



UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO TECNOLÓGICO
SISTEMAS DA INFORMAÇÃO

Gabriel Rodrigues dos Santos Mouta

**Desafios e Estratégias na Implementação da Comunicação entre Microserviços em
Arquiteturas Escaláveis: Um Estudo de Caso na Keeps**

Florianópolis
2025

Gabriel Rodrigues dos Santos Mouta

**Desafios e Estratégias na Implementação da Comunicação entre Microserviços em
Arquiteturas Escaláveis: Um Estudo de Caso na Keeps**

Trabalho de Conclusão de Curso submetido ao curso de Sistemas da Informação do Centro ou Campus Centro Tecnológico da Universidade Federal de Santa Catarina como requisito parcial para a obtenção do título de Bacharel em Sistemas da Informação.

Orientadora: Profa. Patricia Della Méa Plentz

Florianópolis

2025

Ficha catalográfica gerada por meio de sistema automatizado gerenciado pela BU/UFSC. Dados inseridos pelo próprio autor.

Mouta, Gabriel Rodrigues dos Santos

Desafios e Estratégias na Implementação da Comunicação
entre Microsserviços em Arquiteturas Escaláveis : Um
Estudo de Caso na Keeps / Gabriel Rodrigues dos Santos
Mouta ; orientadora, Patricia Della Múa Plentz, 2025.
94 p.

Trabalho de Conclusão de Curso (graduação) -
Universidade Federal de Santa Catarina, Centro
Tecnológico, Graduação em Sistemas de Informação,
Florianópolis, 2025.

Inclui referências.

1. Sistemas de Informação. 2. Microsserviços. 3.
Arquitetura de Software. 4. Comunicação Distribuída. 5.
Escalabilidade. I. Plentz, Patricia Della Múa. II.
Universidade Federal de Santa Catarina. Graduação em
Sistemas de Informação. III. Título.

Gabriel Rodrigues dos Santos Mouta

**Desafios e Estratégias na Implementação da Comunicação entre Microsserviços em
Arquiteturas Escaláveis: Um Estudo de Caso na Keeps**

Este Trabalho de Conclusão de Curso foi julgado adequado para obtenção do título de Bacharel e aprovado em sua forma final pelo Curso Sistemas da Informação.

Local Florianópolis, 10 de Dezembro de 2025.



Coordenação do Curso

Banca examinadora



Profa. Patricia Della Méa Plentz, Dra.
Orientadora



Prof. Frank Augusto Siqueira, Dr.
Instituição Universidade Federal de Santa Catarina



Prof. Odorico Machado Mendizabal, Dr.
Instituição Universidade Federal de Santa Catarina

Florianópolis, 2025

RESUMO

Este Trabalho de Conclusão de Curso (TCC) tem como objetivo investigar os desafios e as estratégias para uma comunicação eficiente entre microsserviços em arquiteturas escaláveis, com foco no estudo de caso da empresa Keeps. O trabalho analisa as dificuldades enfrentadas durante a migração de uma arquitetura SOA para microsserviços, abordando latência, consumo de recursos, falhas de integração e demais desafios. Além disso, foram avaliados protocolos e ferramentas como REST, gRPC, RabbitMQ, API Gateway e Kubernetes, com base em literatura especializada e experiências práticas da empresa. A pesquisa busca compreender os fatores que impactam o desempenho e a escalabilidade da comunicação entre microsserviços, apresentando recomendações e estratégias que podem auxiliar outras empresas em processos semelhantes de modernização arquitetural.

Palavras-chave: Microsserviços, Arquitetura de Software, Comunicação Distribuída, Kubernetes, Escalabilidade.

ABSTRACT

This undergraduate final project aims to investigate the challenges and strategies for achieving efficient communication between microservices in scalable architectures, focusing on a case study of the company Keeps. The study analyzes the difficulties encountered during the migration from a Service-Oriented Architecture (SOA) to a microservices-based architecture, addressing issues such as latency, resource consumption, and integration failures. Additionally, the research evaluates communication protocols and tools such as *REST*, *gRPC*, *RabbitMQ*, *API Gateway*, and Kubernetes, supported by academic literature and practical experiences from the company. The project seeks to understand the key factors that impact the performance and scalability of microservices communication, offering recommendations and strategies that may assist other companies undergoing similar modernization processes.

Keywords: Microservices; Software Architecture; Distributed Communication; Kubernetes; Scalability.

SUMÁRIO

1 INTRODUÇÃO	9
1.1 OBJETIVOS	10
1.2 CONTRIBUIÇÕES E ORGANIZAÇÃO DO TRABALHO	11
2 FUNDAMENTAÇÃO TEÓRICA	12
2.1 CONCEITOS E FUNDAMENTOS DA ARQUITETURA DE MICROSERVIÇOS	12
2.2 COMUNICAÇÃO ENTRE MICROSERVIÇOS	13
2.2.1 Padrões de comunicação	13
2.2.1.1 Comunicação Síncrona	13
2.2.1.2 Comunicação assíncrona	14
2.2.1.3 Comunicação híbrida	14
2.2.2 Protocolos e padrões de comunicação	15
2.3 FERRAMENTAS DE SUPORTE	18
2.3.1 RabbitMQ	19
2.3.2 Apache Kafka	20
2.3.3 Kubernetes	20
2.3.4 API Gateway	22
2.3.5 Ferramentas de observabilidade	23
2.3.6 OpenTelemetry Operator	24
2.3.7 Redis	25
2.4 DESAFIOS DA COMUNICAÇÃO ENTRE MICROSERVIÇOS	26
2.4.1 Latência acumulada e Performance	27
2.4.2 Acoplamento temporal	28
2.4.3 Tolerância a falhas	30
2.4.4 Orquestração e escalonamento	31
2.4.5 Consistência e gerenciamento de dados	33
2.4.6 Observabilidade e monitoramento	35
2.4.7 Segurança da Comunicação	37
3 TRABALHOS RELACIONADOS	40
3.1 PESQUISA BIBLIOGRÁFICA	40
3.2 ESTADO DA ARTE	42
4 ESTUDO DE CASO: ARQUITETURA KEEPS	45
4.1 SOBRE A KEEPS	45
4.2 ARQUITETURA DA KEEPS	47
4.2.1 CQRS e Replicação de Dados	47
4.2.2 Comunicação Entre Microserviços API e Workers	49
4.2.3 Kubernetes e Arquitetura Geral de Comunicação	50
4.3 RELATÓRIO DE PROBLEMAS	52
4.3.1 Observabilidade limitada	52
4.3.2 Alta dependência entre aplicações	53
4.3.3 Perda de mensagens e baixa confiabilidade	54
4.3.4 Gargalos de desempenho	55

4.3.5 Falhas não detectadas em serviços ativos	55
4.3.6 Falta de governança e padronização	56
4.3.7 Síntese	56
5 MELHORIAS E RESULTADOS	58
5.1 MELHORIAS IMPLEMENTADAS NA KEEPS	58
5.1.1 Melhorias na observabilidade	59
5.1.2 Padronização da comunicação interna com gRPC	65
5.1.3 Utilização de cache	67
5.1.4 Adoção de consistência eventual	68
5.1.5 Reestruturação e ampliação do uso de mensageria	71
5.1.6 Implementação do API Gateway	72
5.2 RESULTADOS EXPERIMENTAIS E MÉTRICAS DE DESEMPENHO	73
5.2.1 Metodologia dos testes	74
5.2.2 Cenário experimental: Tracking no Konquest	75
5.2.3 Execução e coleta de métricas	75
5.2.4 Resultados obtidos	77
5.2.5 Discussão dos resultados	80
6 CONCLUSÃO	82
6.1 SÍNTESE DOS RESULTADOS	82
6.2 LIMITAÇÕES DO ESTUDO	82
6.3 TRABALHOS FUTUROS	83
6.4 CONSIDERAÇÕES FINAIS	84

1 INTRODUÇÃO

A arquitetura de microsserviços tem se consolidado como uma abordagem eficaz para o desenvolvimento de sistemas escaláveis, flexíveis e altamente disponíveis (FOWLER, 2017; AKSAKALLI et al., 2021). Ao promover a decomposição de aplicações em serviços independentes e especializados, esse modelo arquitetural supera diversas limitações das arquiteturas monolíticas e orientadas a serviços, *Service-Oriented Architecture* (SOA), possibilitando maior autonomia de desenvolvimento, melhor implantação contínua e resiliência operacional (AKSAKALLI et al., 2021). Contudo, a adoção dessa arquitetura traz novos desafios, sendo a comunicação entre microsserviços (distribuída, heterogênea e sujeita à latência) um dos aspectos mais críticos a serem otimizados, especialmente em termos de eficiência, resiliência e consumo de recursos, como também discutido por Aksakalli et al. (2021).

A Keeps é uma EdTech que oferece soluções voltadas à gestão e à experiência de aprendizado corporativo, atuando com sistemas *Learning Management System* (LMS) e *Learning Experience Platform* (LXP). Suas aplicações, distribuídas no modelo *Software as a Service* (SaaS), são utilizadas por empresas que buscam aprimorar a capacitação e o desenvolvimento de pessoas em seus setores de Treinamento e Desenvolvimento (T&D). O principal indicador de crescimento da Keeps está diretamente relacionado à quantidade de usuários ativos em suas plataformas, o que exige um ecossistema tecnológico altamente disponível, escalável e eficiente.

Inicialmente, a empresa operava com uma arquitetura SOA hospedada em ambiente de nuvem baseado em soluções *Infrastructure as a Service* (IaaS) da AWS. Essa abordagem sustentou as operações entre 2015 e 2020, até que, em 2021, a organização iniciou o processo de migração para uma arquitetura de microsserviços, com o objetivo de aumentar a flexibilidade, reduzir o acoplamento entre módulos e melhorar a escalabilidade dos sistemas.

Entretanto, após essa transição, surgiram desafios significativos relacionados à comunicação entre os microsserviços, como alta latência, falhas de integração, dificuldades de monitoramento e uso ineficiente de recursos computacionais. Esses fatores afetam diretamente a performance dos sistemas e, consequentemente, a qualidade dos serviços entregues aos clientes.

A literatura reforça a importância de decisões criteriosas sobre protocolos e padrões de comunicação, pois essas escolhas impactam diretamente a eficiência e a resiliência de sistemas distribuídos. Estudos como os de Aksakalli et al. (2021) identificam que a definição dos padrões de comunicação é um dos principais desafios em arquiteturas de microsserviços, enquanto Niswar et al. (2024) demonstram experimentalmente que a escolha do protocolo — como *REST* (*Representational State Transfer*) ou *gRPC* (*Google Remote Procedure Call*) — influencia o tempo de resposta e o consumo de CPU de forma significativa. Assim, compreender as implicações dessas escolhas é essencial para otimizar a comunicação entre serviços e aprimorar o desempenho geral da arquitetura.

Nesse contexto, a relevância deste estudo torna-se evidente, uma vez que a Keeps enfrenta um cenário representativo dos desafios contemporâneos em arquiteturas distribuídas. A questão central que orientou esta pesquisa é: Como a comunicação entre microsserviços pode ser otimizada para garantir eficiência, resiliência e escalabilidade no ambiente da Keeps?

A motivação para este trabalho surgiu da necessidade prática enfrentada pela Keeps em responder a questão exposta anteriormente. A empresa vivenciou, de forma concreta, problemas como alta latência, falhas de integração e limitações de observabilidade. Ao investigar essas questões no contexto real da Keeps, este estudo buscou gerar conhecimento aplicável não apenas à organização analisada, mas também a outras empresas que enfrentam processos semelhantes de modernização arquitetural.

A escolha deste tema se justifica por sua relevância prática e atual. Embora a arquitetura de microsserviços ofereça escalabilidade e flexibilidade, sua adoção apresenta desafios significativos de comunicação entre serviços distribuídos (AKSAKALLI et al., 2021; NISWAR et al., 2024). Na Keeps, esses desafios impactavam o desempenho, a estabilidade e a qualidade dos serviços. Este trabalho buscou analisar e aplicar soluções consolidadas na literatura para avaliar sua efetividade no contexto real da empresa.

1.1 OBJETIVOS

O objetivo geral deste trabalho é analisar e propor formas de otimizar a comunicação entre microsserviços, visando aprimorar a eficiência, a resiliência e a

escalabilidade no ambiente da empresa Keeps. Para tal, foram definidos os seguintes objetivos específicos:

- a) Analisar os principais desafios enfrentados pela Keeps durante a comunicação entre microsserviços;
- b) Avaliar, por meio de observação e análise prática, as soluções adotadas pela empresa, como *API Gateway*, RabbitMQ, gRPC e Kubernetes;
- c) Comparar as soluções aplicadas pela Keeps com o que é recomendado na literatura especializada;
- d) Propor estratégias e recomendações para otimizar a comunicação entre microsserviços, com foco em eficiência, resiliência e escalabilidade.

1.2 CONTRIBUIÇÕES E ORGANIZAÇÃO DO TRABALHO

Este trabalho fornece contribuições práticas e aplicáveis para organizações que enfrentam desafios de comunicação em ambientes distribuídos. O estudo apresenta um mapeamento dos desafios recorrentes na comunicação entre microsserviços e um conjunto de soluções para a superação desses desafios baseadas na literatura especializada e validadas na prática através do estudo de caso na Keeps, visando fornecer um exemplo concreto para outras empresas em processos de migração ou otimização de arquiteturas escaláveis.

O trabalho está estruturado da seguinte forma: o Capítulo 1 – Introdução apresenta o tema, a questão de pesquisa, a motivação, a justificativa, os objetivos e as contribuições pretendidas. O Capítulo 2 – Fundamentação Teórica aborda os conceitos de microsserviços, comunicação distribuída, padrões de comunicação, ferramentas e protocolos utilizados. O Capítulo 3 – Trabalhos Relacionados apresenta um levantamento de estudos anteriores sobre o tema, destacando abordagens e resultados relevantes. O Capítulo 4 – Arquitetura Atual da Keeps descreve a estrutura da empresa, os desafios enfrentados e as soluções adotadas, contextualizando o estudo de caso. O Capítulo 5 – Melhorias e Resultados detalha as melhorias implementadas na plataforma, os testes realizados e a análise de resultados obtidos. Por fim, o Capítulo 6 – Conclusão apresenta as conclusões finais do trabalho, as limitações encontradas e as etapas previstas para a continuidade da pesquisa.

2 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo é apresentada a fundamentação teórica necessária para compreender os conceitos e as práticas envolvidas na arquitetura de microsserviços, com ênfase nos desafios e soluções relacionadas à comunicação entre microsserviços em sistemas distribuídos. Sendo abordados os principais padrões de comunicação, como síncrona e assíncrona, além de ferramentas utilizadas, como REST, GraphQL e gRPC. O capítulo também discute as ferramentas de suporte essenciais, como Kubernetes, RabbitMQ, e API Gateway, e como essas tecnologias contribuem para a escalabilidade, resiliência e eficiência das arquiteturas de microsserviços.

2.1 CONCEITOS E FUNDAMENTOS DA ARQUITETURA DE MICROSERVIÇOS

A arquitetura de microsserviços tem se consolidado como uma abordagem moderna e eficaz para o desenvolvimento de sistemas escaláveis, modulares e resilientes, sendo amplamente adotada por sua capacidade de promover flexibilidade, autonomia e alta disponibilidade. De acordo com Fowler (2017), essa arquitetura organiza uma aplicação como um conjunto de pequenos serviços independentes, cada um responsável por uma função específica e capaz de se comunicar com os demais por meio de interfaces.

