	298.53	177.68	298.53	2159.72

Tabela 3: Tempo médio por requisição, em milissegundos, por modelo arquitetural de serviço web REST e GraphQL

	RocketBox		TwitterC		Tech Teams	
	REST	**GraphQL**	**REST**	**GraphQL**	**REST**	**GraphQL**
100/1	253.09	249.42	155.48	105.80	383.30	289.07
100/10	277.26	285.39	153.62	05.46	277.25	744.84
100/100	1530.95	1323.59	352.92	222.29	744.85	497.91
1000/1	250.79	275.05	153.62	138.41	383.30	497.91
1000/500	620.92	488.32	306.33	193.18	584.80	517.20
3000/1	1946.20	1089.88	300.67	187.83	300.65	517.20

Tabela 4: Tempo médio por requisição concorrente, em milissegundos, por modelo arquitetural de serviço web REST e GraphQL

4.4 QPe4: Qual é a taxa média de transferência dos documentos, em kilobytes por segundo (kbps), por modelo arquitetural de serviço web?

De acordo com os resultados obtidos na tabela 5, o resultado mais notável é para a aplicação RocketBox, onde os valores para as taxas de transferência de documento são praticamente idênticos entre as cargas de trabalho testadas, com exceção da carga de trabalho 3000/1, onde houve uma redução de 369.91 para 393.79 milissegundos, no que se refere ao tempo médio por requisições concorrentes. Tais resultados podem indicar que serviços entre as arquiteturas REST e GraphQL só começam a exibir resultados relevantes a partir de cargas de trabalho mais estressantes.

Para as demais aplicações, houveram reduções mais significativas em termos de Kbytes por segundo. No caso da aplicação TwitterC em particular, para a carga de trabalho 1000/500, serviços implementados na arquitetura REST foram capazes de transferir 486.43 Kbytes por segundo, enquanto que após a migração para GraphQL, serviços obtiveram uma taxa de 268 *Kbytes* por segundo. Essa diferença indica que o corpo da resposta está sendo menor, portanto, menos bytes são transferidos por segundo. Resultados para esta métrica mostram que ao usar REST como modelo arquitetural, clientes precisaram processar largos documentos no formato *JSON* para consumir apenas alguns campos fornecidos pelos serviços, um problema conhecido pela comunidade de software de como *over-fetching*, ou seja, há uma recuperação de dados com uma carga extra de dados não utilizadas pelo cliente. Em contraste, ao adotar GraphQL, as consultas dos clientes especificaram exatamente os campos que precisamente dos servidores, refletidos pela redução no que se refere a média da taxa de transferência de documentos.

Em resumo, a principal qualidade do GraphQL está possuir um design de recuperação de dados de forma mais declarativa[4], em contraste a serviços web REST tradicionais onde recursos são pontos definidos em pontos de acesso fixos. Já o GraphQL, trata em descrever os recursos e requisitos dos modelos de dados para aplicações cliente-servidor como sua principal função [4], enquanto

	RocketBox		TwitterC		Tech Teams	
	REST	**GraphQL**	**REST**	**GraphQL**	**REST**	**GraphQL**
100/1	28.32	28.32	64.56	6.00	268.64	12.59
100/10	247.68	247.68	256.98	128.36	127.47	248.32
100/100	470.94	470.94	449.11	268.99	269.12	117.45
1000/1	28.71	28.71	271.00	127.36	6.72	12.59
1000/500	399.65	399.65	486.43	268.99	99.25	99.02
3000/1	369.91	393.79	300.65	128.82	15.96	95.81

Tabela 5: Taxa de transferência de documento, em Kbytes por segundo, por modelo arquitetural de serviços web REST e GraphQL

REST se concentra em manter a confiabilidade dos serviços como objetivo principal[6]. Mesmo que um serviço REST retorne apenas uma resposta parcial, o serviço ainda estaria transferindo mais dados, enquanto o GraphQL está sempre visando a menor solicitação possível. Com estas observações, podemos estabelecer que o experimento gerou resultados parcialmente acurados para um ambiente controlado com um conjunto de dados bem definidas, o que trouxe uma variedade de métricas na qual possíveis experimentos podem iterar no futuro.

