Pesquisa experimental acerca de tratamentos para latência em ambientes de microsserviços

Mateus Zucco[1], Hugo André Klauck[1]

[1]Instituto Federal Do Rio Grande Do Sul - (IFRS) - Campus Farroupilha
95174-274 - Farroupilha - RS - Brasil

mateus.zucco@aluno.farroupilha.ifrs.edu.br,
hugo.klauck@farroupilha.ifrs.edu.br

Abstract. This study addresses microservices architecture, highlighting its challenges related to intra-service communication, which is essentially hampered by latency. This critical issue jeopardizes application functionality by degrading the end-user experience. Due to this factor, the main objective of this study is to obtain results, through experimental research, on two possible solutions that eliminate latency or mitigate its damage in microservice applications. The possible solutions tested are: distributed cache, worked through the Redis database, and HTTP/2-based RPC communication, worked through the gRPC tool. The functionalities of these two techniques were subjected to collective tests, generating data in three different experiments, which were evaluated and compared to the initial model, considering the metrics of average latency, P50, P90, P99, and total responses with status 2**. Finally, through the analysis of the data obtained, it was possible to evaluate the gains generated by the implemented treatment, with emphasis on the model that experiences latency in the data query layer and has a higher response load.

Resumo. Este presente estudo aborda a arquitetura de microsserviços, destacando seus desafios relacionados à comunicação intrasserviços, dificultada essencialmente pela latência. Este ponto crítico coloca em risco a funcionalidade das aplicações pela degradação da experiência final do usuário. Devido a este fator, o objetivo principal deste estudo é obter resultados, através de pesquisa experimental, acerca de duas possíveis soluções que eliminem a latência, ou mitiguem seus danos em aplicações de microsserviços. As possíveis soluções experimentadas são: cache distribuído, trabalhado através do banco de dados Redis e a comunicação RPC baseada em HTTP/*2*, trabalhada através da ferramenta gRPC. As funcionalidades destas duas técnicas submeteram-se a testes coletivos, gerando dados, em três diferentes experimentos, que foram avaliados e comparados ao modelo inicial, considerando as métricas de latência média, P50, P90, P99 e total de respostas com status 2**. Por fim, através da análise dos dados obtidos, foi possível avaliar os ganhos gerados pelo tratamento implementado, com destaque para o modelo que presencia latência na camada de consulta de dados e possui carga de resposta mais elevada.

1. Introdução

Nas últimas décadas, a arquitetura de microsserviços vem ganhando destaque e se consolidando como uma das principais escolhas no desenvolvimento de sistemas modernos. Caracterizada pela divisão de uma aplicação em pequenos serviços

independentes, cada um com uma função específica, essa abordagem traz inúmeras vantagens, como maior flexibilidade e modularidade, se comparada a arquitetura monolítica. (FOWLER; LEWIS, 2014)

Apesar de suas vantagens, este modelo arquitetônico, também, apresenta alguns obstáculos em sua utilização. Sendo considerada um dos principais desafios na arquitetura de microsserviços, a latência caracteriza-se pelo excesso de tempo em que o servidor, ou serviço, leva para processar uma requisição, desde a sua chegada até sua entrega. (BJØRNDAL et al., 2021) A presença de um tempo para resposta em aplicações baseada em serviços que se comunicam entre si é inevitável, devido ao fato de que toda comunicação em rede necessita de um tempo para transporte e processamento de dados, porém a acentuação deste tempo de resposta causa fortes prejuízos às aplicações. (BJØRNDAL et al., 2021)

Visando auxiliar a indústria de software no tratamento da problemática de latência na arquitetura estudada, este estudo investiga, de maneira experimental, possíveis técnicas para mitigar os impactos, ou resolver a causa, deste desafio que afeta a arquitetura de microsserviços. Baseando-se em dados colhidos e analisados através de dois modelos de testes implementados em um ambiente controlado e hospedado em nuvem, esta pesquisa busca realizar experimentos com duas possíveis soluções: cache distribuído, via database-in-cache, que possibilita o armazenamento de dados em memória de alta eficiência, e comunicação eficiente, via HTTP/2, que é considerada uma versão mais atualizada e eficiente do protocolo de transferência de hipertexto (HTTP). Os métodos testados foram avaliados de maneira coletiva, levando em consideração métricas de diminuição no tempo de resposta e total de requisições bem sucedidas.

O decorrer da pesquisa compõe-se dos capítulos de: referencial teórico contendo tópicos que contextualizam a arquitetura de microsserviços e o desafio presente nela; metodologia, apresentando os modelos iniciais utilizados para experimentos, juntamente com a descrição de suas tecnologias utilizadas e as soluções aplicadas; desenvolvimento, descrevendo a implementação e evolução do modelo inicial; resultados obtidos, explanando as métricas, dados e análises utilizados/gerados em cada experimento; finalizando com as conclusões, resumindo aquilo que foi possível produzir e avaliar no decorrer desta pesquisa, além de possíveis pesquisas futuras. Por fim, são apresentadas as referências bibliográficas utilizadas e os anexos do texto.

2. Referencial teórico

Para tratar dos aspectos propostos por esta pesquisa, é essencial começar com um estudo introdutório sobre a arquitetura de microsserviços, seguido por uma contextualização acerca dos desafios relacionados à latência deste modelo. Esse levantamento permite destacar os principais conceitos da arquitetura, suas distinções em relação à modelagem tradicional, sua relevância no contexto dos softwares modernos e seus principais desafios.

2.1. Principais conceitos da arquitetura de microsserviços

A arquitetura de microsserviços trata de um conceito referente à modelagem que um software pode tomar, baseando-se na divisão da aplicação em diversas unidades

independentes, denominadas de serviços, que podem ou não se conectarem umas com as outras.

De acordo com Thönes (2015):

> Um microsserviço, na minha opinião, é uma pequena aplicação que pode ser implantada de forma independente, escalada de forma independente e testada de forma independente e que tem uma única responsabilidade. É uma única responsabilidade no sentido original de que tem um único motivo para mudar e/ou um único motivo para ser substituída. Mas o outro eixo é uma única responsabilidade no sentido de que faz apenas uma coisa e só uma coisa e pode ser facilmente compreendida. (THÖNES, 2015, p.116, tradução própria).[1]

O autor referido conceituou os microsserviços como aplicações de pequeno porte, projetadas para serem implantadas, escaladas e testadas de forma autônoma. Sendo assim, esta arquitetura trata de um modelo oposto à arquitetura tradicionalista presente nas aplicações monolíticas em que "toda a sua lógica para lidar com uma solicitação é executada em um único processo [...]"(FOWLER; LEWIS, 2014, tradução própria).[2]

Outrossim, os microsserviços caracterizam-se através da descentralização e fragmentação da aplicação, organizando o software de uma maneira modularizada em unidades autônomas, sendo cada unidade projetada para possuir independência, sem a necessidade obrigatória da integração direta com as outras, embora essa interconexão possa ser realizada.

2.1.1. Diferenças em relação à modelagem tradicional

Enquanto a modelagem tradicional costuma reunir todas as funcionalidades em uma estrutura integrada, conhecida como monólito, os microsserviços se concentram em dividir o sistema em módulos menores e independentes. De acordo com Mendonça et al. (2021, p. 17 - 18), os microsserviços e as arquiteturas monolíticas diferem significativamente em sua estrutura e abordagem de desenvolvimento. Enquanto os microsserviços são compostos por unidades independentes, que permitem maior agilidade e flexibilidade, os sistemas monolíticos integram várias funções em uma única unidade, o que pode limitar a escalabilidade e a eficiência dos testes.

No que tange à parte estrutural das arquiteturas, verifica-se uma grande diferença na disposição do(s) processo(s) e na escalabilidade dos servidores de cada metodologia, conforme exemplificado pela Figura 1.

---

[1] "A microservice, in my mind, is a small application that can be deployed independently, scaled independently, and tested independently and that has a single responsibility. It is a single responsibility in the original sense that it's got a single reason to change and/or a single reason to be replaced. But the other axis is a single responsibility in the sense that it does only one thing and one thing alone and can be easily understood."

[2] "All your logic for handling a request runs in a single process [...]"