A autora destaca que o principal objetivo desse modelo é promover a autonomia de desenvolvimento, implantação e operação de cada serviço, permitindo que equipes distintas atuem de forma simultânea e independente. Essa separação favorece o isolamento de falhas, o escalonamento individual de componentes e a manutenção contínua, reduzindo o impacto de interrupções e acelerando o ciclo de entrega de software. Fowler (2017) também enfatiza que os microsserviços devem ser pequenos o suficiente para desempenhar uma única função de negócio, assegurando maior flexibilidade e resiliência em sistemas distribuídos.

Além disso, os microsserviços são comumente implementados em plataformas de contêineres, como o Docker, e gerenciados por ferramentas de orquestração, como o Kubernetes, que automatizam a implantação, o escalonamento e a recuperação de serviços. Essas tecnologias contribuem significativamente para garantir alta disponibilidade, balanceamento de carga

eficiente e resiliência operacional em ambientes distribuídos (AKSAKALLI et al., 2021).

2.2 COMUNICAÇÃO ENTRE MICROSERVIÇOS

A comunicação é um dos elementos mais críticos em arquiteturas de microserviços. Diferentemente das aplicações monolíticas, em que os componentes compartilham memória ou se comunicam dentro de um mesmo processo, os microserviços trocam informações por meio de mensagens entre serviços logicamente independentes. Essa característica aumenta a complexidade do sistema e exige estratégias robustas para garantir desempenho, confiabilidade e escalabilidade (AKSAKALLI et al., 2021).

2.2.1 Padrões de comunicação

Existem diferentes padrões de comunicação que podem ser adotados conforme os requisitos do sistema e as características dos serviços envolvidos. A literatura classifica os padrões mais recorrentes em três categorias: comunicação síncrona, comunicação assíncrona e comunicação híbrida. Cada uma dessas abordagens apresenta vantagens e limitações relacionadas à latência, ao acoplamento temporal e à tolerância a falhas (AKSAKALLI et al., 2021).

2.2.1.1 Comunicação Síncrona

A comunicação síncrona é caracterizada pela dependência temporal entre os serviços envolvidos: o serviço solicitante envia uma requisição e aguarda pela resposta antes de continuar seu processamento. Esse tipo de comunicação é frequentemente utilizado em interações diretas entre microserviços, especialmente quando há necessidade de resposta imediata ou forte dependência de dados entre os serviços (AKSAKALLI et al., 2021).

De acordo com Aksakalli et al. (2021), a comunicação síncrona segue o modelo *request-response*, no qual a execução de um serviço depende da resposta do outro para continuar. Esse padrão pode ser implementado por diferentes tecnologias, como a comunicação no padrão REST utilizando *HTTP (Hypertext Transfer Protocol)* ou gRPC.

A principal limitação da comunicação síncrona está no acoplamento temporal: qualquer indisponibilidade ou lentidão no serviço chamado impacta diretamente o serviço cliente, o que pode gerar efeitos em cascata em arquiteturas altamente distribuídas. Além disso, esse modelo aumenta a latência geral do sistema, especialmente quando há cadeias longas de chamadas entre múltiplos serviços. Assim, em ambientes que exigem alta disponibilidade e resiliência, é comum a adoção de mecanismos adicionais, como *circuit breakers*, *timeouts* e *fallbacks*, para mitigar o impacto de falhas e prevenir sobrecargas (AKSAKALLI et al., 2021).

2.2.1.2 Comunicação assíncrona

A comunicação assíncrona é caracterizada pela independência temporal entre os serviços, permitindo que as mensagens sejam enviadas sem a necessidade de aguardar a resposta imediata. Em uma das formas de implementação desse modelo, um serviço funciona como produtor, enviando mensagem a um intermediário — como uma fila, um tópico ou um *event bus* — e segue o processamento normalmente. O outro serviço funciona como consumidor, por sua vez, processa a mensagem de forma autônoma, no seu próprio tempo.

Segundo Aksakalli et al. (2021), esse padrão é amplamente utilizado para aumentar a resiliência e o desacoplamento entre serviços, pois elimina a dependência direta entre o tempo de execução das operações. Na implementação descrita no parágrafo anterior, a comunicação costuma envolver *message brokers* e sistemas de eventos ou de *stream*.

Entre as principais vantagens da comunicação assíncrona estão a facilitação da escalabilidade horizontal, maior tolerância a falhas e a redução do acoplamento temporal. Por outro lado, ela impõe desafios adicionais relacionados à consistência de dados e ao monitoramento, uma vez que as mensagens podem ser processadas fora de ordem ou com atrasos. Estratégias como *dead-letter queue* (DLQ), *retry policies* e *event sourcing* são comumente adotadas para garantir confiabilidade e rastreabilidade das mensagens em sistemas assíncronos.

2.2.1.3 Comunicação híbrida

Em sistemas complexos, é comum a adoção de um modelo híbrido de comunicação, que combina elementos das abordagens síncrona e assíncrona. Essa estratégia permite equilibrar a necessidade de respostas imediatas com o processamento assíncrono de tarefas mais demoradas ou menos críticas. Por exemplo, uma *API (Application Programming Interface)* pode responder ao cliente de forma síncrona confirmando o recebimento de uma solicitação, enquanto o processamento detalhado dessa solicitação é delegado a um *worker* que a consome de uma fila de mensagens.

A comunicação híbrida oferece maior flexibilidade e resiliência operacional, mas exige um controle rigoroso sobre consistência e monitoramento, pois combina fluxos distintos de execução. O uso de padrões de projeto como *saga orchestration* e *event-driven workflows* é recomendado para coordenar as dependências entre transações distribuídas (AKSAKALLI et al., 2021).

2.2.2 Protocolos e padrões de comunicação

A escolha dos protocolos e dos padrões de comunicação afeta diretamente o desempenho, a latência e o consumo de recursos do sistema. É importante distinguir que protocolos definem regras de transporte e serialização dos dados, utilizados pelos padrões e ferramentas que determinam como as interfaces serão estruturadas e acessadas. Em arquiteturas de microsserviços, essa escolha influencia não apenas a eficiência, mas também a interoperabilidade e a facilidade de manutenção (LERCHER et al., 2024).

O REST é um estilo arquitetural que define princípios para construção de APIs orientadas a recursos, baseadas em operações HTTP (GET, POST, PUT, DELETE) e geralmente utilizando *JSON (JavaScript Object Notation)*, como formato de dados. Sua popularidade decorre da simplicidade, compatibilidade ampla e fácil integração com diferentes linguagens e plataformas. No entanto, por operar sobre HTTP/1.1 e lidar com transferências textuais, o REST pode apresentar sobrecarga (*overfetching*) e latência elevada em cenários de alta demanda (NISWAR et al., 2024).

O gRPC, por sua vez, é um *framework* de comunicação de alto desempenho desenvolvido pelo Google. Ele utiliza o HTTP/2 como protocolo de transporte e o *Protocol Buffers* para serialização binária, o que reduz o tamanho das mensagens e acelera a comunicação entre serviços. Além disso, o gRPC oferece suporte nativo a

streaming bidirecional e contratos fortemente tipados, tornando-o ideal para sistemas de baixa latência e alto volume de chamadas simultâneas (NISWAR et al., 2024).

Já o GraphQL é uma linguagem de consulta e especificação de API desenvolvida pelo Facebook (atual Meta), cujo principal objetivo é oferecer maior flexibilidade na obtenção de dados. Diferentemente do REST, que segue um estilo arquitetural orientado a recursos, o GraphQL permite que o cliente defina exatamente quais campos deseja receber, evitando problemas de *overfetching* e *underfetching* comuns em APIs REST.

Essa flexibilidade, contudo, pode introduzir custos adicionais dependendo do tipo de operação executada. Segundo Niswar et al. (2024), em alguns cenários de consultas complexas com dados aninhados (*nested data*), o GraphQL apresentou maior consumo de CPU e maior tempo de resposta quando comparado a REST e gRPC. Esse comportamento deve-se à necessidade de percorrer múltiplas camadas de relacionamento para compor a resposta final, o que eleva o custo computacional do servidor.

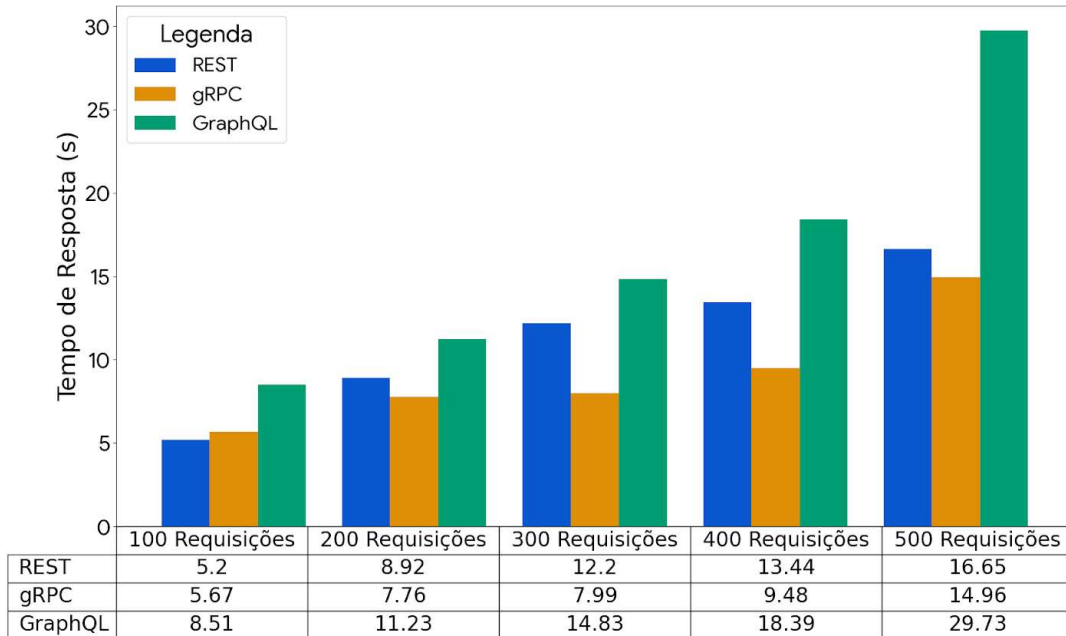
É importante destacar, entretanto, que esses resultados não representam o desempenho do GraphQL em todas as situações. Em consultas simples ou quando o cliente precisa apenas de subconjuntos específicos de informações, o GraphQL pode ser mais eficiente que o REST, pois reduz a quantidade de dados trafegados e elimina requisições redundantes (KHAN et al., 2023; AKSAKALLI et al., 2021).

Nos testes de Niswar et al. (2024), foram implementados três microsserviços em contêineres, com cada serviço utilizando Redis e MySQL para armazenar dados. Os testes foram realizados com dois tipos de dados: dados planos (*flat data*) e *nested data*. O desempenho foi avaliado em termos de tempo de resposta e utilização de CPU durante diferentes quantidades de requisições (100, 200, 300, 400 e 500 requisições). O Apache JMeter foi utilizado para simular a carga de requisições e medir o impacto nos dois parâmetros de desempenho.

Nas Figuras 1 e 2, adaptadas do artigo, observamos as comparações entre os três protocolos em relação ao tempo de resposta e ao uso de CPU. A Figura 1 mostra que, conforme o número de requisições aumenta, o gRPC apresenta um tempo de resposta significativamente menor do que o REST e GraphQL, que aumentam consideravelmente à medida que a carga de requisições cresce. Já a Figura 2 exibe a utilização de CPU, onde o gRPC se mostra mais eficiente, com o

GraphQL apresentando um uso muito mais alto de recursos, especialmente em cenários de alta carga (500 requisições).

Figura 1 – Tempo de resposta para recuperar dados



Fonte: Niswar et al. 2024 (adaptado)

Figura 2 – Uso de CPU para recuperar dados



Fonte: Niswar et al. 2024

A Figura 2 exibe a utilização de CPU. Nota-se que o *GraphQL* apresentou um consumo de recursos significativamente maior, ultrapassando 100% em vários cenários. Isso ocorre porque a métrica contabiliza a soma do uso dos núcleos

disponíveis, onde 100% equivale a um core completo, evidenciando que o GraphQL exigiu processamento paralelo intenso e o uso de múltiplos núcleos para lidar com a complexidade da interpretação das consultas, enquanto gRPC e REST se mantiveram mais eficientes, operando majoritariamente dentro da capacidade de um único núcleo.

Por fim, o Quadro 1 resume de forma clara as principais diferenças entre os protocolos citados, suas vantagens e desvantagens.

Quadro 1 – Protocolos de comunicação

Abordagem	Vantagens	Desvantagens
REST	Simplicidade, compatibilidade ampla, fácil adoção	Latência e <i>overfetching</i>
GraphQL	Flexibilidade e consultas seletivas	Alto custo em dados aninhados
gRPC	Alta performance e baixo overhead	Complexidade de implementação

Fonte: elaborado pelo autor.

A literatura analisada evidencia que não há uma solução universal para comunicação entre microsserviços. A escolha entre REST, GraphQL e gRPC — e demais soluções — deve considerar o contexto operacional, o volume de dados e o perfil das requisições. REST continua sendo amplamente utilizado por sua simplicidade e compatibilidade; o gRPC destaca-se em cenários de alta performance e comunicação interna entre serviços; e o GraphQL é mais vantajoso quando há necessidade de consultas flexíveis e controle sobre o volume de dados retornados (AKSAKALLI et al., 2021; NISWAR et al., 2024). A decisão adequada depende do equilíbrio entre eficiência, manutenção e escalabilidade dentro da arquitetura adotada.

2.3 FERRAMENTAS DE SUPORTE

A arquitetura de microsserviços exige o uso de ferramentas especializadas que assegurem uma comunicação eficiente, a orquestração adequada dos serviços e o monitoramento contínuo do sistema. O emprego correto dessas ferramentas é fundamental para garantir escalabilidade, resiliência e visibilidade operacional em ambientes distribuídos (KHAN et al., 2023).

Dentre as principais tecnologias que sustentam esse ecossistema, destacam-se os sistemas de mensageria, como o RabbitMQ, voltado à troca assíncrona de mensagens entre microsserviços, e o Apache Kafka, uma plataforma distribuída de *streaming* de eventos, projetada para o processamento contínuo de fluxos de dados e o registro imutável de mensagens (AKSAKALLI et al., 2021; RODRIGUES; RITO SILVA; AVRITZER, 2025).

Também se incluem as plataformas de orquestração, como o Kubernetes, responsáveis pela automação do ciclo de vida e escalonamento dos serviços e os mecanismos de gerenciamento de APIs, que centralizam o controle de acesso e versionamento (RODRIGUES; RITO SILVA; AVRITZER, 2025).

2.3.1 RabbitMQ

O RabbitMQ é um message broker amplamente utilizado para viabilizar a comunicação assíncrona entre microsserviços. Ele implementa o protocolo AMQP (*Advanced Message Queuing Protocol*), que define padrões para envio, roteamento e consumo de mensagens em sistemas distribuídos. De acordo com a documentação oficial (RABBITMQ, 2023), o RabbitMQ oferece suporte a múltiplos padrões de troca de mensagens, como *publish/subscribe*, *point-to-point* e *request/reply*. Em resumo, os produtores enviam mensagens para filas, que são então consumidas pelos consumidores de forma ordenada e configurável.