5 LIMITAÇÕES

Existem ameaças que podem afetar a validade do presente trabalho. Primeiramente, as aplicações alvo empregadas não contemplam todos os possíveis tipos de aplicações que se beneficiam dos modelos arquiteturais REST e GraphQL. Além disso, essas aplicações também não exercitam todos os tipos de comportamentos empregados por serviços web, da mesma forma que são limitados em termos da utilização de outros tipos de serviços internamentos (como banco de dados, bibliotecas e outros serviços de terceiros). Ainda, as aplicações são relativamente pequenas, quando observado o total de 1.922 linhas de código utilizadas (média de 320 linhas de código por serviço). No entanto, na era dos microserviços, serviços web com poucas centenas de linhas de códigos não são necessáriamente incomuns [12]. Ademais, o número de requisições proposto nas cargas de trabalho também limitou o alcance dos experimentos para um intervalo restrito de requisições e nível de concorrência, fazendo com que resultados sejam restritos a um determinado intervalo de carga de trabalho.

Como um estudo de migração e teste de desempenho, os resultados estão sujeitos a um número de fatores externos, mesmo embora alguns deles tenham sido mitigados pelo emprego da metodologia escolhida. Além disso, o impacto nas métricas de tempo medidas pelas questões de pesquisa QP1 e QP2 poderia ter sido alvo de investigações mais detalhadas, uma vez que foi observado que a aplicação *RocketBox* obteve resultados similares em ambas implementações em REST e GraphQL. Um atento leitor pode comentar, porém, que ao invés de implementar seis aplicações do zero (três usando modelo REST e três usando modelo GraphQL), talvez fosse mais conveniente buscar três aplicações iniciais que estejam implementadas em um modelo, e depois realizar somente três outras implementações. Apesar dessa abordagem parecer mais representativa (pois poderia se beneficiar de aplicações já conhecidas) e eventualmente mais simples do que a proposta nesta análise, essa abordagem mostrou-se formidavelmente complexa, pois requeria um esforço para (1) ambientação com a aplicação inicial para em seguida (2) transformação da aplicação inicial para um novo modelo. Para mitigar que outros autores se deparem com o mesmo problema exposto, o código fonte das seis aplicações criadas neste estudo estão disponíveis para consulta e eventual replicação do trabalho (a ser divulgado após aceitação).

6 TRABALHOS RELACIONADOS

Embora existam vários trabalhos na literatura que explorem diversos aspectos de arquiteturas REST e/ou GraphQL (por exemplo, [8, 9, 14–16, 20]), há bem menos no que se restringem a mensurar métricas de desempenho desse tipo de modelo arquitetural [2, 3, 19].

Entre eles, destaca-se o estudo realizado por Cederlund e colegas [3]. Este trabalho mediu a latência e volume de dados para distinguir as diferenças de desempenho do serviços utilizando os modelos arquiteturais de serviço REST e GraphQL. Cederlund concluiu que não há uma solução para todos os diferentes casos de uso no qual realizou os testes. No caso de sua pesquisa, os diferentes casos envolveram buscar dados de diferentes pontos de acesso, retornando diferentes quantidades de dados para cada ponto. Com base em dados que não apenas são de tipos diferentes mas que estão sendo solicitados de forma diferentes, essas diferenças nos dados solicitados podem ser migradas para um *API* GraphQL, onde clientes apenas requisitam uma simples consulta para um campo, como um título de um artigo, ou o próprio corpo do artigo inteiro. O estudo mencionado explora diferentes conjuntos de dados, no entanto, o autor fez a pesquisa em cooperação com uma agência de notícias conhecida na Suécia, o que limita os dados de teste em um certo grau para não mencionar que, colaborando com um agência privada, os dados que estão sendo usados não estão diretamente disponíveis para replicação e estudo exato com os mesmos dados.

Há também estudos científicos já feitos sobre testes de desempenho [19], onde o objetivo era medir o tempo de latência e o tempo de resposta de um serviço web REST, o experimento foi conduzido e executado em várias iterações, com cada iteração com duração aproximadamente 1 hora. Em cada iteração, são realizadas requisições arbitrárias repetidamente em intervalos de 500 ms. O tempo médio de resposta foram então registrados em intervalos de 5 minutos.