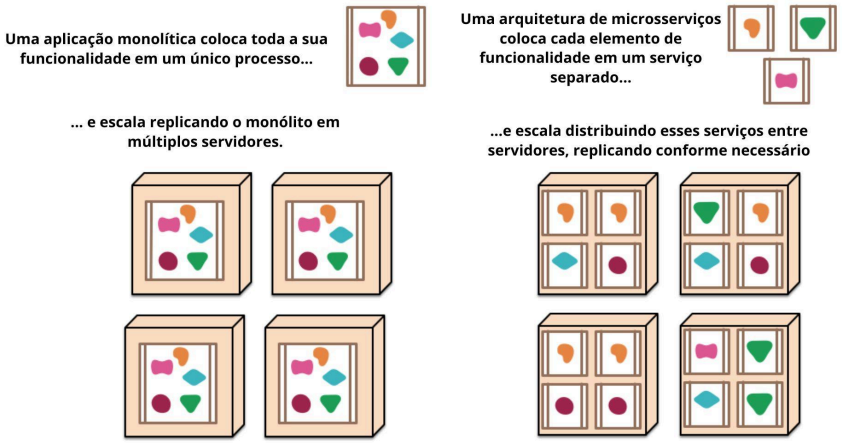

Figura 1. Monólitos e microsserviços

Fonte: (FOWLER; LEWIS, 2014, tradução própria)

Na representação do sistema monolítico, no qual todos os componentes estão em uma única estrutura, identifica-se todas as funcionalidades da aplicação em um único processo, que necessita de múltiplos servidores, para serem escalonadas. Em contraste, a parte que ilustra a arquitetura de microsserviços identifica cada componente em um serviço independente, permitindo que eles sejam escalonados de maneira autônoma em servidores distribuídos, mantendo a sua modularidade e ao mesmo tempo sendo organizados em conjunto.

2.2. Relevância no contexto dos softwares modernos

O termo "microsserviços" começou a ser utilizado e difundido de forma clara por meados de 2010. Desde então, tanto o termo quanto o conceito dessa arquitetura vêm se consolidando progressivamente no mercado de software. (FOWLER; LEWIS, 2014)

Como retratou o autor Fowler e Lewis (2014):

> O circuito de conferências em 2013 estava cheio de exemplos de empresas que estão se movendo para algo que poderia ser classificado como microsserviços [...]. Além disso, há muitas organizações que há muito tempo estão fazendo o que classificaríamos como microsserviços, mas sem nunca usar o nome. (Muitas vezes isso é rotulado como SOA - embora, como dissemos, SOA venha em muitas formas contraditórias.) (FOWLER; LEWIS, 2014, tradução própria).[3]

Esse trecho evidencia que, já em 2013, mais de uma década antes desta pesquisa, diversas empresas se posicionavam para criar ou migrar suas aplicações para a arquitetura de microsserviços. Ademais, ele reforça que muitas companhias já estruturavam seus softwares de acordo com os princípios dos microsserviços, embora não empregassem formalmente essa nomenclatura.

---

[3] "The conference circuit in 2013 was full of examples of companies that are moving to something that would class as microservices [...]. In addition there are plenty of organizations that have long been doing what we would class as microservices, but without ever using the name. (Often this is labelled as SOA - although, as we've said, SOA comes in many contradictory forms.)"

Em concordância, o pesquisador Thönes cita um exemplo emblemático envolvendo uma das maiores empresas do mercado de software: “O exemplo canônico é a Amazon. A Amazon começou com um grande banco de dados e depois mudou para uma arquitetura orientada a serviços.” (THÖNES, 2015, p. 113, tradução própria).[4]

Provando, assim, que os microsserviços estão não só sendo aceitos por grandes companhias do mercado, mas também estão sendo utilizados para substituir arquiteturas tradicionais que não oferecem as mesmas vantagens para determinadas aplicações.

2.3. Desafios existentes na arquitetura

Embora haja diversos pontos positivos na arquitetura de microsserviços, como escalabilidade e independência modular, toda e qualquer metodologia adotada para um sistema irá apresentar desafios que precisam ser considerados. Como elucidado pelo artigo *No Silver Bullet - Essence and Accident in Software Engineering*:

> Mas, ao olharmos para o horizonte de uma década à frente, não vemos nenhuma solução mágica. Não há nenhum desenvolvimento único, seja em tecnologia ou técnica de gestão, que por si só prometa até mesmo uma ordem de magnitude de melhoria em produtividade, confiabilidade e simplicidade. (BROOKS, 1987, p.2, tradução própria).[5]

Assim, compreende-se que no ambiente da engenharia de software, não há uma solução instantânea que possa ser empregada. Como comenta o cientista da computação, David Parnas, após 20 anos da publicação de BROOKS, em (FRASER e MANCL, 2008, tradução própria): “[...] Por que falar sobre isso quando 20 anos provaram que ele estava certo? Projetar software é difícil e sempre será difícil. Não há respostas fáceis aqui!”[6]. Demonstrando que apesar do passar das décadas, a construção de aplicações persiste sendo um desafio para os desenvolvedores de sistemas.

Concordando com esta teoria, a arquitetura de microsserviços, apesar de seus benefícios, também apresenta pontos problemáticos, que geram dificuldades em sua implementação e utilização. Pela natureza da arquitetura ser modularizada em pequenas unidades de serviço, a comunicação interna entre essas unidades gera um desafio contínuo para o desempenho da aplicação.

Considerando o desempenho e a constância algo vital para o tempo de vida útil dos softwares, a latência presentes nos sistemas que utilizam da arquitetura de microsserviços deve ser levado em consideração para todos aqueles que optem por essa modelagem.

---

[4] “The canonical example is Amazon. Amazon started with a big database and then moved to a service oriented architecture”

[5] “But, as we look to the horizon of a decade hence, we see no silver bullet. There is no single development, in either technology or management technique, which by itself promises even one order of magnitude improvement in productivity, in reliability, in simplicity. “

[6] Why was it necessary for Fred to write ‘No Silver Bullet?’ Why talk about it when 20 years have proved it right? Designing software is hard and will always be hard. No easy answers here!

2.3.1. Latência na comunicação

Em comparação com as aplicações monolíticas, a arquitetura de microsserviços é mais sensível à latência de rede. A identificação de violações de tráfego na rede, como latência média ou máxima acima do limite, é um desafio crescente, já que os atrasos podem propagar-se rapidamente para as demais unidades, resultando na desaceleração da aplicação como um todo. (GAN et al., 2019 apud ZHU et al., 2022, p. 4818 - 4819)

Sendo presenciada durante atrasos na comunicação dos serviços, a latência forma uma relação exponencial entre seus atrasos e suas complicações. Como demonstrado pelos gráficos das Figuras 2 e 3, resultados de um experimento comparativos entre diferentes arquiteturas, simulando um sistema bibliotecário:

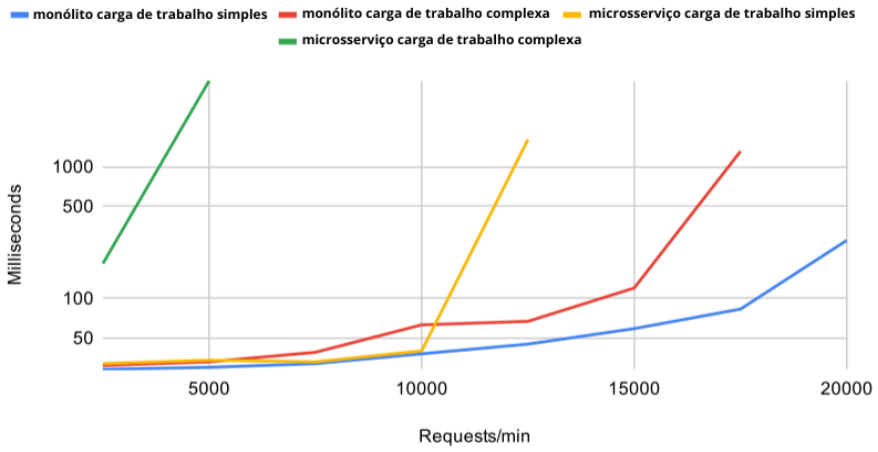

Figura 2. Latência por requisições/minuto

Fonte: (BJØRNDAL et al., p. 9, 2021, tradução própria)

Através da Figura 2, é possível perceber como a latência, dos modelos de microsserviços testados, acentua-se de maneira precoce, atingindo o topo da latência do gráfico, com apenas 5000 requisições por minuto. Em contraponto, seu equivalente na arquitetura monolítica, tem uma crescente exponencial em sua latência, somente após as 15000 requisições por minuto.