O RabbitMQ possui uma simplicidade de configuração, confiabilidade operacional e entrega garantida de mensagens, com foco na integridade das comunicações. Ele é adotado em sistemas de processamento assíncrono, integrações entre serviços e operações de *backend* que exigem consistência e controle de fluxo (AKSAKALLI et al., 2021; RODRIGUES; RITO SILVA; AVRITZER, 2025).

Segundo Aksakalli et al. (2021), o RabbitMQ é uma das soluções mais consolidadas para comunicação assíncrona em microsserviços, oferecendo suporte a filas persistentes, confirmação de entrega (*acknowledgment* ou *ACK*) e redirecionamento de mensagens falhas por meio de DLQs. Esses recursos contribuem para a resiliência e recuperação de falhas em sistemas distribuídos.

Em síntese, o RabbitMQ é ideal para arquiteturas que demandam confiabilidade, simplicidade e controle transacional em fluxos assíncronos de comunicação entre microsserviços.

2.3.2 Apache Kafka

O Apache Kafka, conforme Confluent (2023), é uma plataforma distribuída de processamento de streams projetada para o tráfego contínuo e em larga escala de eventos. Diferente de um *message broker* tradicional, o Kafka é estruturado com base no conceito de *logs* imutáveis e particionados, onde as mensagens são gravadas de forma sequencial e persistente. Ainda segundo a Confluent (2023), esse modelo possibilita o reprocessamento de eventos, o armazenamento histórico de mensagens e a distribuição de carga entre múltiplos consumidores, o que traz uma característica maior de escalabilidade. Ou seja, pode ser adequado para sistemas que demandam alta vazão (*throughput*) e baixa latência.

Segundo Aksakalli et al. (2021), o Kafka é amplamente utilizado em ambientes corporativos de grande porte devido à sua combinação de persistência, tolerância a falhas e escalabilidade horizontal. O Kafka permite que múltiplos consumidores leiam simultaneamente a mesma sequência de mensagens, proporcionando maior flexibilidade e paralelismo no processamento. Além disso, o Kafka facilita a construção de *pipelines* de dados reativos, em que os eventos são processados e transformados em tempo real, integrando-se a sistemas de monitoramento e análise contínua. Essa característica distingue o Kafka de outras soluções de mensageria, posicionando-o como uma ferramenta estratégica em ecossistemas orientados a eventos.

Em resumo, o Apache Kafka se destaca como uma solução de alta performance e escalabilidade, sendo ideal para o processamento massivo de dados e eventos. Ele pode complementar ferramentas como o RabbitMQ em arquiteturas distribuídas.

2.3.3 Kubernetes

O Kubernetes desempenha um papel central nas arquiteturas de microsserviços, oferecendo uma plataforma robusta para automação da

orquestração de contêineres, com foco na escalabilidade e resiliência operacional. Desenvolvido inicialmente pela Google e atualmente mantido pela Cloud Native Computing Foundation (CNCF), o Kubernetes possibilita a gestão eficiente de ambientes distribuídos, utilizando contêineres como unidade de deploy para microsserviços. A plataforma foi projetada para garantir a alta disponibilidade, o balanceamento de carga, a recuperação automática e o escalonamento dinâmico de serviços, atendendo as necessidades de sistemas que demandam elasticidade e adaptação rápida às variações de carga (AKSAKALLI et al., 2021).

Entre suas principais funcionalidades, destacam-se a implantação automatizada de microsserviços, realizada de forma declarativa por meio de arquivos *YAML*, e o escalonamento horizontal com base em métricas como consumo de CPU e memória. O Kubernetes oferece um balanceamento de carga eficiente, garantindo que o tráfego seja distribuído adequadamente entre as instâncias de serviços. Como destaca Rodrigues et al. (2025), o Kubernetes também facilita a recuperação automática de serviços falhos, minimizando o impacto de instabilidades, um fator essencial para garantir a continuidade operacional em sistemas complexos.

A arquitetura de microsserviços, como descrito por Aksakalli et al. (2021), se beneficia da utilização do Kubernetes para garantir a escalabilidade do sistema, especialmente quando se lida com altos volumes de uso e serviços interdependentes. O *Horizontal Pod Autoscaler (HPA)* ajusta automaticamente o número de instâncias de serviços com base nas métricas de uso, permitindo que o ambiente se ajuste conforme a carga e mantendo a alta disponibilidade mesmo durante picos de tráfego. Além disso, a plataforma permite a utilização de *Ingress Controllers*, que gerenciam o tráfego *HTTP/HTTPS*, garantindo que as requisições sejam corretamente roteadas e equilibradas entre as instâncias do sistema.

Entretanto, como apontado por Aksakalli et al. (2021) e Rodrigues et al. (2025), a adoção de Kubernetes em arquiteturas distribuídas não está isenta de desafios. A escalabilidade e a resiliência operacionais podem ser comprometidas se as políticas de escalonamento e balanceamento de carga não forem configuradas adequadamente, o que pode levar a falhas em cascata, comprometendo a comunicação entre os microsserviços. Ou seja, ajustes contínuos nas configurações e monitoramento das métricas de desempenho são necessários para garantir a eficiência e a resiliência a longo prazo.

Essas práticas de orquestração e automação exemplificam como o Kubernetes é fundamental para manter a eficiência, resiliência e escalabilidade de plataformas, e como ele contribui para a evolução das arquiteturas de microsserviços de forma a garantir a operação contínua e otimizada, mesmo diante de alto *throughput* (KHAN et al., 2023; AKSAKALLI et al., 2021).

2.3.4 API Gateway

O API Gateway é um componente essencial em arquiteturas de microsserviços, responsável por centralizar e gerenciar o tráfego de entrada no sistema. Ele atua como um ponto único de acesso entre os clientes e os serviços internos, roteando as requisições para os microsserviços apropriados. Essa camada de intermediação simplifica a comunicação e reduz o acoplamento entre consumidores e produtores de serviços, promovendo uma arquitetura mais flexível e escalável (FOWLER, 2017; AKSAKALLI et al., 2021).

De acordo com Aksakalli et al. (2021), o API Gateway desempenha papel fundamental na coordenação das requisições externas, permitindo a aplicação de políticas de segurança, autenticação e balanceamento de carga de forma centralizada. Essa abordagem contribui significativamente para a eficiência e a resiliência das comunicações em ambientes distribuídos. Entre suas principais funcionalidades, destacam-se:

- a) o roteamento inteligente, que encaminha as requisições para o serviço correto com base em rotas, métodos HTTP ou regras definidas;
- b) a autenticação e autorização centralizadas, que validam *tokens* e credenciais, protegendo os microsserviços de acessos não autorizados;
- c) o cache de respostas, responsável por armazenar temporariamente resultados de serviços e reduzir a latência em chamadas repetidas;
- d) o controle de carga e o *rate limiting*, utilizados para prevenir sobrecarga e garantir a estabilidade do sistema; e
- e) o versionamento de APIs, que permite disponibilizar diferentes versões de endpoints sem interromper clientes existentes (LERCHER et al., 2024).

Além disso, o API Gateway contribui diretamente para a observabilidade do sistema ao centralizar métricas e *logs* de requisições, o que facilita o monitoramento do tráfego e a identificação de falhas (KHAN et al., 2023). Essa visibilidade integrada

é essencial para arquiteturas que dependem de múltiplos serviços interconectados e visam garantir a rastreabilidade completa das comunicações.

Ferramentas amplamente utilizadas para essa função incluem NGINX, Kong e AWS API Gateway, que oferecem recursos avançados de balanceamento de carga, autenticação baseada em *tokens* e integração com provedores de identidade. Segundo Aksakalli et al. (2021), a adoção de um API Gateway é uma das boas práticas mais recomendadas para aumentar a resiliência, a segurança e a escalabilidade em arquiteturas de microsserviços.

2.3.5 Ferramentas de observabilidade

A observabilidade desempenha um papel essencial na manutenção da estabilidade e do desempenho em arquiteturas de microsserviços. Em sistemas distribuídos, falhas e degradação de performance podem ocorrer de forma localizada, tornando o diagnóstico mais complexo e exigindo o uso de ferramentas especializadas para coleta, análise e correlação de métricas. Nesse contexto, soluções como Prometheus, Grafana e Jaeger constituem um ecossistema amplamente utilizado para o monitoramento e o rastreamento de aplicações distribuídas (KHAN et al., 2023).

O Prometheus é uma ferramenta de monitoramento de código aberto baseada em séries temporais, projetada para coletar métricas de sistemas e aplicações em tempo real. Essa abordagem possibilita acompanhar indicadores como consumo de recursos, latência de requisições, estado de serviços e tráfego de rede. Além disso, ele oferece um mecanismo de alertas nativo, que auxilia na detecção proativa de falhas e na mitigação de incidentes críticos antes que comprometam o funcionamento do sistema (KHAN et al., 2023).

O Grafana, por sua vez, é uma plataforma de visualização de dados que complementa o Prometheus, fornecendo *dashboards* interativos e personalizados para análise de métricas. Essa integração favorece a criação de painéis unificados de observabilidade, que facilitam o acompanhamento de indicadores de desempenho e o diagnóstico de anomalias em tempo real. Devido à sua flexibilidade e compatibilidade com diversas fontes de dados, o Grafana consolidou-se como uma das ferramentas mais utilizadas em ecossistemas baseados em microsserviços (KHAN et al., 2023).

O Jaeger é uma ferramenta voltada ao rastreamento distribuído (*distributed tracing*), utilizada para mapear o percurso completo de uma requisição ao longo de múltiplos serviços. Essa funcionalidade permite identificar gargalos de latência, falhas de integração e dependências críticas entre componentes. Segundo Aksakalli et al. (2021), o *tracing* é um dos mecanismos mais eficazes para compreender o comportamento de sistemas distribuídos, pois fornece uma visão detalhada das interações entre serviços e facilita a detecção de pontos de falha ou de degradação de desempenho.

Em conjunto, Prometheus, Grafana e Jaeger formam uma base consolidada de ferramentas para observabilidade em arquiteturas de microsserviços, oferecendo recursos complementares de coleta, visualização e rastreamento. A utilização integrada dessas soluções contribui para aumentar a confiabilidade operacional e aprimorar o processo de diagnóstico e resolução de falhas em sistemas complexos (KHAN et al., 2023; AKSAKALLI et al., 2021).

Apesar da eficiência dessas ferramentas, a manutenção de múltiplos agentes e a falta de padronização entre fontes de dados ainda representam desafios em larga escala. Nesse contexto, o OpenTelemetry surge como uma abordagem unificada e extensível para instrumentação e coleta de telemetria em ambientes distribuídos (KHAN et al., 2023).

2.3.6 OpenTelemetry Operator

O OpenTelemetry é um projeto mantido pela *Cloud Native Computing Foundation* (CNCF) que fornece uma estrutura unificada para coleta, processamento e exportação de dados de observabilidade, ou seja, é um *framework*. Sua arquitetura é composta por APIs, *SDKs* (*Software Development Kit*) e pelo *Collector*, que atua como ponto central de agregação e roteamento de métricas, *logs* e *traces* (CNCF, 2025). Essa abordagem padronizada permite que diferentes linguagens e ferramentas possam ser integradas, reduzindo o acoplamento entre camadas de observabilidade e facilitando a interoperabilidade entre plataformas.

De acordo com Aksakalli et al. (2021), em sistemas baseados em microsserviços, a adoção de mecanismos de observabilidade padronizados é fundamental para sustentar os atributos de qualidade não funcionais, como desempenho, confiabilidade e escalabilidade. O OpenTelemetry, ao integrar os três

pilares da observabilidade (métricas, *logs* e *traces* distribuídos), permite identificar falhas de comunicação, dependências críticas e variações de latência entre serviços.

Já o OpenTelemetry Operator é uma extensão voltada especificamente para ambientes Kubernetes, projetada para simplificar a implantação e o gerenciamento dos componentes do OpenTelemetry. Operando de forma declarativa, ele automatiza a configuração e a atualização dos Collectors em todo o cluster, garantindo a consistência das políticas de coleta e exportação de dados de telemetria (CNCF, 2025).

Embora o OpenTelemetry seja agnóstico quanto à escolha da plataforma de armazenamento e análise de dados, ele pode ser integrado a diferentes soluções de backend, como o Elastic APM, o Prometheus ou o Grafana Tempo, que atuam como destino final dos dados coletados, permitindo visualização, análise de desempenho e identificação de anomalias (KHAN et al., 2023; ELASTIC, 2025).

Em síntese, o OpenTelemetry e seu operador fornecem um modelo padronizado, extensível e automatizado para monitoramento de microsserviços. Essa padronização contribui diretamente para a escalabilidade e a confiabilidade das aplicações distribuídas, além de fortalecer as práticas de engenharia observável em arquiteturas nativas em nuvem (KHAN et al., 2023; RODRIGUES; RITO SILVA; AVRITZER, 2025).

2.3.7 Redis

O Redis é um sistema de armazenamento de dados em memória. Segundo a Redis Labs (2025), ele oferece suporte a diferentes estruturas de dados, como listas, conjuntos, mapas e *strings*. Também segundo a documentação oficial, ele não é baseado em disco, ou seja, pode realizar operações com baixa latência, o que o torna adequado para aplicações que exigem respostas em tempo real.

Uma das aplicações que cabe ao Redis é o cache de dados frequentemente acessados, reduzindo a carga sobre bancos de dados relacionais e por consequente podendo a vim melhorar o tempo de resposta das aplicações. Essa estratégia de cache é especialmente vantajosa em ambientes distribuídos e de alta concorrência, nos quais a eficiência no acesso a dados é determinante para a escalabilidade e o desempenho global do sistema (AKSAKALLI et al., 2021).

Outra aplicação relevante é o uso do Redis como fila de mensagens leve, uma alternativa simplificada em cenários nos quais a complexidade de ferramentas como RabbitMQ ou Apache Kafka não se justifica. Nesses casos, o Redis pode atuar como broker, permitindo o processamento desacoplado de eventos e mensagens entre serviços (KHAN et al., 2023).

Por fim, o Redis também é empregado na implementação de bloqueios distribuídos (*distributed locks*), utilizados para garantir consistência em operações críticas que envolvem acesso simultâneo a recursos compartilhados. Essa funcionalidade é essencial em sistemas altamente concorrentes, nos quais múltiplas instâncias precisam coordenar atualizações de estado sem comprometer a integridade dos dados (AKSAKALLI et al., 2021).

De modo geral, as ferramentas apresentadas desempenham papéis complementares dentro de uma arquitetura de microsserviços. Enquanto RabbitMQ e Kafka garantem comunicação assíncrona eficiente, o Kubernetes automatiza a orquestração e o escalonamento; o API Gateway centraliza o controle e a segurança das requisições; Prometheus, Grafana e Jaeger asseguram visibilidade operacional; e o Redis otimiza o desempenho e a consistência de dados temporários. Juntas, essas soluções constituem a base tecnológica necessária para sustentar sistemas distribuídos escaláveis, resilientes e observáveis (AKSAKALLI et al., 2021; KHAN et al., 2023; RODRIGUES; RITO SILVA; AVRITZER, 2025).