Outro estudo científico realizado pelo pesquisador Brito [2], onde é comparada a *API* de REST versus a *API* do GraphQL e apresenta como resultado que o GraphQL pode reduzir o tamanho dos documentos retornados por *APIs* REST em 94% (em número de campos) e em 99% (em número de *bytes*), ambos os resultados medianos.

7 CONCLUSÃO

Nesse trabalho foi apresentado um estudo comparativo de serviços web entre modelos de arquitetura REST e GraphQL. O foco principal esteve no desempenho dos serviços em termos requisições por segundo, tempo de reposta (latência) e volume de dados. Implementações baseadas em um aplicações do mundo real foram desenvolvidas para depois serem migradas para utilizar GraphQL. Os resultados permitiram conclusões sobre o desempenho das arquiteturas. Os achados também concluem quando é benéfico adotar um modelo arquitetural ao invés de outro.

Como principal descoberta, foi observado que serviços web migrados para GraphQL apresentaram um aumento no desempenho em dois terços das aplicações testadas, no que se refere ao número médio de requisições por segundo e taxa de transferência de documento em *Kbytes* por segundo, métricas mensuradas nas questões de pesquisa QP1 e QP4. No entanto houveram algumas exceções, foi percebido que serviços em GraphQL obtiveram desempenho abaixo ao serviços em REST para cargas de trabalho de 3000 requisições, variando de 298 para 2159 *Kbytes*, métrica referente a questão de pesquisa QP4. Já para questões de pesquisa QP2 e QP3, resultados mostraram que o modelo arquitetural REST apresentou desempenho abaixo entre todas as aplicações, com exceção de cargas de trabalho com 3000 requisições. Ademais, para cargas de trabalho mais triviais, serviços em ambas arquiteturas apresentaram desempenhos similares, onde valores entre serviços REST e GraphQL variaram de 6.34 para 7.68 milissegundos após o estudo de migração, para cargas de trabalho menores de até 100 requisições. É importante ressaltar também que para níveis de concorrência maiores ou iguais a 1000 usuários, nenhum dos serviços testados foram capazes de servir requisições devido aos serviços estarem indisponíveis.

No entanto, o GraphQL não é uma solução para eliminar todos os problemas associados à implementação de *API*. Os gargalos de desempenho podem acontecer e as causas principais podem ser difíceis de identificar. No entanto, pode ser menos custoso dar manutenção e evoluir aplicações e *APIs* em GraphQL. Assim, é importante que os profissionais de teste trabalhem em estreita colaboração com desenvolvedores de *API* GraphQL para implementar mecanismos de rastreamento adequados e criar testes de desempenho que vão além de medir solicitações simples e intervalos de tempo de resposta entre cliente e servidor. Cada parte de uma implementação GraphQL precisa estar sujeita à análise de desempenho. GraphQL tem muito a oferecer, mas para garantir que seja uma modelo arquitetural eficaz, planejamento e precauções devem ser tomadas por seus implementadores.

ACKNOWLEDGMENTS

We thank the reviewers for their helpful comments. This research was partially funded by CNPq/Brazil (406308/2016-0) and UFPA/-PROPESP.

REFERÊNCIAS

[1] T.Bezboruah A. Bora. 2014. Testing and Evaluation of a Hierarchical SOAP based Medical Web Service. *International Journal of Database Theory Application* 7, 5 (2014).

[2] G. Brito, T. Mombach, and M. T. Valente. 2019. Migrating to GraphQL: A Practical Assessment. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 140–150. https://doi.org/10.1109/SANER.2019.8667986

[3] Matias Cederlund. 2016. Performance of frameworks for declarative data fetching : An evaluation of Falcor and Relay+GraphQL. *SKOLAN FÖR INFORMATIONS* (2016).

[4] Facebook. 2015. GraphQL Specification Versions. https://graphql.github.io/graphql-spec/ Último acesso 4 de Julho 2019.