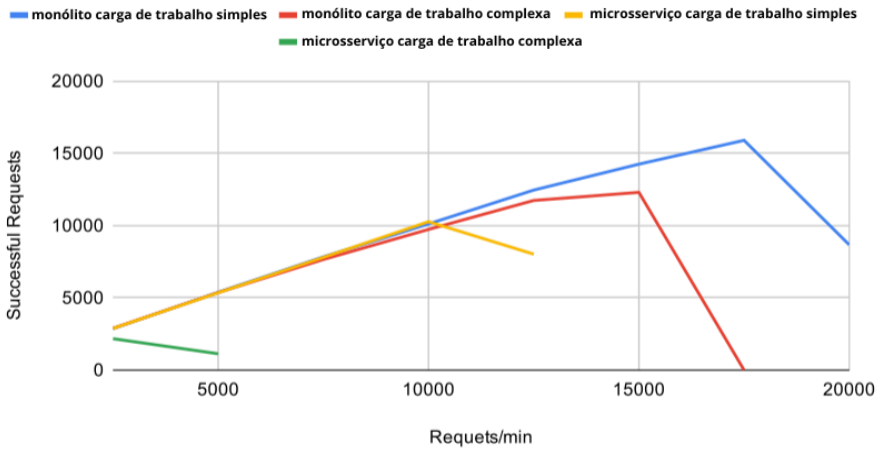

Figura 3. Solicitações com sucesso por requisições/minuto

Fonte: (BJØRNDAL et al., p. 9, 2021, tradução própria)

Conforme demonstra a Figura 3, há um grande enfraquecimento na quantidade de requisições concluídas com sucesso, em ambos modelos de microsserviços, destacando o fato do sistema de arquitetura monolítica possuir um resultado 1,5 vezes melhor em relação a arquitetura de microsserviços com carga de trabalho complexa[7]. (BJØRNDAL et al., p. 9, 2021)

Assentindo com estes dados, ao alinhar a Figuras 2 com a Figura 3 é possível compreender a relação linear entre as métricas de latência média e throughput[8], testada com um modelo de carga de trabalho simples[9] e um modelo de carga de trabalho complexa.

Ambos gráficos apresentam direções exponencialmente opostas para suas linhas do eixo Y, partindo do eixo X em zero, para microsserviços de carga complexa, e do eixo X em 10000, para microsserviços de carga simples. Desta forma, entende-se que a latência é um fator crítico em arquiteturas de microsserviços, pois está diretamente relacionada ao total de requisições realizadas com sucesso.

3. Metodologia

Através desta seção, busca-se caracterizar e apresentar o modelo proposto em seus diferentes estágios que foram submetidos aos experimentos, juntamente com as tecnologias responsáveis pelo seu desenvolvimento e sua implementação, acompanhadas de um breve resumo teórico de cada uma delas.

3.1. Tecnologias

Os serviços foram desenvolvidos utilizando a linguagem TypeScript[10], derivada da linguagem JavaScript[11], porém com a adição de tipagem fortemente acoplada às variáveis, permitindo uma maior confiança e robustez na aplicação, mantendo curva de aprendizagem relativamente atrativa. (MICROSOFT, s.d.)

A implementação de TypeScript em ambiente de servidor, ocorreu através do ambiente de execução Node.js[12], que graças ao mecanismo JavaScript V8[13] possibilitou a implementação de JavaScript, com boa performance, mesmo fora de navegadores. (Nodejs)

O roteamento dos serviços utilizou o framework Express.js[14], permitindo a construção de servidores web, através do Node.js. (EXPRESS, s.d.)

---

[7] Neste experimento o autor define carga de trabalho simples ações como: coletar dados sobre um empréstimo, criar pedido e criar um livro. (BJØRNDAL et al., p. 8, 2021)

[8] Tradução: rendimento.Taxa de solicitações processadas.

[9] Neste experimento o autor define carga de trabalho simples ações como: coletar dados de livro e criar usuário. (BJØRNDAL et al., p. 8, 2021)

[10] Disponível em: https://www.typescriptlang.org/

[11] Linguagem de script, para navegadores

[12] Disponível em: https://nodejs.org/en

[13] Disponível em: https://v8.dev/

[14] Disponível em: https://expressjs.com/pt-br/

A armazenagem de dados construiu-se de maneira unitária, através de bancos de dados individuais por serviços, utilizando a tecnologia MySQL[15], para gerenciamento dos bancos de dados relacionais. (ORACLE, s.d.)

Os modelos para experimentos foram hospedados na plataforma de computação em nuvem GCloud[16], fornecida pela *Google*. Executando a aplicação por meio do serviço, GCloud Cloud Run[17], que permite o gerenciamento e implantação de contêineres em sua infraestrutura, e as bases de dados foram hospedadas por meio do GCloud Cloud SQL[18], que possibilitou a implantação de bancos de dados relacionais. (GOOGLE CLOUD, s.d.) Além disso, a implementação da ferramenta Redis[19], em ambiente de nuvem, utilizou o serviço GCloud MemoryStore[20] para o gerenciamento dos dados armazenados.

Por fim, as coletas resultantes dos experimentos ocorreram através da ferramenta Autocannon[21], que realiza testes de performance em rotas *HTTP(s)*.

3.2. Estrutura base

Como definido durante o referencial teórico, as aplicações de microsserviços caracterizam-se através da divisão da aplicação em diversas unidades independentes, que geralmente, se comunicam entre si.

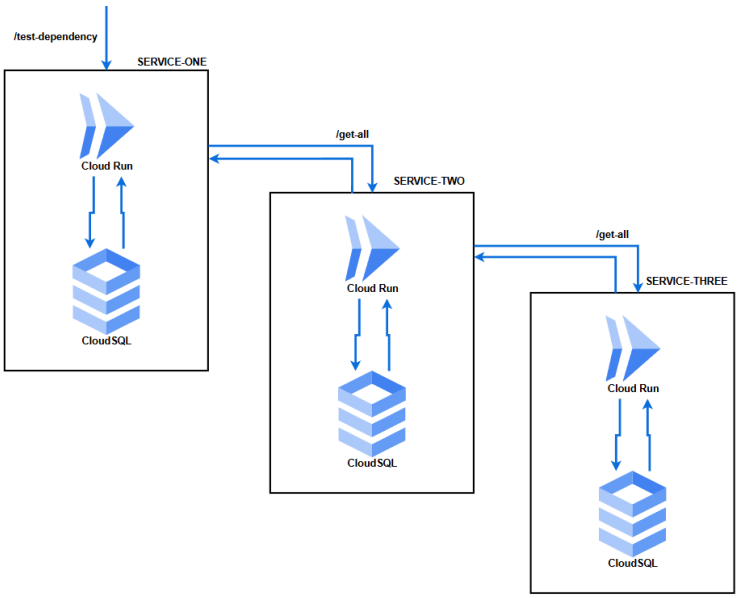

Figura 4. Modelo de alta dependência

Fonte: Autoria própria

[15] Disponível em: https://www.mysql.com/

[16] Disponível em: https://cloud.google.com

[17] Disponível em: https://cloud.google.com/run

[18] Disponível em: https://cloud.google.com/sql

[19] Disponível em: https://redis.io

[20] Disponível em: https://cloud.google.com/memorystore

[21] Disponivel em: https://www.npmjs.com/package/autocannon

Através desta arquitetura de software, foram propostos dois modelos: o primeiro é demonstrado através da Figura 4, sendo formado por uma cadeia de três serviços conectados, formando um grau de dependência entre eles, denominado de “modelo de alta dependência”.