2.4 DESAFIOS DA COMUNICAÇÃO ENTRE MICROSERVIÇOS

Embora a arquitetura de microsserviços ofereça benefícios em termos de escalabilidade, modularidade e resiliência, a comunicação entre serviços distribuídos continua sendo um dos aspectos mais complexos e desafiadores desse modelo. Em contraste com sistemas com maior centralização, onde os componentes compartilham o mesmo espaço de execução e a troca de informações ocorre internamente, nas arquiteturas distribuídas cada microsserviço opera de forma independente e interage com os demais por meio de troca de mensagens. Essa descentralização, apesar de trazer flexibilidade, introduz uma série de problemas técnicos e operacionais relacionados à latência, sincronização, segurança e observabilidade (AKSAKALLI et al., 2021).

Segundo Fowler (2017), a comunicação entre microsserviços envolve não apenas a troca de dados, mas também a coordenação entre componentes autônomos, que podem falhar, escalar ou se atualizar de forma independente. Essa característica impõe desafios de integração e consistência, especialmente em cenários de alta concorrência e dependências complexas. Além disso, o desempenho global de uma aplicação distribuída tende a ser limitado pelo serviço mais lento da cadeia, o que exige estratégias específicas para mitigar falhas em cascata, acoplamento temporal e latência acumulada.

Compreender e mitigar os desafios de comunicação é essencial para assegurar que os princípios de escalabilidade, eficiência e resiliência sejam de fato alcançados.

2.4.1 Latência acumulada e Performance

A latência acumulada está diretamente relacionada à forma como os serviços interagem entre si. Em arquiteturas distribuídas, cada requisição entre microsserviços adiciona um tempo de resposta adicional, que se soma ao longo de cadeias de chamadas síncronas, mas pode ocorrer em cadeias híbridas também. Segundo Aksakalli et al. (2021), quanto maior o número de serviços intermediários envolvidos no processamento de uma requisição, maior o risco de acúmulo de latência e de falhas em cascata, comprometendo tanto a escalabilidade quanto a experiência do usuário final.

Esse fenômeno é amplificado pela natureza das chamadas síncronas, nas quais cada serviço precisa aguardar a resposta do próximo antes de continuar seu processamento. Em um fluxo complexo, com múltiplas dependências, qualquer atraso em um ponto intermediário propaga-se por toda a cadeia, resultando em aumento exponencial do tempo total de resposta. Fowler (2017) destaca que esse tipo de acoplamento temporal é um dos principais fatores de degradação de desempenho em microsserviços, uma vez que multiplica o número de interações de rede e amplia o tempo médio de resposta (*RTT – Round Trip Time*).

Além da latência de rede, outros fatores contribuem para a degradação de performance, como serialização e desserialização de mensagens, formatação de dados, sobrecarga de APIs intermediárias e balanceadores de carga. Rodrigues, Rito Silva e Avritzer (2025) observam que esses elementos, quando combinados,

podem reduzir significativamente a escalabilidade do sistema e aumentar o consumo de recursos computacionais, especialmente em ambientes com alta concorrência e grande volume de transações.

Para mitigar o problema da latência acumulada, Aksakalli et al. (2021) e Fowler (2017) recomendam a adoção de estratégias como: Desacoplamento assíncrono, reduzindo dependências diretas entre serviços e permitindo processamento paralelo de tarefas; *Cache local* e distribuído, diminuindo o número de chamadas remotas desnecessárias; Agrupamento e compactação de chamadas (*batching* e *compression*), reduzindo o *overhead* de comunicação; Monitoramento contínuo de tempo de resposta e análise de métricas, permitindo identificar gargalos de performance e ajustar a arquitetura conforme o perfil de uso.

Rodrigues, Rito Silva e Avritzer (2025) destacam que o gerenciamento de latência em microsserviços não deve ser tratado apenas como uma questão técnica, mas como um requisito arquitetural. O design das dependências, o número de hops entre serviços e o uso de protocolos de comunicação eficientes devem ser definidos estrategicamente desde o planejamento do sistema, visando garantir escalabilidade sustentável e desempenho previsível mesmo em cenários de alta carga.

Portanto, a latência acumulada é uma consequência inerente da comunicação distribuída e pode comprometer tanto a eficiência operacional quanto a qualidade do serviço percebida pelo usuário. Controlá-la exige uma combinação de boas práticas arquitetônicas, escolha adequada de protocolos e implementação de mecanismos de observabilidade capazes de identificar e isolar gargalos em tempo real.

2.4.2 Acoplamento temporal

O acoplamento temporal compromete a independência e a resiliência da arquitetura. Esse tipo de acoplamento ocorre quando a disponibilidade ou o tempo de resposta de um serviço afeta diretamente a execução de outro, criando uma dependência temporal entre eles. Em sistemas com maior centralização, essas dependências são internas e mais fáceis de controlar; entretanto, em arquiteturas distribuídas, algumas interações entre serviços dependem da disponibilidade remota, tornando o sistema mais vulnerável a falhas e latências (AKSAKALLI et al., 2021).

De acordo com Fowler (2017), o acoplamento temporal é particularmente crítico em comunicações síncronas, nas quais um serviço solicitante precisa aguardar a resposta de outro para continuar seu processamento. Quando um serviço *downstream* apresenta lentidão ou indisponibilidade, o tempo de resposta global aumenta e pode desencadear um efeito cascata, em que múltiplos serviços são bloqueados em sequência. Esse fenômeno reduz a escalabilidade e pode levar à indisponibilidade parcial ou total do sistema.

Rodrigues, Rito Silva e Avritzer (2025) destacam que esse tipo de dependência aumenta a fragilidade operacional da arquitetura, pois a falha de um único componente pode se propagar rapidamente em sistemas com alta densidade de chamadas síncronas. Isso é especialmente problemático em fluxos críticos de negócio, como autenticação, pagamento ou sincronização de dados, onde múltiplos serviços precisam responder dentro de limites rígidos de tempo. Será discutido mais sobre a tolerância a essas falhas a seguir.

Para reduzir os impactos do acoplamento temporal, a literatura recomenda a adoção de padrões de resiliência e mecanismos de desacoplamento. Entre as estratégias mais eficazes estão: *Circuit breaker*: atua como um disjuntor que interrompe temporariamente a comunicação com serviços instáveis, prevenindo sobrecarga e permitindo que o sistema degrade de forma controlada (FOWLER, 2017; AKSAKALLI et al., 2021); *Retry e backoff exponencial*: permitem repetir automaticamente requisições falhas, introduzindo intervalos progressivos entre tentativas para evitar saturar serviços em recuperação; *Fallback*: define respostas alternativas ou comportamentos padrão quando um serviço dependente está indisponível, garantindo a continuidade parcial das operações (FOWLER, 2017); *Timeouts configuráveis*: limitam o tempo de espera por respostas, evitando que recursos fiquem presos em chamadas bloqueantes e reduzindo o tempo de recuperação global.

Além desses mecanismos, a adoção de comunicação assíncrona, por meio de mensageria, é amplamente recomendada para reduzir o acoplamento temporal. Ao desacoplar as interações entre serviços, é possível manter o fluxo de processamento mesmo em casos de falha temporária ou lentidão, aumentando a resiliência e a disponibilidade do sistema, no entanto *backpressure* e *queue overflow* podem ocorrer em sistemas síncronos (RODRIGUES; RITO SILVA; AVRITZER, 2025).

2.4.3 Tolerância a falhas

Além da questão do acoplamento temporal, outro aspecto diretamente relacionado é a necessidade de tolerância a falhas, um dos pilares fundamentais para garantir a confiabilidade e a continuidade operacional em arquiteturas baseadas em microsserviços. Segundo Aksakalli et al. (2021), a natureza distribuída desses sistemas os torna intrinsecamente mais suscetíveis a falhas parciais, como interrupções de rede, indisponibilidade temporária de serviços ou inconsistências na comunicação entre nós. Diferentemente de sistemas monolíticos, em que uma falha geralmente é localizada e previsível, nas arquiteturas distribuídas os efeitos podem se propagar de forma imprevisível e causar falhas em cascata.

Fowler (2017) destaca que, em ambientes compostos por dezenas ou centenas de microsserviços interdependentes, é impossível evitar completamente falhas; portanto, a arquitetura deve ser projetada para resistir e se recuperar delas. Isso implica adotar estratégias que garantam a continuidade do sistema mesmo quando componentes individuais se tornam inoperantes. Um dos mecanismos mais utilizados para esse fim é o circuit breaker (FOWLER, 2017; AKSAKALLI et al., 2021).

Outro padrão amplamente adotado é o SAGA, que trata da consistência em transações distribuídas. Nesse modelo, uma operação de negócio é dividida em uma sequência de transações locais, cada uma executada por um microsserviço distinto. Caso uma dessas etapas falhe, ações de compensação são executadas para desfazer os efeitos das operações anteriores, garantindo que o sistema retorne a um estado consistente (AKSAKALLI et al., 2021; RODRIGUES; RITO SILVA; AVRITZER, 2025).

A comunicação assíncrona também exerce papel central na tolerância a falhas. Ao desacoplar temporalmente os serviços, ela evita que a indisponibilidade de um componente bloqueie o processamento dos demais. Tecnologias de mensageria, como RabbitMQ, permitem armazenar mensagens em filas ou tópicos até que o consumidor esteja disponível para processá-las, reduzindo o impacto de interrupções temporárias e promovendo recuperação automática após falhas (AKSAKALLI et al., 2021).

Além disso, muitos sistemas empregam mecanismos de DLQ, filas específicas para mensagens que não puderam ser entregues ou processadas corretamente após múltiplas tentativas. Essa abordagem impede a perda definitiva de dados e facilita a análise de erros, permitindo que as mensagens sejam reprocessadas ou investigadas posteriormente (KHAN et al., 2023).

Rodrigues, Rito Silva e Avritzer (2025) complementam que a implementação de mecanismos de tolerância a falhas deve considerar o equilíbrio entre resiliência e desempenho, pois a redundância e o reprocessamento podem aumentar a latência e o consumo de recursos. Assim, a eficácia dessas estratégias depende de um planejamento cuidadoso da comunicação entre microsserviços, priorizando degradação controlada em vez de falha total.

Em suma, a tolerância a falhas em microsserviços vai além da simples recuperação de erros: ela envolve projetar a comunicação de modo que falhas sejam esperadas, detectadas e isoladas rapidamente, garantindo que o sistema continue operando de forma previsível mesmo em condições adversas.

2.4.4 Orquestração e escalonamento

À medida que o número de microsserviços em uma aplicação cresce, a complexidade da comunicação entre eles aumenta exponencialmente. Gerenciar o ciclo de vida, a disponibilidade e a interação entre dezenas ou centenas de serviços requer mecanismos automatizados de orquestração e escalonamento, capazes de coordenar implantações, balancear carga e garantir resiliência.

Segundo Aksakalli et al. (2021), a orquestração em arquiteturas de microsserviços consiste na automatização das tarefas operacionais que envolvem o provisionamento, a configuração, o monitoramento e a recuperação de serviços distribuídos. Ferramentas como o Kubernetes desempenham papel central nesse processo, pois permitem definir políticas de implantação e de escalonamento de forma declarativa, assegurando consistência operacional e eficiência em ambientes altamente dinâmicos (AKSAKALLI et al., 2021).

Um dos elementos centrais nesse contexto é o mecanismo de descoberta de serviços (*service discovery*), responsável por permitir que microsserviços localizem e se comuniquem entre si sem dependência de endereços fixos. A literatura descreve três abordagens principais para esse processo. Na descoberta *client-side*, o próprio

serviço cliente consulta um registro central (*service registry*), como Consul ou Eureka, e escolhe a instância mais adequada do serviço alvo, executando também o balanceamento de carga. Já na descoberta *server-side*, o cliente delega a resolução ao balanceador de carga ou ao orquestrador, que identifica a instância correta e direciona a requisição. Por fim, modelos híbridos combinam ambas as abordagens, utilizando *sidecars* ou *service meshes* (como Istio ou Linkerd) para realizar a resolução e o roteamento de tráfego de forma transparente e padronizada (KHAN et al., 2023).

No contexto de ambientes modernos de orquestração, o Kubernetes adota predominantemente o modelo *server-side*, no qual o próprio cluster gerencia o processo de descoberta. Cada serviço implantado é registrado automaticamente no *CoreDNS*, que resolve o nome do serviço para o endereço do *ClusterIP* correspondente. Dessa forma, quando um microsserviço cliente realiza uma chamada para outro serviço, a resolução do destino ocorre internamente via DNS, sem a necessidade de intervenção do desenvolvedor ou lógica de descoberta manual no código. Esse mecanismo simplifica a comunicação, aumenta a resiliência e reduz o acoplamento entre microsserviços, uma vez que novas instâncias podem ser criadas, removidas ou reiniciadas sem impacto sobre as conexões ativas (RODRIGUES; RITO SILVA; AVRITZER, 2025).

Além da descoberta de serviços, o balanceamento de carga dinâmico é uma função essencial da orquestração. No Kubernetes, esse papel é desempenhado automaticamente pelos *services* de tipo *ClusterIP* ou *LoadBalancer*, que distribuem requisições entre múltiplas instâncias de *pods*, agrupamentos de um ou mais containers que são executados juntos em um mesmo nó de um *cluster*. Essa camada de abstração é complementada por políticas de *readiness* e *liveness probes*, que garantem que apenas instâncias saudáveis participem do balanceamento. Essa automação reduz a probabilidade de falhas em cascata e melhora a disponibilidade do sistema em cenários de alto tráfego (AKSAKALLI et al., 2021).

O escalonamento automático, por sua vez, é outro componente essencial para a manutenção da eficiência e do desempenho do sistema. Ele pode ocorrer de forma horizontal, com a replicação de instâncias de serviços conforme a demanda, ou vertical, por meio do aumento de recursos computacionais alocados a um *pod*. Segundo Rodrigues, Rito Silva e Avritzer (2025), o escalonamento horizontal é

amplamente adotado em ambientes distribuídos, pois favorece a elasticidade e a resiliência do sistema, mas exige mecanismos de sincronização e monitoramento contínuos para evitar inconsistências entre as instâncias e garantir a coerência dos estados das aplicações.

Outro aspecto relevante é a coordenação entre orquestração e comunicação. Em sistemas que utilizam múltiplos protocolos e padrões de integração, a orquestração deve garantir que o escalonamento e o redirecionamento de tráfego não interrompam transações ativas nem provoquem perda de mensagens em fluxos assíncronos. Conforme Khan et al. (2023), a integração entre orquestração e observabilidade é fundamental para detectar gargalos de comunicação, ajustar políticas de escalonamento em tempo real e manter a disponibilidade do sistema sob diferentes cargas de trabalho.

Conforme dito, a orquestração e o escalonamento em arquiteturas de microsserviços são elementos indispensáveis para a comunicação eficiente e resiliente, mas também representam uma das principais fontes de complexidade operacional. Conforme Aksakalli et al. (2021) e Rodrigues, Rito Silva e Avritzer (2025), a capacidade de gerenciar dinamicamente a alocação de recursos, monitorar fluxos de comunicação e adaptar o sistema de forma autônoma é determinante para garantir a escalabilidade, a confiabilidade e a continuidade dos serviços em ambientes distribuídos.

2.4.5 Consistência e gerenciamento de dados

Um dos principais desafios nas arquiteturas baseadas em microsserviços é o gerenciamento de dados de forma consistente e eficiente. Conforme destaca Fowler (2017), a descentralização de dados é um dos princípios fundamentais desse modelo arquitetural: cada microsserviço deve possuir e gerenciar seu próprio banco de dados, mantendo independência em relação aos demais. Essa prática favorece a autonomia e o isolamento de falhas, mas também introduz um problema significativo: A manutenção da consistência entre diferentes contextos de dados distribuídos.