[5] Roy T Fielding and Richard N Taylor. 2000. *Architectural styles and the design of network-based software architectures*. Vol. 7. University of California, Irvine Doctoral dissertation.

[6] Roy T. Fielding and Richard N. Taylor. 2000. Principled Design of the Modern Web Architecture. In *Proceedings of the 22Nd International Conference on Software Engineering (ICSE '00)*. 407–416.

[7] The Apache Software Foundation. 2002. ApacheAb. https://httpd.apache.org/ Último acesso 4 de Julho 2019.

[8] Florian Haupt, Frank Leymann, and Cesare Pautasso. 2015. A conversation based approach for modeling REST APIs. In *2015 12th Working IEEE/IFIP Conference on Software Architecture*. IEEE, 165–174.

[9] Ana Ivanchikj, Cesare Pautasso, and Silvia Schreier. 2018. Visual modeling of RESTful conversations with RESTalk. *Software and System Modeling* 17, 3 (2018), 1031–1051. https://doi.org/10.1007/s10270-016-0532-2

[10] Nicolai M. Josuttis. 2007. *SOA in Practice:The Art of Distributed System Design*. Vol. 1.

[11] S. Kumari and S. K. Rath. 2015. Performance comparison of SOAP and REST based Web Services for Enterprise Application Integration. In *2015 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*. 1656–1660. https://doi.org/10.1109/ICACCI.2015.7275851

[12] Welder Pinheiro Luz, Everton Agilar, Marcos César de Oliveira, Carlos Eduardo R. de Melo, Gustavo Pinto, and Rodrigo Bonifácio. 2018. An experience report on the adoption of microservices in three Brazilian government institutions. In *Proceedings of the XXXII Brazilian Symposium on Software Engineering, SBES 2018, Sao Carlos, Brazil, September 17-21, 2018*. 32–41.

[13] R. Mizouni, M. A. Serhani, R. Dssouli, A. Benharref, and I. Taleb. 2011. Performance Evaluation of Mobile Web Services. In *2011 IEEE Ninth European Conference on Web Services*. 184–191. https://doi.org/10.1109/ECOWS.2011.12

[14] Guy Pardon and Cesare Pautasso. 2014. Atomic distributed transactions: a RESTful design. In *23rd International World Wide Web Conference, WWW '14, Seoul, Republic of Korea, April 7-11, 2014, Companion Volume*. 943–948. https://doi.org/10.1145/2567948.2579221

[15] Cesare Pautasso and Olaf Zimmermann. 2018. The Web as a Software Connector: Integration Resting on Linked Resources. *IEEE Software* 35, 1 (2018), 93–98. https://doi.org/10.1109/MS.2017.4541049

[16] Cesare Pautasso, Olaf Zimmermann, and Frank Leymann. 2008. Restful Web Services vs. "Big"' Web Services: Making the Right Architectural Decision. In *Proceedings of the 17th International Conference on World Wide Web (WWW '08)*. ACM, 805–814. https://doi.org/10.1145/1367497.1367606

[17] Matheus Seabra. 2019. GraphQL Specification Versions. https://graphql.github.io/graphql-spec/ Último acesso 4 de Julho 2019.

[18] Socket.IO. 20015. Realtime application framework (Node.JS server). https://socket.io/ Último acesso 4 de Julho 2019.

[19] K. Suryanarayanan and K. J. Christensen. 2000. Performance evaluation of new methods of automatic redirection for load balancing of Apache servers distributed in the Internet. , 644–651 pages. https://doi.org/10.1109/LCN.2000.891111

[20] Ruben Taelman, Miel Vander Sande, and Ruben Verborgh. 2018. GraphQL-LD: Linked Data Querying with GraphQL.. In *International Semantic Web Conference (P&D/Industry/BlueSky)*.

[21] Russell Lang Dave Kotz Thomas Williams, Colin Kelley. 2019. Gnuplot. http://www.gnuplot.info Último acesso 4 de Julho 2019.

[22] Xinyang Feng, Jianjing Shen, and Ying Fan. 2009. REST: An alternative to RPC for Web services architecture. In *2009 First International Conference on Future Information Networks*. 7–10. https://doi.org/10.1109/ICFIN.2009.5339611