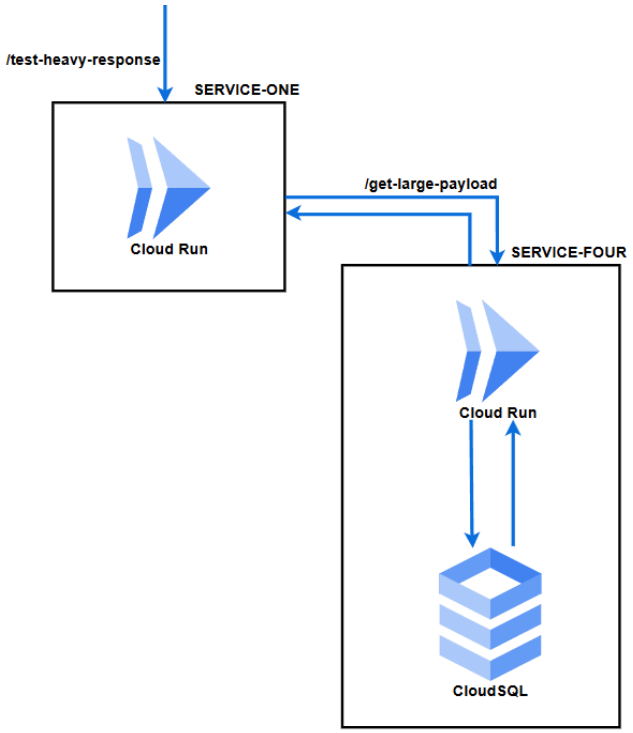

Figura 5. Modelo de resposta custosa

Fonte: Autoria própria

O segundo modelo, representado pela Figura 5, foi composto por dois serviços conectados em um cenário em que o serviço filho tem uma consulta e resposta propositalmente custosa a aplicação, denominado de “modelo de resposta custosa”. Estes modelos, foram arquitetados de maneira proposital a fim de expor a latência presente em altas cargas de trabalho.

Os modelos formaram-se por quatro serviços, sendo eles: service-one, service-two, service-three e service-four. Sendo o service-one responsável por receber todas as requisições em ambos os modelos, o service-two e service-three responsáveis por receber requisições e retornar dados no modelo de alta dependência e service-four responsável por executar consultas mais custosa ao banco de dados e retornar respostas fatigantes ao serviço.

Através de ambos modelos, foi possível realizar experimentos em dois ambientes de latência distintos, primeiramente pelo modelo de alta dependência foi possível examinar a latência decorrida da cadeia de dependências dos serviços, segundamente pelo modelo de resposta custosa tornou-se possível analisar a latência decorrida do gargalo na coleta e no transporte de dados.

3.3. Soluções propostas

A primeira solução proposta a ser abordada neste trabalho trata-se da técnica de cache[22] distribuído, que foi implementada através da ferramenta Redis, um banco de dados em memória de alto desempenho, proposto no modelo chave-valor. Através desta ferramenta foi possível a pulverização do armazenamento dos dados em memória em diversos nós da rede, visando melhorias de escalabilidade, tolerância a falhas e redução do tempo de resposta. (REDIS)

Como segunda solução proposta, este artigo traz o framework gRPC[23], utilizado para chamadas de procedimento remoto, possibilitando a conexão eficiente entre as diversas unidades do ambiente de microsserviços, através da transmissão bidirecional, entre os serviços, utilizando de transporte de dados baseado em HTTP/2. (GRPC)

Assim, através destas duas propostas de solução, este presente estudo busca reduzir os atrasos de rede a ponto de mitigar os principais impactos que a latência causa na arquitetura de microsserviços.

4. Desenvolvimento

Neste capítulo busca-se contextualizar a aplicação prática da metodologia, a fim de descrever os processos utilizados na formação dos modelos de testes.

4.1. Modelos iniciais

Para os modelos iniciais dos experimentos, desenvolveu-se todos os quatro microsserviços utilizando Typescript, como linguagem de programação, Node.js ambiente de execução, juntamente com Express, como roteador dos serviços e MySQL como banco de dados relacional.

Assim, todos os serviços foram desenvolvidos para cumprir suas funções conforme proposto pelos modelos demonstrados nas Figuras 4 e 5, tendo como ponto de entrada o service-one. Nele foram implementadas duas rotas, sendo cada uma delas responsável por um modelo de experimento, sendo elas:

- /test-dependence que desencadeia em uma chamada ao banco de dados dentro do service-one e uma chamada /get-all ao service-two, que por sua vez realiza uma chamada ao seu banco de dados e também uma requisição ao /get-all do service-three que, por fim, retorna os dados coletados em sua base de dados. Formando assim, uma cadeia de dependência, conforme demonstrado pela Figura 4.
- /test-heavy-response qual gera uma chamada do service-one ao service-four, que executa consultas ao seu banco de dados, por meio de uma transaction[24] e retorna uma resposta contendo um JSON[25] de

[22] Forma de armazenar dados com alta eficiência.

[23] Disponível em: https://grpc.io/

[24] Tradução: transação. Técnica utilizada para executar um ou mais comandos SQL em uma única operação

[25] Notação de Objetos JavaScript. Formato para transporte de dados baseado em texto.

aproximadamente 70.000 linhas e aproximadamente 1.66 MB de carga, conforme demonstrado pela Figura 5.

Para o andamentos dos experimentos, cada serviço foi hospedado na plataforma do GCLOUD RUN sendo configurado com um processador de duas CPUs, mínimo de zero instâncias e máximo de quatro instâncias, suporte a 40 requisições simultâneas, um GiB de memória, 3600 segundos de tempo limite em cada solicitação e hospedagem na região us-central1.

Visando formar as estruturas e população de dados presentes nos testes, todos os bancos de dados foram criados com uma tabela denominada users[26], contendo 1000 linhas cadastradas, além disso, para o banco de dados do service-four foi criado a tabela users-access-historic, contendo 2000 linhas cadastradas, conforme esquematizados pela Figura 6.

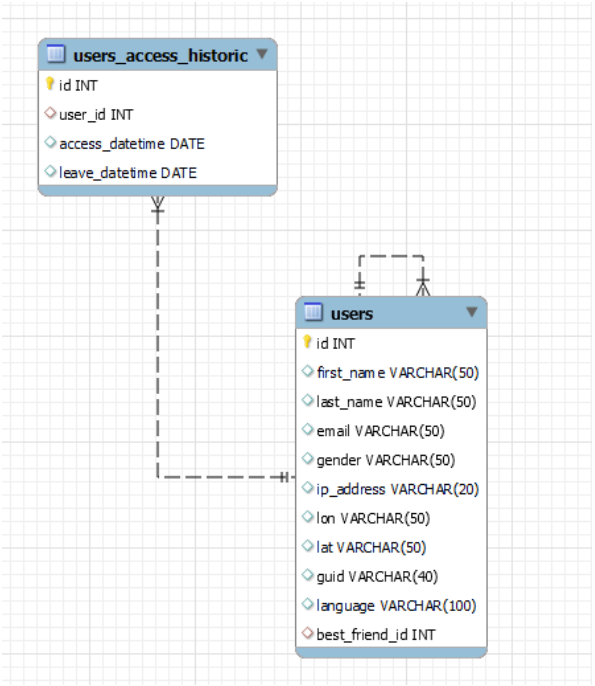

Figura 6. Modelo ER do service-four

Fonte: Autoria própria

Além destas estruturas, as bases de dados foram hospedadas na plataforma GCLOUD SQL, contendo as seguintes configurações: banco de dados na versão MySQL 8.0.41, máquina tipo db-f1-micro, um vCPU, 628,74 MB de memória RAM, 10 GB de memória em SSD e hospedagem na região us-central1.

Através destes processos de programação e hospedagem em nuvem foi possível estabelecer um ambiente de alta disponibilidade e eficiência, que não gerasse anomalias significativas nas coletas de amostras de dados.

[26] Tradução: usuários

4.2. Modelos tratados

Tomando como base os modelos iniciais propostos, foram implementados dois tratamentos para serem testados nos experimentos. Visando diminuir o tempo de espera para a consulta aos dados das bases relacionais, foi proposto a ferramenta Redis, e buscando diminuir a latência gerada na camada de comunicação dos serviços, foi oferecido a solução gRPC.

O primeiro tratamento implementado foi o Redis, tratando-se de um banco de dados de alta eficiência, sua responsabilidade seria de armazenar os dados vindos da base de dados MySQL em uma memória de alta eficiência, a fim de diminuir o total de consultas do CLOUD RUN ao CLOUD SQL e consequentemente diminuir o tempo de espera.

Sua implementação foi realizada no model[27] de cada serviço e seu tempo para expiração foi definido para 10 segundos, para que sua utilização não distorcesse os testes de maior duração e tornassem os resultados mais plausíveis com a realidade.

Além destas mudanças no código dos serviços, também foi estruturada uma instância do MemoryStore na plataforma GCloud para suportar o armazenamento dos dados em memória.