Em aplicações monolíticas, a consistência é normalmente garantida por meio de transações atômicas (ACID), que asseguram que todas as operações sejam concluídas com sucesso ou revertidas em caso de erro. Entretanto, em um sistema

distribuído, onde os microsserviços se comunicam por rede e mantêm seus próprios repositórios de dados, transações distribuídas ACID tornam-se inviáveis ou extremamente custosas em termos de desempenho (AKSAKALLI et al., 2021; LENARDUZZI et al., 2020).

Para lidar com essa limitação, muitos sistemas adotam o modelo consistência eventual, no qual os dados podem ficar temporariamente divergentes, mas são sincronizados posteriormente até alcançar um estado consistente. Essa abordagem melhora a escalabilidade e a disponibilidade, mas impõe desafios de controle e rastreabilidade de eventos. Aksakalli et al. (2021) ressaltam que, embora a consistência eventual aumente a resiliência do sistema, ela exige mecanismos complementares de coordenação e compensação de falhas.

Dentre as estratégias para lidar com inconsistências temporárias estão o padrão SAGA, já citado anteriormente, e o *event sourcing*. O padrão SAGA define uma sequência de transações locais distribuídas, em que cada operação bem-sucedida é seguida de uma ação de compensação correspondente em caso de falha subsequente, garantindo assim uma forma de reversão lógica (AKSAKALLI et al., 2021). Já o *event sourcing* propõe o armazenamento do histórico completo de eventos que modificam o estado do sistema, permitindo reconstruir o estado atual a partir da sequência de mudanças, o que facilita a auditoria e a recuperação de consistência após falhas (RODRIGUES; RITO SILVA; AVRITZER, 2025).

Além disso, Fowler (2017) recomenda a aplicação de boas práticas de isolamento de domínios, inspiradas nos princípios de *Domain-Driven Design* (DDD), de modo que os microsserviços sejam organizados em torno de contextos bem delimitados (*bounded contexts*). Esse isolamento reduz o acoplamento entre domínios de negócio e evita a necessidade de coordenação transacional entre serviços distintos, diminuindo a probabilidade de inconsistências.

Outra abordagem recorrente na literatura é a replicação seletiva de dados, em que cada serviço mantém cópias parciais de informações externas estritamente necessárias para sua operação, podendo ser utilizado pode ser combinado com CQRS. Segundo Lenarduzzi et al. (2020), essa estratégia reduz o número de chamadas remotas e, conseqüentemente, a latência e a vulnerabilidade a falhas de comunicação, mas requer mecanismos de sincronização e versionamento de dados para evitar desatualizações.

Por fim, Rodrigues, Rito Silva e Avritzer (2025) destacam que a escolha entre consistência forte e eventual deve ser orientada pelos requisitos não funcionais do sistema, como disponibilidade, desempenho e tolerância a falhas. Em sistemas de alta criticidade, como transações financeiras, a consistência forte continua sendo necessária, enquanto em aplicações de grande escala voltadas à experiência do usuário, a consistência eventual é mais adequada por favorecer escalabilidade e desempenho.

Através do exposto, pode-se dizer que o gerenciamento de dados em microsserviços requer um equilíbrio entre autonomia e consistência. A adoção de padrões como SAGA, event sourcing e replicação seletiva, aliada ao isolamento de domínios e a um monitoramento contínuo, constitui um conjunto de práticas essenciais para garantir confiabilidade e previsibilidade em sistemas distribuídos modernos.

2.4.6 Observabilidade e monitoramento

A comunicação distribuída em arquiteturas de microsserviços introduz desafios significativos de monitoramento, rastreamento e correlação de eventos, uma vez que cada serviço opera de forma independente e comunica-se com os demais por meio de interfaces remotas. Diferentemente das aplicações monolíticas, em que o comportamento do sistema pode ser observado a partir de um único ponto de controle, em sistemas distribuídos as falhas e degradações de desempenho frequentemente ocorrem de maneira fragmentada e difícil de rastrear (AKSAKALLI et al., 2021).

Segundo Khan et al. (2023), a ausência de mecanismos de observabilidade adequados é uma das principais causas de aumento do MTTD e do tempo médio de recuperação de falhas (MTTR) em sistemas baseados em microsserviços. Como os componentes estão isolados e hospedados em múltiplas instâncias, a simples análise de *logs* locais não é suficiente para compreender o comportamento global da aplicação. Falhas intermitentes, dependências circulares ou degradações de rede podem se propagar silenciosamente entre serviços, dificultando a identificação da causa raiz e comprometendo a confiabilidade do sistema.

Um dos desafios mais críticos está na fragmentação das fontes de dados. Cada microsserviço gera suas próprias métricas, *logs* e *traces*, geralmente em

formatos distintos e com granularidades variadas. A correlação entre esses dados, necessária para compreender o percurso de uma requisição ponta a ponta, requer padronização, centralização e sincronização temporal (KHAN et al., 2023). Essa heterogeneidade aumenta a complexidade de análise e pode levar a diagnósticos imprecisos, especialmente em arquiteturas que não possuem um mecanismo unificado de observabilidade, como o OpenTelemetry discutido anteriormente.

Além da fragmentação, há o problema da sobrecarga de instrumentação. Conforme apontam Rodrigues, Rito Silva e Avritzer (2025), a coleta intensiva de métricas e *logs* pode introduzir impacto perceptível na performance dos microsserviços, principalmente em sistemas de alta taxa de requisições. O excesso de dados gera ruído operacional, aumenta o consumo de recursos de armazenamento e processamento e pode degradar o próprio desempenho do sistema que se pretende observar. Assim, torna-se necessário definir limites e políticas de amostragem que equilibrem visibilidade e eficiência.

Outro desafio relevante é a correlação de falhas em ambientes dinâmicos. Em plataformas orquestradas por Kubernetes, por exemplo, instâncias de serviços podem ser escaladas, reiniciadas ou migradas entre nós de forma automática. Isso faz com que os identificadores de instância e os contextos de execução mudem constantemente, tornando o rastreamento de uma transação distribuída uma tarefa complexa. Ferramentas de tracing distribuído, como Jaeger e OpenTelemetry, mitigam esse problema ao propagar identificadores de correlação (trace IDs) ao longo das requisições, mas a eficácia desse processo depende de uma instrumentação consistente em todos os serviços (KHAN et al., 2023; AKSAKALLI et al., 2021).

Além disso, a observabilidade em tempo real representa um desafio operacional e econômico. O volume de dados gerado por métricas e *logs* pode alcançar níveis massivos em sistemas de grande escala, tornando inviável o armazenamento integral dessas informações. Conforme discutem Aksakalli et al. (2021), a adoção de *pipelines* de dados otimizados, amostragem inteligente e compressão de *logs* é essencial para equilibrar custo e desempenho sem comprometer a detecção de anomalias e o diagnóstico de falhas.

Por fim, há o desafio cultural e organizacional. Fowler (2017) ressalta que a observabilidade não deve ser tratada apenas como um problema técnico, mas como uma prática de engenharia integrada ao ciclo de vida do software. Isso implica

incorporar métricas de saúde e rastreamento desde o design dos serviços, promovendo uma cultura de monitoramento proativo e colaboração entre times de desenvolvimento e operações. Sem essa maturidade organizacional, mesmo ferramentas avançadas falham em gerar insights relevantes, resultando em reatividade em vez de prevenção.

Em resumo, a observabilidade e o monitoramento representam não apenas um requisito técnico, mas um desafio sistêmico nas arquiteturas de microsserviços. A integração de *logs*, métricas e *traces* distribuídos requer padronização, governança de dados e disciplina de engenharia. A ausência desses elementos compromete a confiabilidade, dificulta a resolução de incidentes e reduz a capacidade de evolução contínua dos sistemas distribuídos.

2.4.7 Segurança da Comunicação

A segurança é um dos desafios mais críticos na comunicação entre microsserviços, pois envolve proteger a integridade, a confidencialidade e a autenticidade das interações entre componentes distribuídos. Em aplicações monolíticas, o controle de acesso e a validação de dados são realizados em um único ponto central, o que simplifica a implementação de políticas de segurança. No entanto, nas arquiteturas baseadas em microsserviços, essa responsabilidade é descentralizada: cada serviço expõe interfaces próprias, o que amplia significativamente a superfície de ataque (AKSAKALLI et al., 2021).

Segundo Fowler (2017), a fragmentação dos componentes impõe a necessidade de replicar mecanismos de autenticação, autorização e criptografia em múltiplos pontos da arquitetura. Isso torna o gerenciamento de credenciais, certificados e *tokens* de acesso mais complexo, além de aumentar o risco de configurações inconsistentes entre serviços. A falta de padronização desses mecanismos pode levar a vulnerabilidades como exposição indevida de endpoints, interceptação de dados e escalonamento não autorizado de privilégios.

Um dos principais desafios é o gerenciamento de identidade distribuída. Enquanto em sistemas centralizados é comum o uso de autenticação baseada em sessão, nas arquiteturas distribuídas recomenda-se o uso de *tokens* portáteis, como JWT (*JSON Web Tokens*), emitidos por provedores de identidade e validados por cada microsserviço de forma independente. Essa abordagem, conhecida como

token-based authentication, permite a verificação autônoma de permissões, mas requer mecanismos robustos de renovação e revogação para evitar uso indevido de credenciais expiradas (AKSAKALLI et al., 2021; RODRIGUES; RITO SILVA; AVRITZER, 2025).

Outro aspecto essencial é a criptografia do tráfego interno e externo. O uso de TLS/SSL é fundamental para prevenir ataques de interceptação (man-in-the-middle) e assegurar a confidencialidade das mensagens trocadas entre microsserviços. Contudo, como destacam Aksakalli et al. (2021), a manutenção de certificados e chaves em ambientes de alta escala pode introduzir sobrecarga operacional e riscos de configuração incorretos. Estratégias como mutual TLS (mTLS) e service mesh (ex.: Istio, Linkerd) são frequentemente adotadas para automatizar a autenticação mútua entre serviços e garantir a integridade das conexões.

Além da autenticação e criptografia, há o desafio da validação de payloads e filtragem de tráfego. Cada microsserviço precisa validar os dados recebidos antes do processamento, a fim de prevenir injeções maliciosas, estouros de buffer e outros ataques que exploram vulnerabilidades em APIs públicas (FOWLER, 2017; KHAN et al., 2023). Essa prática deve ser acompanhada por mecanismos de rate limiting, input sanitization e request throttling, especialmente em serviços expostos a usuários externos.

O uso de um API Gateway é amplamente recomendado como camada de segurança adicional. Ele atua como ponto central de controle, concentrando autenticação, autorização, cache, limitação de taxa e monitoramento de requisições. Conforme Aksakalli et al. (2021) e Khan et al. (2023), o gateway também permite a integração com provedores de identidade (como OAuth 2.0 e OpenID Connect), simplificando a aplicação de políticas de segurança consistentes em todo o ecossistema.

Resumidamente, a segurança da comunicação entre microsserviços transcende o uso de criptografia e autenticação: ela depende de um conjunto de práticas integradas que incluem governança de identidade, monitoramento contínuo, automação de certificados e políticas de confiança mínima. A ausência dessas medidas pode transformar a comunicação distribuída, originalmente projetada para garantir autonomia e escalabilidade, em um vetor de vulnerabilidade e exposição de dados sensíveis.

O Quadro 2 abaixo resume os principais desafios levantados, resumindo seus principais impactos no sistema e as estratégias de solução ou prevenção.

Quadro 2 – Síntese dos principais desafios

Desafio	Impacto no Sistema	Estratégia de Mitigação
Latência acumulada	Aumento do tempo de resposta e degradação da experiência do usuário final.	Utilização de gRPC e protocolos binários de baixo overhead; comunicação assíncrona; uso de cache distribuído e compressão de chamadas (batching).
Acoplamento temporal	Bloqueio de serviços interdependentes e propagação de falhas em cascata.	Implementação de padrões de resiliência como Circuit Breaker, Fallback, Retries e Timeouts; adoção de mensageria assíncrona.
Tolerância a falhas	Interrupção parcial do sistema e perda temporária de dados.	Uso de filas e tópicos de mensagens (RabbitMQ, Kafka); padrão SAGA para compensação; replicação e isolamento de falhas.
Orquestração e descoberta	Escalonamento ineficiente, falhas de roteamento e inconsistência entre instâncias.	Kubernetes, service mesh, service registry
Consistência de dados	Dados conflitantes ou desatualizados entre serviços.	Eventual consistency, SAGA, event sourcing
Observabilidade	Dificuldade de diagnóstico, rastreamento e correlação de falhas.	Logging centralizado, tracing, métricas
Segurança	Vazamento de dados, interceptação de tráfego e vulnerabilidade de rede.	API Gateway, autenticação, TLS/SSL

Fonte: elaborado pelo autor.

Em síntese, a fundamentação teórica deste trabalho consolidou os principais conceitos relacionados à arquitetura de microsserviços e aos desafios que envolvem sua implementação em ambientes distribuídos. Foram abordados aspectos essenciais como comunicação entre serviços, orquestração, consistência de dados, observabilidade e segurança. Todos fundamentais para o entendimento das fragilidades e oportunidades de melhoria presentes na infraestrutura da Keeps. O estudo desses conceitos, aliado à análise de abordagens propostas por autores como Aksakalli et al. (2021), Khan et al. (2023) e Rodrigues, Rito Silva e Avritzer (2025), proporcionaram a base teórica necessária para o desenvolvimento das melhorias técnicas apresentadas nos capítulos seguintes, garantindo alinhamento entre as soluções aplicadas e as práticas recomendadas pela literatura contemporânea.

3 TRABALHOS RELACIONADOS

Adiante, neste capítulo, apresenta-se uma revisão das principais pesquisas e estudos acadêmicos que abordam os desafios e soluções no campo da comunicação entre microsserviços, especialmente em arquiteturas escaláveis e distribuídas. Este capítulo explora os avanços na literatura sobre os padrões de comunicação, as práticas recomendadas, as tecnologias emergentes e os resultados experimentais que têm influenciado a implementação e otimização dessas arquiteturas. Além disso, discute as lacunas existentes e como os estudos revisados oferecem embasamento teórico para as melhorias aplicadas no estudo de caso da Keeps.

3.1 PESQUISA BIBLIOGRÁFICA

A pesquisa bibliográfica teve como objetivo identificar e analisar estudos recentes que abordam os desafios e as estratégias de comunicação entre microsserviços em arquiteturas escaláveis. Buscou-se construir uma base teórica sólida e atual, que sustentasse a discussão sobre os aspectos de desempenho, resiliência, monitoramento e consistência em sistemas distribuídos. As consultas foram realizadas na base científica ScienceDirect (Elsevier) reconhecida pela ampla cobertura de publicações revisadas por pares na área de Engenharia de Software.

As palavras-chave foram definidas em inglês para garantir maior abrangência internacional dos resultados, com combinações articuladas por operadores booleanos (AND, OR), a fim de refinar o escopo da pesquisa. As principais combinações utilizadas foram:

As principais combinações de palavras-chave utilizadas foram:

- a) *"Microservices" AND "Communication"*;
- b) *"Microservices" AND "Communication" AND "Scalability"*;
- c) *"Microservices" AND "Protocols" AND "Performance"*;
- d) *"Microservices" AND "Message Broker" AND "gRPC"*;
- e) *"Microservice architecture" AND "performance" AND "scalability"*;
- f) *"Monolithic" AND "Microservices" AND "Migration"*;
- g) *"Microservices" AND ("Monitoring" OR "Observability" OR "Failures")*.