Como segunda solução adicionada, foi implementado o framework gRPC para a utilização do protocolo RPC nas chamadas aos outros serviços e aplicação do protocolo HTPP/2 para o transporte de dados serializados utilizando Protocol Buffers[28].

A solução foi inserida ao código substituindo a arquitetura REST dos serviços intermediários, por clientes e servidores gRPC que realizavam comunicação através de chamadas utilizando arquivos .proto[29].

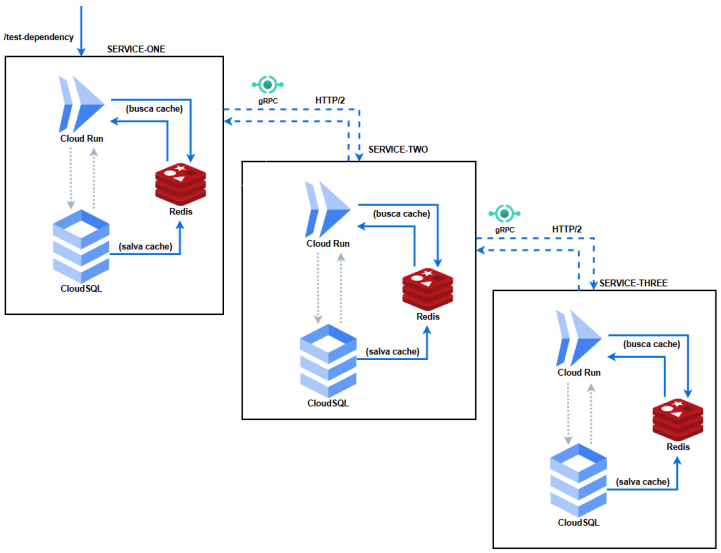

Figura 7. Modelo de alta dependência tratado

Fonte: Autoria própria

---

[27] Parte da arquitetura MVC que lida com a lógica de negócio e coleta de dados

[28] Tecnologia utilizada para serializar dados estruturados

[29] Extensão de arquivos de Protocol Buffers

Com ambas soluções implementadas aos modelos iniciais, conforme exibe a Figura 7, a arquitetura dos modelos mantiveram suas propostas e funções previamente estabelecidas, porém com novas estruturas para a coleta de dados e novos canais de comunicação para o transporte de dados.

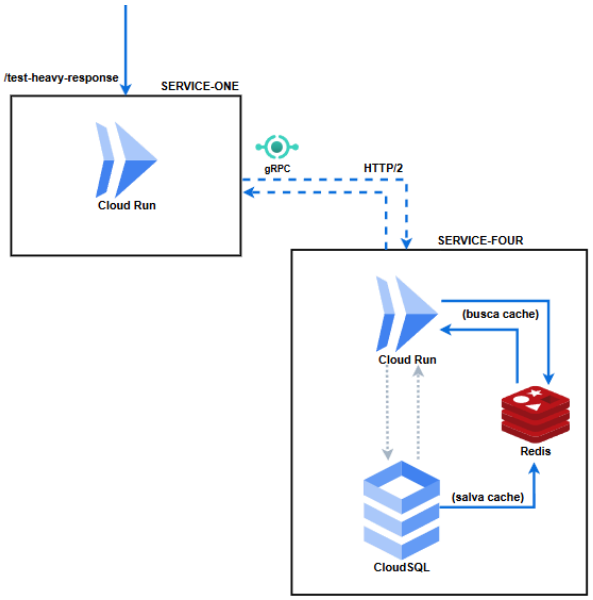

Figura 8. Modelo de resposta custosa tratado

Fonte: Autoria própria

Desta maneira, ao realizar comparações entre a Figura 8 e a Figura 5, pode-se perceber que em relação aos modelos iniciais as bases de dados hospedadas no GCLOUD SQL alimentam as bases de dados Redis, que tornaram-se a principal fonte de consulta de dados dos serviços, além disso, também é perceptível que toda a comunicação intra-serviço foi substituída por HTTP/2 através do gRPC.

4.3. Definição dos experimentos

Com intuito de diversificar os experimentos, para obter contextos mais condizentes com a realidade, foram elencados três cenários para cada modelo, mesclando as variáveis connections, que define o total de conexões aplicadas no teste, duration, que limita o tempo total do teste, e amount, responsável por fixar o total de respostas que o teste aguarda.

Desta forma, como demonstrado pelos Quadros 1 e 2, foram elencados seis experimentos, sendo três para cada modelo, além de cada experimento submeter-se a três coletas de amostras de resultados, totalizando 18 coletas.

	Connections	Duration	Amount
Coleta base	50	10 segundos	máximo possível
Estresse sustentado	80	60 segundos	máximo possível
Por quantidade	65	mínimo possível	6000

Quadro 1. Configuração de testes para modelo de alta dependência

Fonte: Autoria própria

	Connections	Duration	Amount
Coleta base	5	20 segundos	máximo possível
Estresse sustentado	25	60 segundos	máximo possível
Por quantidade	10	mínimo possível	150

Quadro 2. Configuração de testes para modelo de resposta custosa

Fonte: Autoria própria

Os experimentos foram denominados de: coleta base, representando apenas uma coleta inicial de dados, estresse sustentado, buscando atingir os níveis máximos suportados pelos modelos, e por quantidade, que também seguia buscando atingir os níveis máximos suportados, porém com o limite sendo estabelecido pela quantidade de requisições ao invés do tempo total. As coletas foram feitas utilizando a ferramenta Autocannon, em diferentes momentos e com todos os serviços já aquecidos, para evitar interferências de latência na inicialização dos servidores.

4.4. Métricas utilizadas

Através das três coletas que cada experimento se submeteu, foi calculado a média aritmética dos seguintes dados:

- latência média: caracterizada pela média aritmética da latência presente em todas as requisições realizadas.
- latência percentil 50 (P50): valor qual divide o conjunto de dados exatamente ao meio, evitando a interferência de valores extremos em ambas as pontas do conjunto.
- latência percentil 90 (P90): indicador sob os 90% maiores valores do conjunto de dados.
- latência percentil 99 (P99): ponto superior a 99% dos dados do conjunto, representando 1 a cada 100 casos.

- média de respostas com status 2** (MR2xx): demonstra a média aritmética do total de requisições que retornaram sucesso durante o teste.

Por meio destas medidas foi possível avaliar os dados gerados nos testes de maneira que permitiu a análise da latência presente nos experimentos em diferentes partes do conjunto de dados obtidos.

5.Resultados

Através deste capítulo espera-se explanar os resultados decorridos do desenvolvimento do projeto, além de, caracterizar as métricas definidas para suas análises, buscando descrever os processos considerados para as avaliações de perdas e ganhos.

5.1.Dados gerados

Utilizando as métricas elencadas, juntamente com as médias dos dados coletadas, foram formadas dois quadros de resultados, sendo uma para cada modelo, divididos por experimento.

	Média (ms)	**P50 (ms)**	**P90 (ms)**	**P99 (ms)**	**MR2xx**
Coleta Base					
Modelo inicial	1628	1659	2188	2983	280
Modelo tratado	1586	1512	2274	3292	293
Estresse Sustentado					
Modelo inicial	1386	1372	2178	4097	3451
Modelo tratado	1196	1108	1892	3205	3981
Por Quantidade					
Modelo inicial	1681	1723	3100	3900	6000
Modelo tratado	2438	809	8939	16476	5361

Quadro 3. Resultados no modelo de alta dependência

Fonte: Autoria própria

Graças aos dados apresentados no Quadro 3, foi possível obter-se uma primeira visão dos ganhos e perdas ocorridos no modelo de alta dependência, comparando as linhas superiores de cada experimento, representando o modelo sem tratamento, com as respectivas linhas inferiores, que representam os modelos com tratamento aplicado.

	Média (ms)	P50 (ms)	P90 (ms)	P99 (ms)	MR2xx
Coleta Base					
Modelo inicial	2144	2167	2976	3701	45
Modelo tratado	1098	749	1969	4041	89
Estresse Sustentado					
Modelo inicial	9036	8351	11518	26533	90
Modelo tratado	4097	3716	10710	12953	293
Por Quantidade					
Modelo inicial	17127	4780	40159	48482	150
Modelo tratado	1825	1311	4666	6646	150

Quadro 4. Resultados no modelo de resposta custosa

Fonte: Autoria própria

Da mesma maneira, através dos dados elencados na Quadro 4, foi possível comparar os resultados que a aplicação do tratamento obteve nos três testes em que o modelo de resposta de custos foi submetido.

5.2. Avaliando ganhos e perdas

Iniciando pelo modelo de alta dependência, os testes base apresentaram uma redução praticamente irrelevante na maioria das métricas, além de uma leve piora na métrica de P99. Já os testes de estresse sustentado no modelo tratado apresentaram melhoras em todas as métricas, especialmente em P99 (ms) e MR2xx. Por fim, indo contra as expectativas, os testes avaliados em quantidade apresentaram pioras significativas, especialmente as métricas MR2xx, P90 e P99.

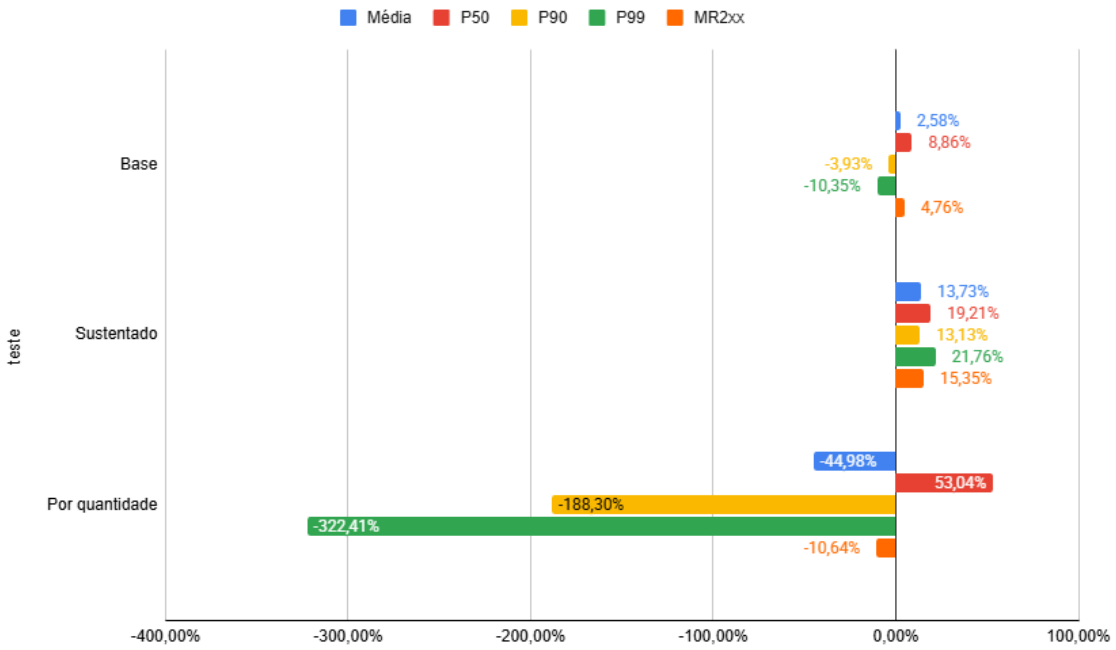

Gráfico 1. Percentuais de ganhos e perdas do tratamento no modelo de alta dependência

Fonte: Autoria própria

Através da comparação gerada pelo Gráfico 1, foi constatado leves melhorias nas métricas de média, P50 e MR2**, dos testes base, apesar de manifestar decadência nos resultados de P90 e P99, demonstrando aumentos de latência presente acima dos 10% piores casos da amostra de dados.

Também foi possível perceber melhorias de mais de 21% em relação à métrica de P99 e 19% na métrica de P50, nos resultados dos testes de estresse sustentado, indicando que o tratamento melhorou consideravelmente a latência na mediana e nos piores casos. Além disso, pode ser analisada uma melhora de mais de 15% em relação ao total de respostas com status de sucesso que o modelo tratado apresentou no teste de estresse sustentado, em relação ao modelo inicial.

Apesar disso, conforme o Gráficos 1 demonstra, os testes por quantidade, que mais realizaram requisições aos serviços, apresentaram perdas severas em relação às médias de P90, que saturou a latência do modelo em mais de 188% e também em relação a P99 que tornou-se 322% mais latente, além da métrica MR2xx que decai em mais de 10% de um modelo para outro. Em contradição, a métrica de P50 melhorou em mais de 53% ao comparado com o modelo inicial, contrariando isso, a latência média piorou em aproximadamente 45%. Estes números apontam que, nestes testes, os resultados presentes nos piores casos da amostra de dados, interferiram significativamente na degradação do resultado de média, já os valores mais ao centro da curva de resultados apresentaram melhoras em relação a diminuição da latência, contribuindo para as melhorias nos resultados da mediana (P50).

Analisando os dados apresentados nos parágrafos acima, foi possível entender que o modelo tratado apresentou melhorias em relação a diminuição dos tempos de

respostas para os testes mais sutis, porém ele possui tendências a perder sua resiliência conforme o número de requisições cresce, tornando sua performance inferior ao modelo sem tratamento, nestes casos.

Seguindo com o teste de resposta custosa, os resultados de redução no tempo de resposta são facilmente perceptíveis em todos os três testes aplicados, especialmente na coleta baseada em quantidade, responsável por ter sido o teste que mais estressou o modelo inicial, como demonstrado pelo Gráfico 2.

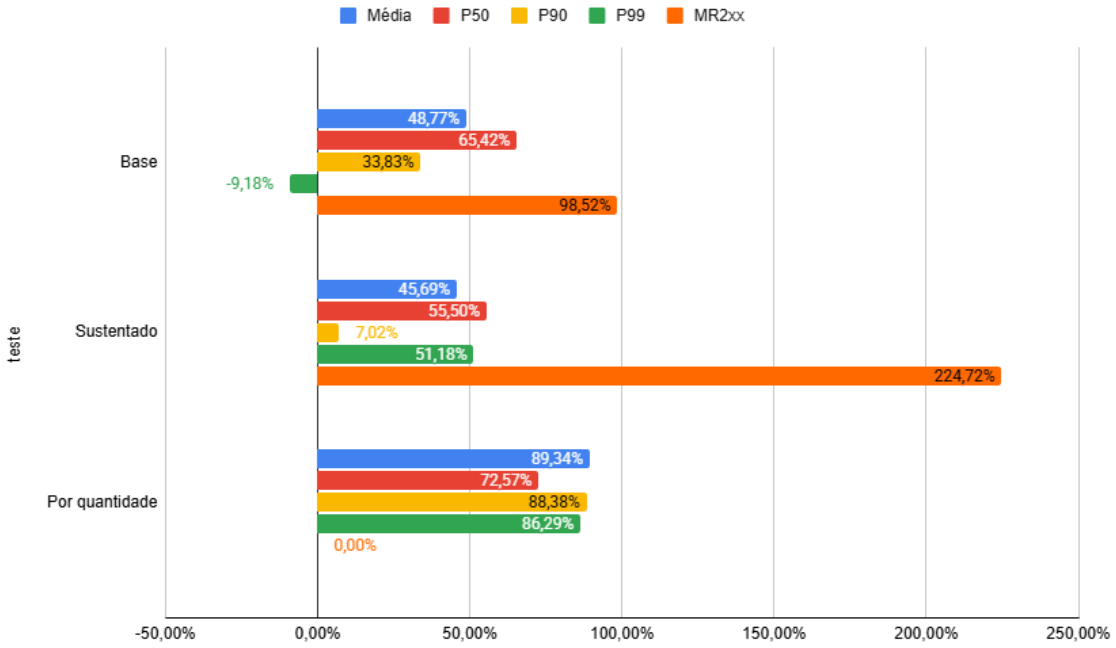

Gráfico 2. Percentuais de ganhos e perdas do tratamento no modelo de resposta custosa

Fonte: Autoria própria

Diferentemente dos resultados do modelo anterior, o Gráfico 2 demonstra que neste modelo a aplicação do tratamento resultou melhorias significativas em praticamente todas as métricas, exceto pelos resultados de P99 nos testes de base.

Iniciando pela coleta base, nela podemos perceber grandes melhorias nas métricas de média, P50, P90 e MR2xx, com destaque para melhorias de quase 99% no total de respostas com status 2**, além de uma queda de mais de 65% no tempo de resposta de P50. Apesar disso, houve um pequeno aumento na métrica de P99, possivelmente causado por algum ponto fora da curva, nos 1% piores casos.