O período de busca foi delimitado entre 2020 e 2025, de modo a contemplar os estudos mais recentes e relevantes sobre o tema. Foram aplicados filtros para restringir os resultados a *Research Articles* e *Review Articles*, garantindo a inclusão apenas de trabalhos completos, revisados por pares e com validação empírica.

Durante a fase inicial, as consultas retornaram aproximadamente 5.000 publicações somadas entre as bases. O processo de seleção seguiu três etapas sistemáticas:

1. Filtragem por título e palavras-chave: Foram descartados artigos cujo título não indicava relação direta com comunicação, desempenho ou integração em microserviços.
2. Leitura de resumos e palavras-chave: Nesta fase, foram excluídos estudos voltados exclusivamente a aspectos de modelagem conceitual, ensino ou comparação entre *frameworks* sem abordagem experimental.
3. Avaliação completa do conteúdo: Foram analisados integralmente os artigos com validação prática, métricas de desempenho ou propostas de arquitetura testadas em ambiente real ou simulado.

Como critério de exclusão, foram descartados artigos:

- a) Repetidos em diferentes bases;
- b) Sem aplicação prática no contexto da comunicação entre microserviços;
- c) Desatualizados ou focados apenas em aspectos conceituais sem validação experimental.

Após a análise dos principais resultados e leitura de resumos, foram selecionados os artigos mais relevantes, considerando:

- a) Adequação direta ao tema de comunicação entre microserviços;
- b) Discussão aprofundada sobre protocolos, desempenho, escalabilidade, resiliência, monitoramento ou migração arquitetural;
- c) Contribuições teóricas e práticas que pudessem enriquecer o desenvolvimento deste trabalho.

Além disso, é importante destacar que o artigo de Niswar et al. (2024) não foi encontrado diretamente nessa plataforma. No entanto, ele foi identificado e selecionado por meio do Google Scholar, que se revelou uma ferramenta útil para encontrar o artigo publicado na IJET (*International Journal of Electronics and Telecommunications*), uma revista acadêmica amplamente respeitada na área.

A Tabela 1 apresenta um resumo das buscas realizadas e dos artigos selecionados.

Tabela 1 – Busca Bibliográfica

Artigo Selecionado	Base Consultada	Palavras-chave Utilizadas	Retornados x Selecionados
AKSAKALLI, I. K. et al. (2021)	ScienceDirect	"Microservices" AND "Communication" AND "Scalability"	1.331 x 2
LERCHER, T. et al. (2024)	ScienceDirect	"Microservices" AND "Communication" AND "Scalability"	1.331 x 2
NISWAR, M. et al. (2024)	Google Scholar / IJET	"Microservices" AND "Communication" AND "gRPC" AND "REST"	~ x 1
RODRIGUES, H. et al. (2025)	ScienceDirect	"Microservice architecture" AND "performance" AND "scalability"	752 x 1
KHAN, M. U. et al. (2023)	ScienceDirect	"Microservices" AND ("Monitoring" OR "Observability" OR "Failures")	1.468 x 1

Fonte: elaborado pelo autor.

A metodologia adotada garantiu transparência e reprodutibilidade na seleção dos artigos, documentando as palavras-chave, os filtros, o período e a quantidade de resultados analisados em cada etapa. No entanto, ressalta-se que não foram lidos todos os resumos de todos os artigos retornados pela listagem, ou seja, o viés da “relevância” da listagem da base afetou a seleção dos estudos. Essa abordagem possibilitou construir um referencial teórico, oferecendo base comparativa para a análise prática desenvolvida no contexto da Keeps.

3.2 ESTADO DA ARTE

O estado da arte sobre arquitetura de microsserviços reúne estudos que exploram tanto os benefícios quanto os desafios da comunicação entre serviços distribuídos. O objetivo desta seção é identificar as principais abordagens, lacunas e estratégias apontadas pela literatura recente, com ênfase nos fatores que influenciam o desempenho, a escalabilidade e a resiliência em ambientes complexos.

Aksakalli et al. (2021) avançam na categorização dos padrões de comunicação utilizados em ambientes distribuídos. Apresentam uma *Systematic Literature Review* (SLR) que categoriza os padrões de comunicação mais utilizados em arquiteturas de microsserviços, como a comunicação síncrona via REST e a comunicação assíncrona baseada em mensageria. O estudo evidencia que a

escolha inadequada de padrões pode gerar acoplamento excessivo, falhas em cascata e aumento da latência, comprometendo a resiliência e a eficiência do sistema.

Complementando essa análise teórica, Niswar et al. (2024) realizaram uma avaliação experimental controlada comparando os protocolos REST, GraphQL e gRPC em um ambiente de microsserviços. Os resultados demonstram que o gRPC apresenta menor latência e consumo de CPU, especialmente em cenários de alta carga e estruturas de dados aninhadas. Esse tipo de análise é especialmente relevante para a Keeps, que possui picos de carga e enfrenta desafios relacionados à escolha de protocolos eficientes.

Além dos desafios técnicos da comunicação, a evolução das APIs ao longo do tempo é um fator crítico em ambientes de microsserviços. Lercher et al. (2024) conduziram um estudo empírico envolvendo quinze organizações que utilizam microsserviços em produção, analisando as estratégias de versionamento e evolução de APIs. Os autores identificam desafios de compatibilidade, fragmentação e manutenção da retrocompatibilidade, destacando o impacto da evolução contínua das interfaces sobre a estabilidade dos sistemas distribuídos. Na Keeps, problemas similares já foram observados, principalmente durante atualizações de bibliotecas clientes e serviços de suporte.

O suporte à tomada de decisão arquitetural é explorado por Rodrigues et al. (2025), que realizaram uma revisão sistemática focada na avaliação de desempenho e escalabilidade em arquiteturas de microsserviços. O estudo identifica fatores arquitetônicos críticos que influenciam diretamente a comunicação eficiente entre serviços e orienta arquitetos e desenvolvedores na seleção de estratégias que atendam aos objetivos de escalabilidade, resiliência e otimização de recursos. Essas orientações são diretamente aplicáveis ao cenário da Keeps, que busca consolidar práticas defensivas e eficientes para suportar seu crescimento.

O trabalho de Gadelha (2023) apresenta um estudo de caso acadêmico que integra arquitetura de microsserviços e Domain-Driven Design (DDD) no desenvolvimento de um sistema de gerenciamento de pavimentos. Embora se trate de um trabalho de conclusão de curso, a pesquisa oferece evidências práticas sobre a importância da correta delimitação de domínios e da aplicação de padrões de comunicação eficientes, corroborando as práticas observadas na Keeps.

Por fim, Khan et al. (2023) destacam a importância das ferramentas de monitoramento para a operação eficiente de arquiteturas baseadas em microsserviços. O estudo sistematiza as principais ferramentas *DevOps* utilizadas para garantir observabilidade, como *Prometheus*, *Grafana* e *Jaeger*, e aponta que a ausência de monitoramento eficaz pode dificultar a identificação da origem de falhas, comprometer a confiabilidade do sistema e aumentar os riscos operacionais. Esse desafio está alinhado à realidade da Keeps, que possui dificuldades no tracing e no tratamento eficiente de falhas intermitentes.

Em suma, os trabalhos revisados convergem quanto à relevância da comunicação eficiente como elemento central da arquitetura de microsserviços. Entretanto, observa-se uma lacuna entre os estudos conceituais e as implementações práticas no contexto de plataformas corporativas específicas. Assim, este trabalho busca contribuir ao analisar, sob a perspectiva aplicada da Keeps, como diferentes estratégias e ferramentas de comunicação influenciam a escalabilidade, o desempenho e a resiliência de sistemas distribuídos em produção.

4 ESTUDO DE CASO: ARQUITETURA KEEPS

4.1 SOBRE A KEEPS

Atualmente, a Keeps atende várias empresas, abrangendo setores como varejo, serviços financeiros, tecnologia, saúde e indústria. Seu ecossistema digital possui mais de 500 mil colaboradores cadastrados, não é possível informar os números de usuários ativos na base, nem informações sobre os picos de usos por motivos de conformidade com políticas de sigilo. No entanto, os números internos refletem o crescimento contínuo de clientes e a necessidade de uma arquitetura escalável, resiliente e observável, fatores diretamente relacionados ao escopo deste estudo.

A plataforma é fornecida no modelo Software as a Service (SaaS), hospedada quase integralmente na infraestrutura da Amazon Web Services (AWS), o que favorece as características de elasticidade e implantação contínua de novas versões. Os ambientes são segmentados em clusters distintos, com controle de acesso, isolamento de dados e escalabilidade horizontal automatizada.

Entre as principais aplicações da plataforma destacam-se:

- a) Konquest – Aplicação *LMS/LXP* responsável pela gestão de cursos, trilhas de aprendizagem, matrículas, certificações e relatórios de progresso. Também integra funcionalidades de gamificação e desafios corporativos.
- b) MyAccount – Sistema de gerenciamento de contas e *workspaces*, que centraliza o controle de usuários, empresas, permissões e configurações de ambiente.
- c) Analytics – Aplicação voltada à geração de relatórios e painéis de Business Intelligence (BI), utilizando bases de dados não relacionais e consultas otimizadas para análise de desempenho.

Além dessas aplicações de maior porte, a Keeps opera um conjunto de microsserviços especializados, responsáveis por tarefas específicas dentro da arquitetura. No Quadro 3 estão listados os principais microsserviços e suas respectivas funções.

Quadro 3 – Microsserviços da Keeps

Microsserviço	Função Principal
Konquest Server	Serviço responsável pelo gerenciamento de cursos, matrículas, pulses e demais funções.
Analytics Server	Serviço responsável pela geração de relatórios e indexação de dados para alimentar BIs.
Corp	Serviço corporativo principal, integra dados organizacionais e autenticação central com os clientes.
Search	Serviço de indexação e busca de conteúdos.
Notification	Serviço de notificações assíncronas e push.
User Registration	Registro e onboarding de novos usuários.
Regulatory Compliance	Gerenciamento de conformidades regulatórias (NRs).
MyAccount	Gerenciamento de workspaces, contas e permissões.
Certificate Manager	Emissão e controle de certificados de matrículas.
Sisyphus	Serviço de processamento em background de ações em lote.
Custom Sections	Gerenciamento de seções customizadas em ambientes administrativos.
SmartZap	Integração com mensageria externa e automação de comunicação com o WhatsApp.
Keycloak	Sistema de autenticação e controle de acesso (Identity Provider), utilizado em diversos serviços.

Fonte: elaborado pelo autor.

A arquitetura técnica da Keeps é projetada para sustentar operações contínuas via CI/CD (*Continuous Integration / Continuous Delivery Continuous Deployment*). O sistema atende simultaneamente múltiplas organizações, com isolamento lógico por *tenant*, o que exige uma comunicação eficiente e segura entre microsserviços, além de observabilidade robusta para garantir confiabilidade. Sobre o time técnico da empresa, atualmente possui sete desenvolvedores, um time relativamente pequeno, incluindo o CTO que atua fortemente nas implementações de alto níveis técnicos, como a migração feita da antiga arquitetura para a orientada à serviços.

Essa complexidade tecnológica torna a Keeps um caso real para o estudo dos desafios e estratégias de comunicação entre microsserviços, tema central deste trabalho. Os próximos tópicos apresentam a arquitetura técnica adotada e as principais dificuldades observadas em sua camada de comunicação.

4.2 ARQUITETURA DA KEEPS

A empresa Keeps fez a transição de uma arquitetura baseada em SOA para uma arquitetura de microsserviços. Essa mudança visou aumentar a escalabilidade, a resiliência e a eficiência operacional da plataforma, permitindo que a empresa acompanhe seu crescimento e atenda a demandas cada vez mais dinâmicas. Outro objetivo do CTO, líder dessa migração, foi tornar a arquitetura da Keeps mais independente de serviços da AWS, facilitando uma possível troca no futuro do provedor de computação em nuvem.

Na estrutura atual da Keeps, os microsserviços são responsáveis por processar diferentes funcionalidades, no entanto alguns ainda possuem um agregado muito grande de funções, como o microsserviço Konquest Server, antiga aplicação principal. Apesar de operar na infraestrutura de microsserviços, o serviço Konquest é um “macro-serviço”, dado que centraliza uma alta concentração de regras de negócio e responsabilidades (gestão de cursos, matrículas, gamificação). Esta característica é um dos gargalos que impedem a escalabilidade, justificando as estratégias de desacoplamento.

Os microsserviços da Keeps, em grande maioria, possuem dois padrões de comunicações: síncrona, com chamadas diretas por meio de REST, e assíncrona, por meio de *brokers* de mensagens como RabbitMQ.

A comunicação assíncrona entre os microsserviços, interação entre microsserviços autônomos (*service-to-service*) da Keeps ocorre por meio do RabbitMQ. Essa arquitetura conecta as aplicações web (APIs) aos microsserviços de *backend* (*workers*), que executam tarefas de processamento, às vezes intensivo, e não bloqueantes.

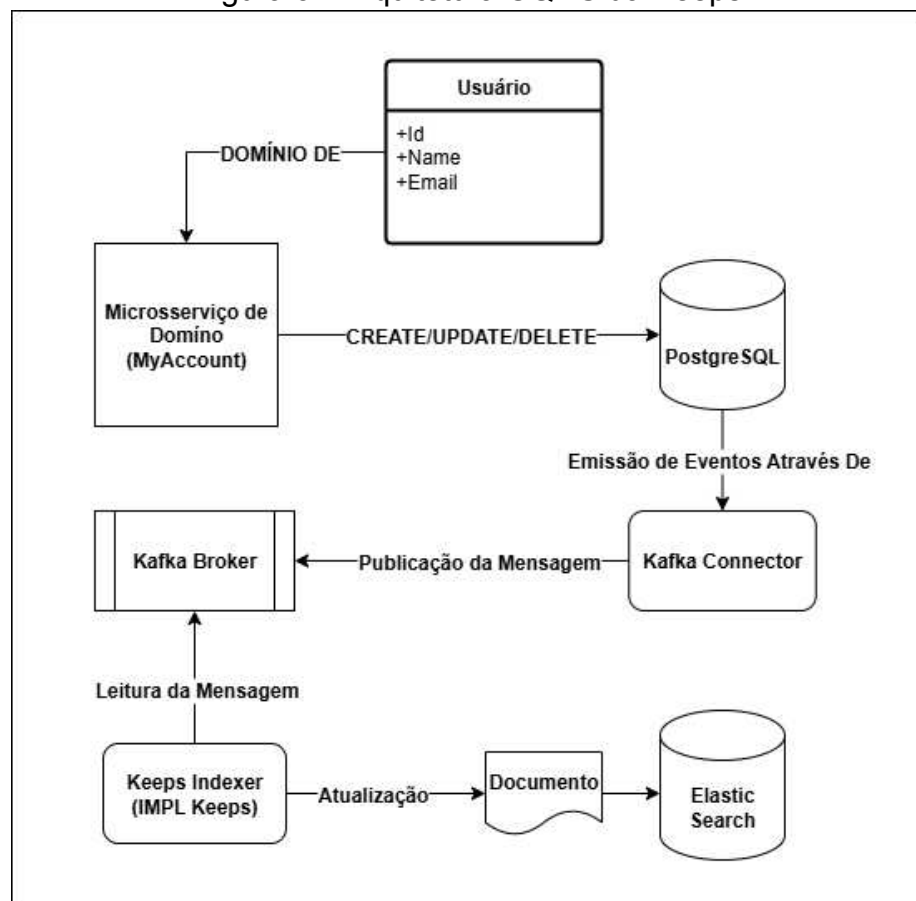
4.2.1 CQRS e Replicação de Dados

Além das comunicações síncronas e assíncronas tradicionais, o Apache Kafka é utilizado em conjunto com o Kafka Connect em alguns processos específicos da arquitetura da Keeps. Seu principal uso está na implementação do padrão CQRS (*Command Query Responsibility Segregation*), no qual as alterações ocorridas nas bases de dados relacionais dos microsserviços, responsáveis por

diferentes domínios de negócio, são convertidas em eventos de *streaming* processados em tempo real.