Seguindo com os testes de estresse sustentado, as métricas apresentaram-se extremamente positivas para o modelo tratado, indicando melhorias em todos os pontos avaliados, com grande destaque para reduções de 55% na métrica P50, 45% na latência média e um aumento de aproximadamente 225% no total de respostas com status 2**.

Por fim temos o teste em que o tratamento mais apresentou resultados positivos, durante os experimentos. Os resultados dos testes por quantidade, apresentaram grandes melhorias nas quatro métricas relacionadas a latência e atingiu o total de requisições solicitadas com 100% de respostas com status de sucesso. Neste teste as métricas de

latência média, P50, P90 e P99 apresentaram, respectivamente, reduções superiores a 89%, 72%, 88% e 86%. Destacando-se uma diminuição de 41.836 ms na latência presente no 1% dos piores casos.

Em comparação aos resultados do modelo de alta dependência, os dados obtidos nos testes do modelo de resposta custosa demonstram ganhos consideravelmente superiores, provando que o tratamento aplicado obteve mais performance nos casos em que a latência presente ocorre na camada de consulta de dados e em respostas de carga mais elevada.

6. Conclusões

Ao fim deste trabalho, os tratamentos propostos foram devidamente aplicados aos modelos propostos e os experimentos realizados cumpriram com os principais objetivos propostos, gerando dados capazes de avaliar a combinação das tecnologias Redis e gRPC na diminuição da latência presente em diferentes ambientes de microsserviços.

Os resultados obtidos nos experimentos demonstraram que o tratamento aplicado ao modelo de alta dependência, resultou em leves melhorias no tempo de resposta dos serviços em cenários de poucas requisições, porém apresentaram-se, também, complicações conforme o total de requisições aumentava, resultando e retardos na latência do modelo.

Já os resultados obtidos através do modelo de resposta custosa apresentaram grandes melhorias em todos os experimentos realizados, especialmente nos testes com números volumosos de requisições realizadas. Demonstrando que a aplicação do tratamento obteve maior eficácia em modelos com gargalo presente na consulta aos bancos de dados.

Para pesquisas futuras resta investigar mais aprofundadamente a eficácia individual de cada tratamento aplicado, especialmente nos cenários de maior dependência, a fim de explicar o motivo pelo qual os resultados dos testes neste modelo não foram amplamente positivos. Além disso, novas técnicas e tecnologias podem ser aplicadas aos modelos elencados, visando ampliar a performance da arquitetura, como por exemplo: adição de algoritmos de balanceamento de carga, alterações na granularidade dos serviços, aumento da escalabilidade vertical da aplicação e aplicação de streams nas respostas gRPC.

7. Referências

BJØRNDAL, Nichlas; ARAÚJO, Luiz Jonatã Pires de; BUCCHIARONE, Antonio; DRAGONI, Nicola; MAZZARA, Manuel; DUSTDAR, Schahram. Benchmarks and performance metrics for assessing the migration to microservice-based architectures. Journal of Object Technology, v. 20, n. 2, p. 2:1–17, 2021. DOI: https://doi.org/10.5381/jot.2021.20.2.a3.

BROOKS, F. P., Jr. No Silver Bullet: Essence and Accidents of Software Engineering. Computer, v. 20, n. 4, p. 10-19, abr. 1987. DOI: 10.1109/MC.1987.1663532

EXPRESS. Express - Node.js web application framework. Express, [s.d.]. Disponível em: https://expressjs.com/pt-br/. Acesso em: 30 jun. 2025.

FRASER, Steven; MANCL, Dennis. No silver bullet: software engineering reloaded. IEEE Software, v. 25, n. 1, p. 91–94, 2008. DOI: https://doi.org/10.1109/MS.2008.14.

FOWLER, Martin; LEWIS, James. Microservices - a definition of this new architectural term. 25 mar. 2014. Disponível em: https://martinfowler.com/articles/microservices.html. Acesso em 23 set. 2024.

GAN, Yu; ZHANG, Yanqi; HU, Kelvin; CHENG, Dailun; HE, Yuan; PANCHOLI, Meghna; DELIMITROU, Christina. Seer: Leveraging Big Data to Navigate the Complexity of Performance Debugging in Cloud Microservices. In: International Conference On Architectural Support For Programming Languages And Operating Systems, 24., 2019, Providence. Proceedings... New York: Association for Computing Machinery, 2019. p. 19-33. DOI: https://doi.org/10.1145/3297858.3304004.

GOOGLE CLOUD. Google Cloud. Google Cloud, [s.d.]. Disponível em: https://cloud.google.com. Acesso em: 30 jun. 2025.

GRPC. About gRPC. gRPC, [s.d.]. Disponível em: https://grpc.io/about/. Acesso em: 30 jun. 2025.

MENDONÇA, Nabor C.; BOX, Craig; MANOLACHE, Costin; RYAN, Louis. The Monolith Strikes Back: Why Istio Migrated From Microservices to a Monolithic Architecture. IEEE Software, v. 38, n. 5, p. 17-22, 2021. DOI: 10.1109/MS.2021.3080335.

MICROSOFT. TypeScript: JavaScript with syntax for types. TypeScript, [s.d.]. Disponível em: https://www.typescriptlang.org/. Acesso em: 30 jun. 2025.

ORACLE. MySQL. MySQL, [s.d.]. Disponível em: https://www.mysql.com/. Acesso em: 30 jun. 2025.

REDIS. Distributed caching. Redis Documentation. Disponível em: https://redis.io/glossary/distributed-caching/. Acesso em: 30 jun. 2025.

THÖNES, Johannes. Microservices. IEEE Software, v. 32, n. 1, p. 116, 2015. DOI: 10.1109/MS.2015.11.

ZHU, Jianyong; YANG, Renyu; SUN, Xiaoyang; WO, Tianyu; HU, Chunming; PENG, Hao; XIAO, Junqing; ZOMAYA, Albert Y.; XU, Jie. QoS-Aware Co-Scheduling for Distributed Long-Running Applications on Shared Clusters. IEEE Transactions on Parallel and Distributed Systems, v. 33, n. 12, p. 4818-4834, 2022.

Anexos

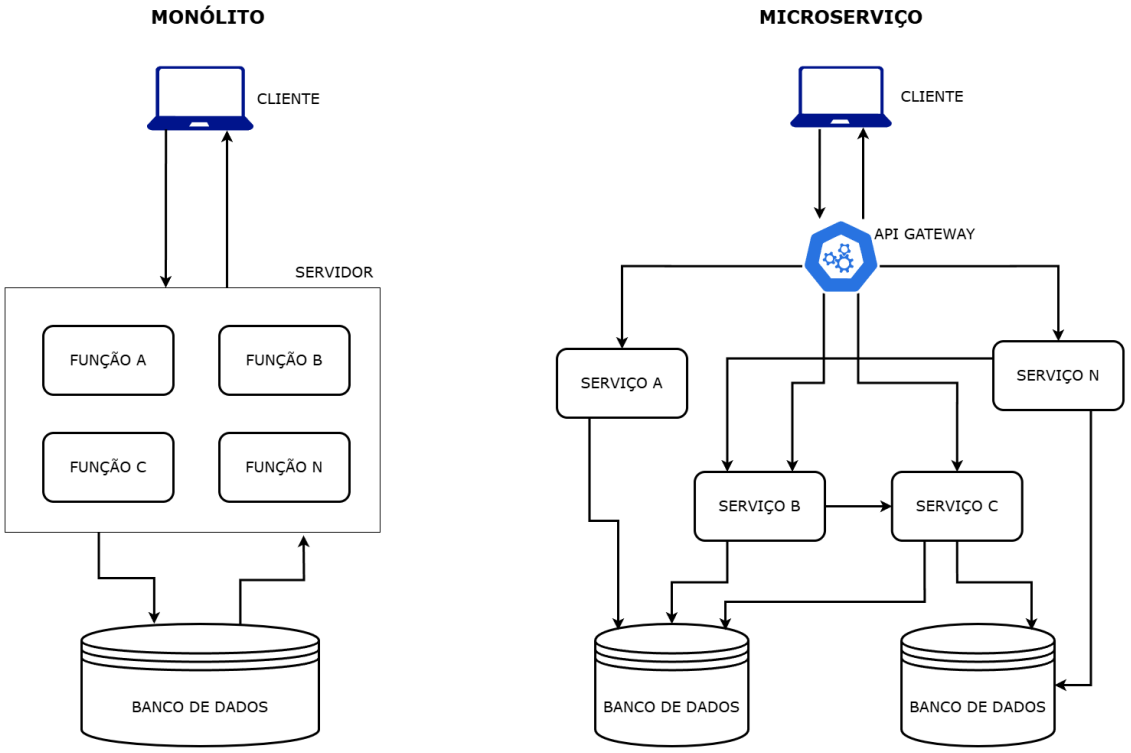

Figura 9 . Comparativo de arquiteturas

Fonte: Autoria própria

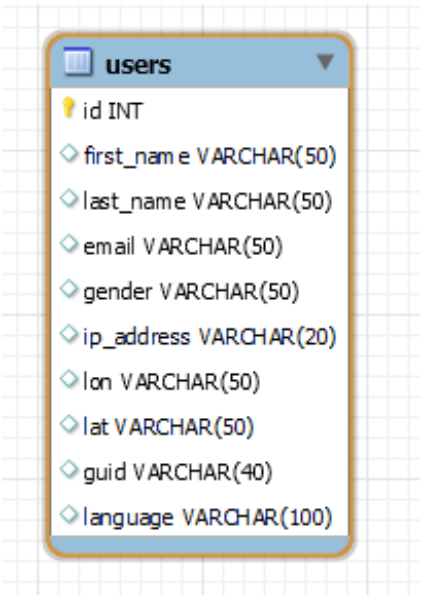

Figura 10. Modelo ER dos demais serviços

Fonte: Autoria própria

```
const getAll = async () => {
  try {
    const allUsersFromCache = (await app.cacheClient.get('allUsers')) as string;

    if (allUsersFromCache) {
      return JSON.parse(allUsersFromCache);
    } else {
      const response = await connection.query(`
        SELECT
          *
        FROM users
      `);

      await app.cacheClient.set('allUsers', JSON.stringify(response), {
        expiration: { type: 'EX', value: 10 }
      });

      return response;
    }
  } catch (err) {
    console.log(err);
    return err;
  }
};
```

Figura 11. Redis implementação

Fonte: Autoria própria

usersFour.proto

service-four > src > usersFour.proto > ...

```
syntax = "proto3";

package users;

message User {
    uint32 id = 1;
    string first_name = 2;
    string last_name = 3;
    string email = 4;
    string gender = 5;
    uint32 ip_address = 6;
    double lon = 7;
    double lat = 8;
    string guid = 9;
    string language = 10;
    uint32 best_friend_id = 11;
    uint32 countBestFriends = 12;
}

message UserAccessHistoric {
    uint32 id = 1;
    string first_name = 2;
    string last_name = 3;
    string email = 4;
    string gender = 5;
    uint32 ip_address = 6;
    double lon = 7;
    double lat = 8;
    string guid = 9;
    string language = 10;
    uint32 best_friend_id = 11;
    uint32 user_id = 12;
    string access_datetime = 13;
    string leave_datetime = 14;
}

message UsersList {
    repeated User user = 1;
    repeated UserAccessHistoric accessHistoric = 2;
}

message Void {}

service Users {
    rpc GetUsers(Void) returns (UsersList) {}
}
```

Figura 12. Implementação arquivo proto

Fonte: Autoria própria

Coluna 1	média (ms)	p50 (ms)	p90 (ms)	p99 (ms)	MR2xx
resultado 1	1790	1752	2448	3754	258
resultado 2	1400	1437	1891	2641	331
resultado 3	1696	1789	2227	2555	251
média	1629	1659	2189	2983	280

Figura 13. Modelo de alta dependência, teste base, sem tratamento

Fonte: Autoria própria

Coluna 1	média (ms)	p50 (ms)	p90 (ms)	p99 (ms)	# MR2xx
resultado 1	1734	1560	2680	4105	266
resultado 2	1453	1399	2015	2793	322
resultado 3	1573	1578	2129	2978	292
média	1587	1512	2275	3292	293

Figura 14. Modelo de alta dependência, teste base, com tratamento

Fonte: Autoria própria

Coluna 1	média (ms)	p50 (ms)	p90 (ms)	p99 (ms)	MR2xx
resultado 1	1568	1495	2366	6595	3028
resultado 2	1350	1365	2135	3049	3518
resultado 3	1241	1257	2035	2647	3809
média	1386	1372	2179	4097	3452

Figura 15. Modelo de alta dependência, teste de estresse sustentado, sem tratamento

Fonte: Autoria própria

Coluna 1	média (ms)	p50 (ms)	p90 (ms)	p99 (ms)	# MR2xx
resultado 1	1347	1257	2034	4534	3467
resultado 2	1137	1038	1877	2611	4171
resultado 3	1104	1031	1767	2472	4306
média	1196	1109	1893	3206	3981

Figura 16. Modelo de alta dependência, teste de estresse sustentado, com tratamento

Fonte: Autoria própria

Coluna 1	média (ms)	p50 (ms)	p90 (ms)	p99 (ms)	MR2xx
resultado 1	1127	1184	1676	2228	6000
resultado 2	3034	3324	5962	6770	6000
resultado 3	884	662	1664	2704	6000
média	1682	1723	3101	3901	6000

Figura 17. Modelo de alta dependência, teste por quantidade, sem tratamento

Fonte: Autoria própria

Coluna 1	média (ms)	p50 (ms)	p90 (ms)	p99 (ms)	# MR2xx
resultado 1	3772	507	20810	27450	5090
resultado 2	1929	967	4285	11029	5367
resultado 3	1613	954	1723	10951	5627
média	2438	809	8939	16477	5361

Figura 18. Modelo de alta dependência, teste por quantidade, com tratamento

Fonte: Autoria própria

Coluna 1	média (ms)	p50 (ms)	p90 (ms)	p99 (ms)	MR2xx
resultado 1	1894	1845	2515	3631	51
resultado 2	2266	2346	2995	3868	42
resultado 3	2272	2310	3417	3604	42
média	2144	2167	2976	3701	45

Figura 19. Modelo de resposta custosa, teste base, sem tratamento

Fonte: Autoria própria

Coluna 1	média (ms)	p50 (ms)	p90 (ms)	p99 (ms)	# MR2xx
resultado 1	1106	775	1735	3973	87
resultado 2	1174	819	2400	4115	84
resultado 3	1015	654	1772	4034	97
média	1098	749	1969	4041	89

Figura 20. Modelo de resposta custosa, teste base, sem tratamento

Fonte: Autoria própria

Coluna 1	média (ms)	p50 (ms)	p90 (ms)	p99 (ms)	# MR2xx
resultado 1	9004	8400	11832	24735	91
resultado 2	8867	8357	11542	25002	90
resultado 3	9237	8296	11180	29863	90
média	9036	8351	11518	26533	90

Figura 21. Modelo de resposta custosa, teste de estresse sustentado, sem tratamento

Fonte: Autoria própria

Coluna 1	média (ms)	p50 (ms)	p90 (ms)	p99 (ms)	# MR2xx
resultado 1	5127	3937	10825	13181	280
resultado 2	4798	3608	10610	12872	298
resultado 3	4796	3604	10694	12807	302
média	4907	3716	10710	12953	293

Figura 22. Modelo de resposta custosa, teste de estresse sustentado, com tratamento

Fonte: Autoria própria

Coluna 1	média (ms)	p50 (ms)	p90 (ms)	p99 (ms)	MR2xx
resultado 1	17183	4652	40941	48610	150
resultado 2	16846	4776	38807	46075	150
resultado 3	17352	4913	40729	50763	150
média	17127	4780	40159	48483	150

Figura 23. Modelo de resposta custosa, teste por quantidade, sem tratamento

Fonte: Autoria própria

Coluna 1	média (ms)	p50 (ms)	p90 (ms)	p99 (ms)	MR2xx
resultado 1	1922	1436	4378	6498	150
resultado 2	1844	1291	4883	6575	150
resultado 3	1710	1207	4737	6865	150
média	1825	1311	4666	6646	150

Figura 24. Modelo de resposta custosa, teste por quantidade, com tratamento

Fonte: Autoria própria