Esses eventos alimentam uma base não relacional, implementada com Elasticsearch, que atua como repositório otimizado para consultas e agregações de dados. Essa estratégia permite desacoplar as operações de leitura das bases relacionais principais (PostgreSQL Servers), reduzindo a sobrecarga de consultas complexas e aumentando a escalabilidade da camada de leitura. A Figura 3 a seguir resume de forma simples a arquitetura atual de CQRS da Keeps.

Figura 3 - Arquitetura CQRS da Keeps



Fonte: elaborado pelo autor.

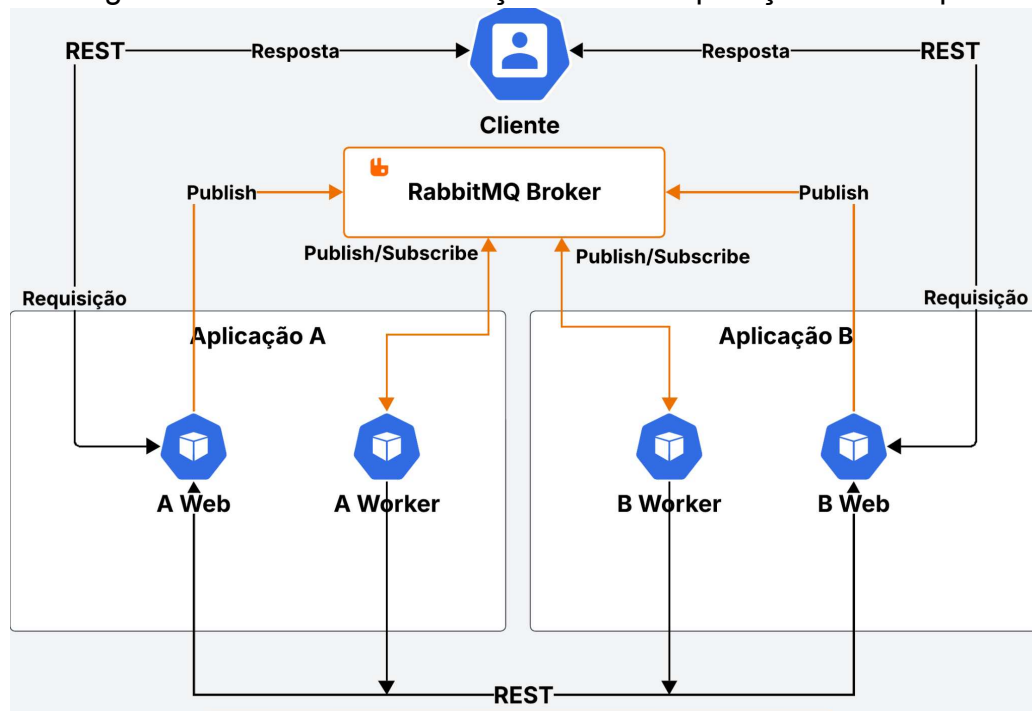
Sobre o Keeps Indexer, que não foi descrito nos microserviços anteriormente, ele também é um microserviço que não atende diretamente nenhum cliente, mas realiza um papel importante de processar esses eventos apresentados em mensagens nos tópicos do *Kafka Broker* atualizando as réplicas dos dados das entidades.

Em outros cenários, o *Keeps Indexer*, não é utilizado no processo de replicação desses dados, apenas o uso *Kafka Connector* já é suficiente para atualizar os dados na base que possui as réplicas quando não é necessário a normalização ou formatação entre os dados de origem e de réplica. Utiliza-se dessa arquitetura também para aplicar a replicação seletiva de dados entre os microsserviços.

4.2.2 Comunicação Entre Microsserviços API e *Workers*

Conforme ilustrado na Figura 4, o fluxo de comunicação da Keeps combina chamadas síncronas e assíncronas dentre as aplicações que fornecem API (*Web*) e seus respectivos *Workers*. Neste padrão de comunicação muito utilizado na Keeps, o cliente realiza uma requisição REST para a aplicação *web* (Aplicação A), que pode acionar um *worker* da própria aplicação ou de outra aplicação (Application B) por meio de chamadas intermediadas pelo RabbitMQ. Após o processamento, o *worker* responde ao RabbitMQ, que realiza a confirmação (ACK) da mensagem consumida. Quando é necessário recuperar dados diretamente de outro microsserviço, a comunicação ocorre de forma síncrona via REST.

Figura 4 - Fluxo de comunicação entre as aplicações da Keeps



Fonte: elaborado pelo autor.

4.2.3 Kubernetes e Arquitetura Geral de Comunicação

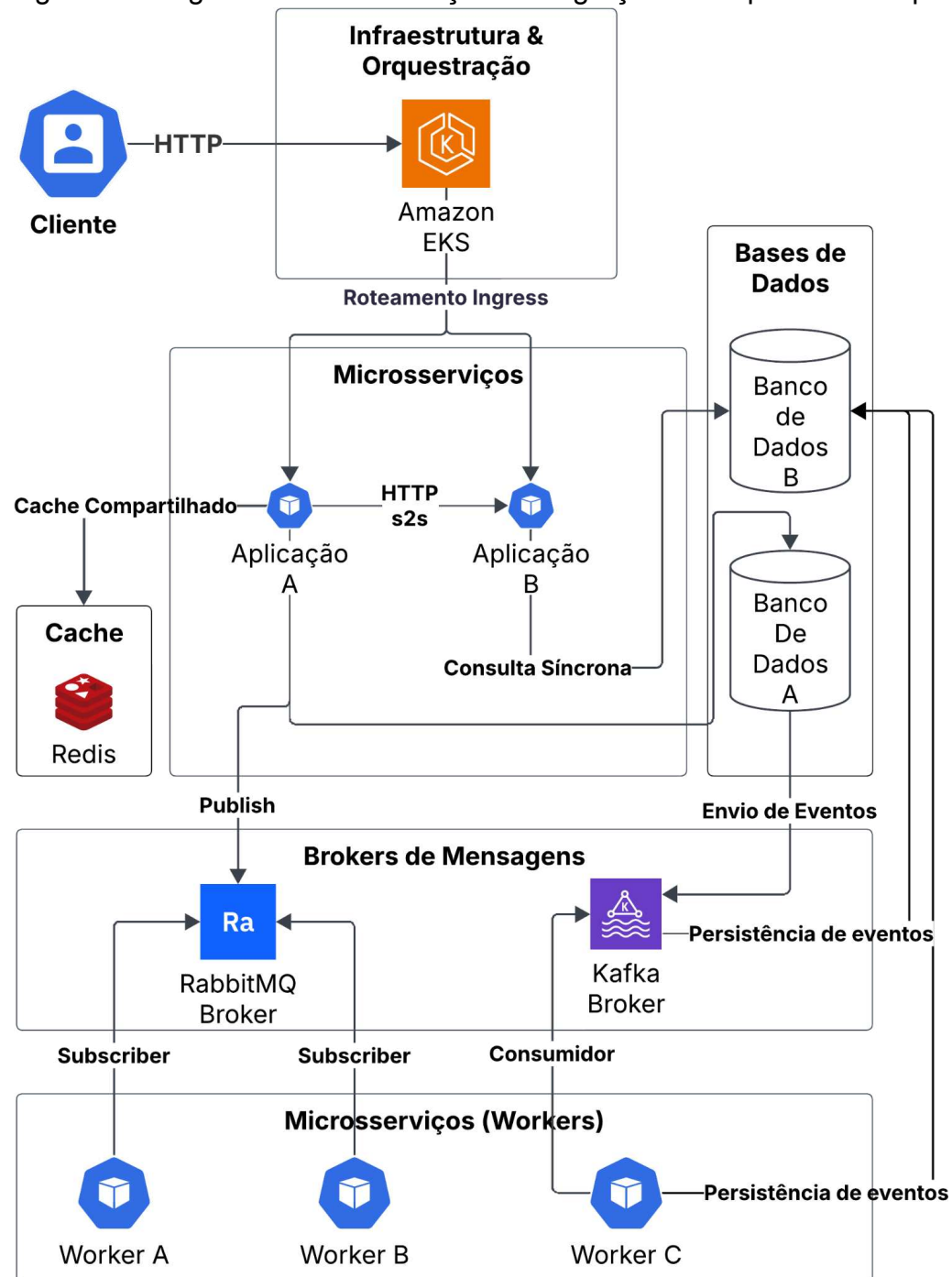
O Amazon EKS, aliado ao Kubernetes, desempenha papel essencial na coordenação e escalonamento dos microsserviços da Keeps. O Amazon EKS é responsável por orquestrar os microsserviços, gerenciar implantações, escalar instâncias e garantir balanceamento de carga.

O *Kubernetes* utiliza *Ingress Controllers* para expor serviços via *HTTP* e segmentar o tráfego por domínio e caminho. O roteamento inteligente das requisições é fundamental para manter a escalabilidade e a organização da comunicação entre as APIs públicas e os serviços internos.

Complementando a arquitetura, a Keeps utiliza Redis como mecanismo de cache em alguns pontos estratégicos da comunicação, especialmente para dados transitórios. Contudo, sua aplicação ainda é limitada. A adoção de cache defensivo, conforme recomendado por Aksakalli et al. (2021), poderia reduzir a dependência entre serviços e minimizar falhas em cascata.

Uma visão mais completa da arquitetura e das integrações pode ser observada no diagrama da Figura 5.

Figura 5 - Diagrama de comunicação e integração da arquitetura Keeps.



Fonte: elaborado pelo autor.

Na Figura 5, é possível visualizar de forma integrada como a comunicação entre os microserviços da Keeps é estruturada. O cliente acessa os serviços da empresa por meio de requisições HTTP, que são inicialmente processadas pelo *Amazon EKS*, responsável pelo balanceamento de carga, roteamento via *Ingress Controller* e escalonamento automático das aplicações.

As APIs Web (Aplicação A e Aplicação B) realizam a comunicação síncrona entre si (*HTTP Service-to-Service*) e também podem realizar chamadas diretas a bancos de dados para operações que exigem consistência imediata (Consulta Síncrona).

Além das operações síncronas, as aplicações publicam mensagens no *RabbitMQ* para execução de processos assíncronos. Essas mensagens são consumidas por *workers* dedicados, que operam desacoplados no tempo e retornam os resultados ou confirmações de processamento (*ACK*).

O diagrama evidencia a coexistência de comunicação síncrona e assíncrona, bem como a integração de camadas de cache e mensageria. Entretanto, observam-se pontos de fragilidade relacionados à latência, dependência entre serviços e falta de padronização nos mecanismos de tolerância a falhas.

4.3 RELATÓRIO DE PROBLEMAS

A análise técnica da arquitetura da Keeps, conduzida a partir da observação direta dos ambientes produtivos, relatórios internos e registros operacionais, permitiu identificar um conjunto de dificuldades recorrentes que comprometem a eficiência, a confiabilidade e a resiliência da comunicação entre microsserviços.

Esses desafios estão relacionados tanto à maturidade da arquitetura quanto às práticas de observabilidade, padronização e outros aspectos. A seguir, apresentam-se as principais fragilidades encontradas e suas implicações no funcionamento da plataforma.

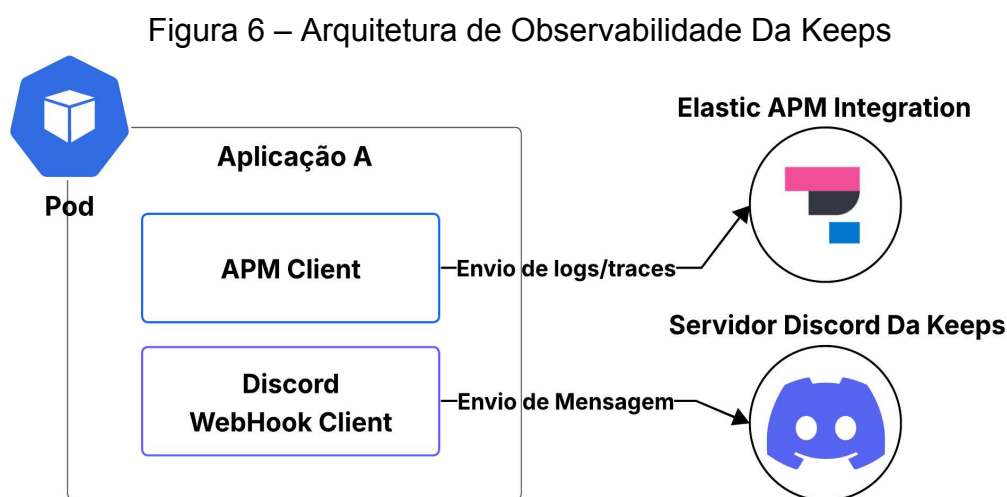
4.3.1 Observabilidade limitada

A infraestrutura de monitoramento da Keeps apresenta baixa cobertura e fragmentação entre as aplicações, o que dificulta a visibilidade de ponta a ponta das transações. Atualmente, não existe um sistema centralizado de *logs*, cada aplicação adota um modelo próprio de registro de eventos, o que impede a correlação de falhas entre serviços.

O tracing distribuído, realizado parcialmente via Elastic APM, não abrange todos os microsserviços, resultando em lacunas na identificação de gargalos e falhas intermitentes. Essa limitação impacta diretamente o tempo de resposta às

ocorrências: incidentes que poderiam ser diagnosticados automaticamente exigem depuração manual e investigação extensa de *logs* isolados.

Além disso, o monitoramento de métricas é inconsistente entre os serviços. Aplicações mais antigas, desenvolvidas em Python, seguem um modelo de coleta baseado em WebHooks do Discord, enquanto aplicações mais recentes, escritas em Node.js, utilizam integrações parciais com o Elastic APM. Essa disparidade tecnológica compromete a padronização dos dados observados, tornando inviável a análise comparativa de desempenho entre módulos. A figura abaixo demonstra o exposto neste parágrafo.



Fonte: elaborada pelo autor

Essa situação reflete uma baixa maturidade de observabilidade, conforme definida por Khan et al. (2023), que destacam a importância da integração entre *logs*, métricas e *traces* para garantir diagnósticos precisos e reduzir o MTTR.

4.3.2 Alta dependência entre aplicações

Apesar da adoção do modelo de microsserviços, a Keeps ainda apresenta forte acoplamento temporal entre algumas de suas aplicações principais, como Konquest, MyAccount e Analytics. Esses módulos frequentemente realizam chamadas síncronas diretas via HTTP para buscar informações em outros serviços, o que gera dependências cruzadas e reduz a autonomia operacional de cada componente.

Em situações de sobrecarga, lentidão ou indisponibilidade de um serviço intermediário, os demais ficam bloqueados, formando uma cadeia de falhas em cascata. Essa fragilidade impacta diretamente a experiência do usuário final, uma vez que requisições simples podem ter seus tempos de resposta multiplicados.

Além disso, não há mecanismos robustos de fallback, circuit breaker ou cache defensivo implementados para absorver essas falhas temporárias, o que agrava a vulnerabilidade da arquitetura.

Conforme Fowler (2017) e Aksakalli et al. (2021), o acoplamento temporal é um dos principais fatores que reduzem a resiliência de sistemas distribuídos, pois compromete o isolamento dos serviços e impede a degradação controlada em casos de erro.

4.3.3 Perda de mensagens e baixa confiabilidade

Durante a análise dos *logs* do RabbitMQ, foram identificados casos de perda de mensagens relacionadas a falhas de entrega e reprocessamento. O principal fator associado a esse problema é a ausência de DLQ em diversas filas de comunicação. Quando uma mensagem falha repetidamente durante o consumo, ela é descartada sem registro adicional, impossibilitando a auditoria e o posterior reprocessamento.

Essa limitação impacta fluxos críticos, como emissão de certificados, sincronização de matrículas e notificações automáticas, que dependem da entrega garantida de mensagens entre serviços. Além disso, há inconsistência na configuração das filas entre diferentes ambientes (produção, homologação e desenvolvimento), o que gera comportamentos divergentes e dificulta a reprodução de erros em testes.

Segundo Aksakalli et al. (2021), a confiabilidade da comunicação assíncrona depende não apenas da escolha do broker, mas também da implementação de padrões como reentrega controlada, DLQ e mecanismos de persistência de mensagens, que garantem recuperação consistente após falhas. Na Keeps, a ausência desses mecanismos representa uma vulnerabilidade crítica em fluxos de alta sensibilidade.

4.3.4 Gargalos de desempenho

Durante períodos de pico de uso, especialmente em lançamentos de programas corporativos ou em grandes acessos simultâneos, observou-se um aumento significativo da latência em chamadas síncronas e redução da vazão dos *workers*. Esses gargalos estão associados a três fatores principais:

1. Configurações inadequadas das filas do RabbitMQ, que não estão otimizadas para alto volume de mensagens;
2. Versões desatualizadas das bibliotecas clientes utilizadas em aplicações Node.js e Python, gerando incompatibilidades de conexão e maior tempo de *handshake*;
3. Ausência de métricas detalhadas de fila, impossibilitando o ajuste dinâmico da capacidade de consumo e o dimensionamento (*auto-scaling*) de *workers* conforme a demanda.

Esses fatores combinados provocam lentidão na propagação de mensagens e bloqueios temporários nas filas de maior tráfego, comprometendo diretamente o tempo de resposta do sistema e a escalabilidade operacional.

Rodrigues, Rito Silva e Avritzer (2025) apontam que o dimensionamento automático de serviços distribuídos exige a integração entre métricas de uso e políticas de escalonamento, algo ainda inexistente na atual configuração da Keeps.

4.3.5 Falhas não detectadas em serviços ativos

Outro problema recorrente ocorre em situações em que microsserviços permanecem em execução, mas incapazes de processar novas requisições. Essas falhas “silenciosas” normalmente estão associadas a bloqueios internos de processamento, *loops* não tratados (a inicialização do serviço presa por alguma lógica de inicialização) ou saturação de filas internas.

Como não há monitoramento de métricas em tempo real, o sistema de observabilidade não detecta automaticamente a degradação do serviço, ele permanece “ativo” aos olhos do Kubernetes, mas sem efetivamente responder às requisições. Esse tipo de falha é particularmente crítico porque não aciona mecanismos de recuperação automática, como o reinício de pods pelo Kubernetes, o que prolonga a indisponibilidade.

Além disso, a falta de alertas relacionados impede que a equipe de operações identifique a causa raiz de forma proativa, resultando em resposta reativa a incidentes.

4.3.6 Falta de governança e padronização

A Keeps ainda enfrenta lacunas na governança técnica da arquitetura, especialmente no que se refere à padronização de processos e políticas de atualização. Não há um modelo unificado de versionamento de bibliotecas, nem de controle de compatibilidade entre serviços, o que gera inconsistências nos ambientes e aumenta o esforço de manutenção.

Além disso, a ausência de documentação técnica consolidada e de políticas de revisão de código padronizadas reduz a capacidade de governança e controle sobre a evolução da arquitetura.

Esses problemas de governança são comuns em empresas em processo de amadurecimento arquitetural, principalmente empresas com times pequenos como a Keeps. Segundo Rodrigues, Rito Silva e Avritzer (2025), a ausência de mecanismos de padronização e observabilidade contínua tende a comprometer tanto a qualidade técnica quanto a previsibilidade do sistema em produção.

4.3.7 Síntese

De forma geral, as dificuldades identificadas na Keeps, mapeadas por meio da observação participante e da análise dos incidentes técnicos recentes, revelam um cenário de adaptação entre a adoção conceitual dos microsserviços e a consolidação de práticas maduras de comunicação e observabilidade. A arquitetura já incorpora elementos fundamentais, como mensageria assíncrona, escalonamento automático e segregação de responsabilidades. Contudo, os registros analisados evidenciam que o sistema ainda carece de mecanismos de resiliência, governança e padronização, indispensáveis para alcançar a estabilidade plena em larga escala.

Essas fragilidades, fundamentadas na realidade operacional da empresa, servem como base para as propostas de melhoria apresentadas no próximo capítulo. Tais propostas fortalecem a comunicação entre microsserviços e aprimoram a capacidade de observação, diagnóstico e recuperação da plataforma.

5 MELHORIAS E RESULTADOS

A partir da análise dos desafios apresentados na comunicação entre microsserviços da Keeps e das evidências coletadas nos relatórios técnicos, foram identificadas e implementadas diversas ações com foco em observabilidade, desempenho, resiliência e simplificação arquitetural. Essas iniciativas buscaram aprimorar a eficiência da comunicação entre serviços, reduzir o acoplamento temporal e aumentar a confiabilidade operacional do ecossistema distribuído da empresa.

As intervenções realizadas foram fundamentadas em práticas consolidadas na literatura especializada (AKSAKALLI et al., 2021; KHAN et al., 2023; RODRIGUES; RITO SILVA; AVRITZER, 2025), que destacam a importância da padronização de protocolos, do monitoramento contínuo e da adoção de mecanismos de isolamento e observabilidade como pilares de arquiteturas escaláveis. A aplicação desses conceitos no ambiente da Keeps permitiu não apenas a otimização técnica da comunicação entre microsserviços, mas também o amadurecimento das práticas de engenharia e governança de software.

Este capítulo apresenta de forma integrada os principais avanços obtidos a partir dessas iniciativas. Inicialmente, são descritas as melhorias efetivamente implementadas na arquitetura da Keeps, que resultaram em ganhos concretos de desempenho e estabilidade. Em seguida, são apresentados os resultados experimentais provenientes dos testes comparativos de vazão, latência e utilização de CPU, realizados em ambiente controlado para validar o impacto das mudanças. Por fim, são discutidas as propostas de melhorias futuras, voltadas à evolução contínua da arquitetura e ao fortalecimento da resiliência sistêmica da plataforma.

5.1 MELHORIAS IMPLEMENTADAS NA KEEPS

As melhorias implementadas na arquitetura da Keeps concentraram-se na resolução dos principais problemas identificados no capítulo anterior, com foco em fortalecer a observabilidade, reduzir o acoplamento entre microsserviços e aprimorar o desempenho operacional da plataforma. As intervenções realizadas foram agrupadas em dois eixos principais: o primeiro voltado à consolidação da observabilidade e padronização do monitoramento distribuído, e o segundo à

padronização e otimização da comunicação interna, abrangendo o uso de gRPC, cache, consistência eventual, mensageria e API Gateway.

5.1.1 Melhorias na observabilidade

Uma das principais etapas do processo de aprimoramento da arquitetura da Keeps concentrou-se no fortalecimento da observabilidade, elemento essencial em ambientes baseados em microsserviços. A ausência de uma estrutura consolidada de monitoramento era apontada internamente como um dos principais entraves operacionais, comprometendo a capacidade da equipe de desenvolvimento em diagnosticar falhas e compreender o comportamento sistêmico das aplicações em produção. Conforme discutido anteriormente na Seção 2.4.6, a observabilidade é considerada um dos pilares fundamentais para garantir confiabilidade, rastreabilidade e eficiência em sistemas distribuídos, permitindo a correlação entre métricas, *logs* e *traces* (KHAN et al., 2023; AKSAKALLI et al., 2021).

Diante desse cenário, foi conduzido um processo de reestruturação da camada de observabilidade, com o objetivo de padronizar a coleta de métricas, unificar *logs* de execução e estabelecer rastreamento distribuído entre todos os microsserviços. A iniciativa buscou não apenas detectar anomalias e prever falhas de forma proativa, mas também fornecer uma base de dados consistente para a análise de desempenho e o aprimoramento contínuo da arquitetura.

Como etapa central dessa reestruturação, foi configurado o *OpenTelemetry Operator* no *cluster Kubernetes* da Keeps. Essa tecnologia foi escolhida por possibilitar instrumentação automática das aplicações por meio de recursos declarativos (*Instrumentation CRDs*), reduzindo a necessidade de agentes manuais e garantindo padronização em toda a infraestrutura.

O *OpenTelemetry Operator* foi configurado para aplicar instrumentação automática nos pods de aplicações *Node.js*, utilizando objetos declarativos que direcionam a exportação de métricas, *logs* e *traces* para um *OpenTelemetry Collector* centralizado, que atua como ponto de agregação de telemetria, responsável por receber, normalizar e encaminhar os dados para o *Elastic APM*, ferramenta escolhida para análise e visualização dentro da pilha *Elastic Stack* (*Elasticsearch* e *Kibana*). Essa integração substituiu a necessidade de múltiplas

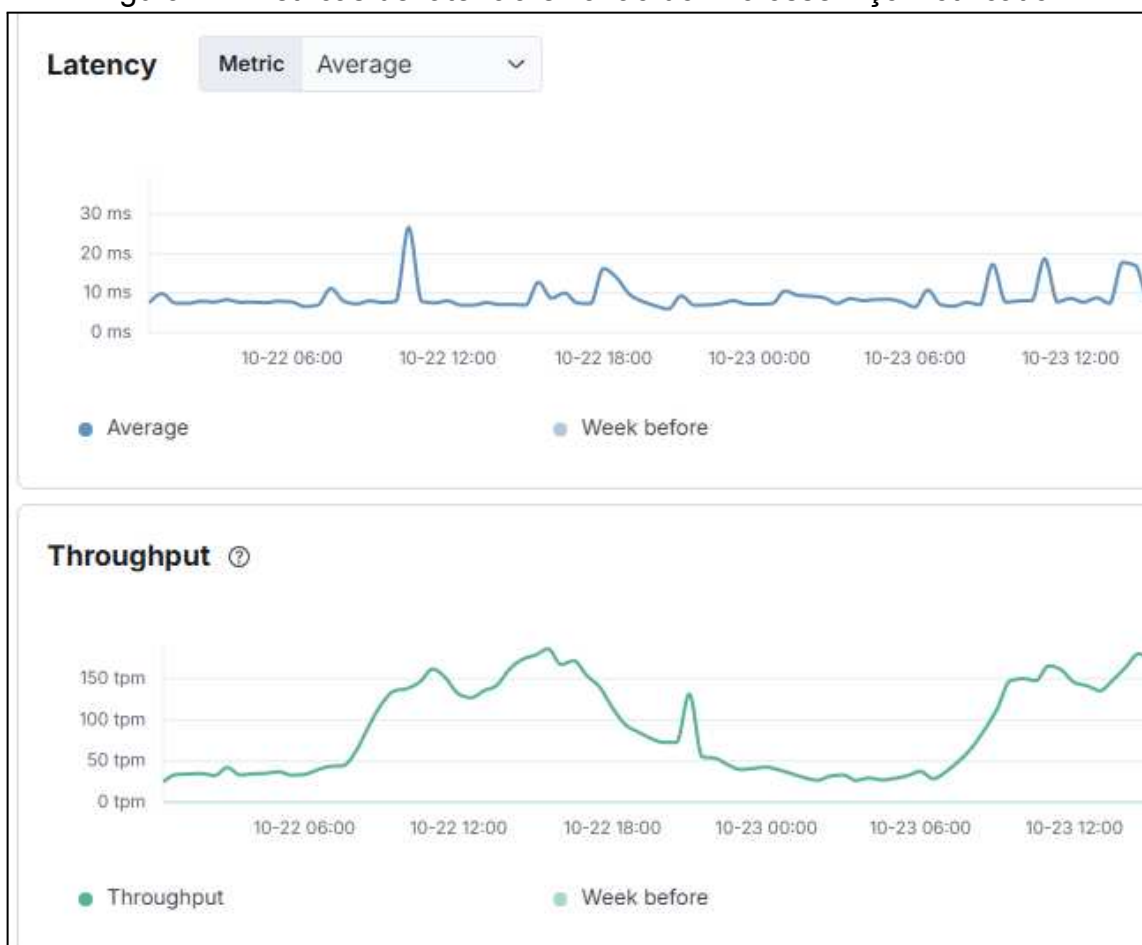
ferramentas, concentrando todo o ciclo de observabilidade, da coleta à análise, em uma única solução de código aberto, escalável e interoperável.

Além da instrumentação automática no nível do cluster, foi desenvolvida uma biblioteca interna de instrumentação para os microsserviços Node.js, com o propósito de inserir automaticamente identificadores de rastreamento (*trace_id*, *span_id*) nos registros de *log* e estruturar as saídas no formato *JSON*, conforme as especificações oficiais do *OpenTelemetry*. Essa padronização é essencial para o rastreamento distribuído, pois viabiliza a correlação entre eventos gerados por diferentes serviços, permitindo uma visão completa do ciclo de vida das requisições (AKSAKALLI et al., 2021; RODRIGUES; RITO SILVA; AVRITZER, 2025).

Apesar dos avanços, a implementação de *logs* estruturados também apresentou desafios. Um dos principais obstáculos foi a configuração correta das variáveis de ambiente em cada microsserviço para garantir a correta identificação do serviço nos *traces*. Para resolver essa questão, foram implementadas práticas de validação que garantem a consistência da configuração em todos os microsserviços. Além disso, foi necessário implementar o *graceful shutdown* para garantir que todos os dados de telemetria fossem coletados e enviados adequadamente antes do encerramento dos serviços, evitando perda de informações importantes.

A integração entre a biblioteca de instrumentação e o *OpenTelemetry Collector* formou um *pipeline* unificado de observabilidade, capaz de correlacionar métricas de infraestrutura, *logs* de aplicação e *traces* de comunicação. Essa arquitetura trouxe ganhos operacionais, com expectativas de redução no tempo médio de detecção de falhas, *Mean Time to Detect (MTTD)*, e maior precisão na identificação de gargalos entre serviços. A centralização dos dados no *Elastic Kibana Observability* possibilita a criação de painéis analíticos personalizados, com indicadores em tempo real sobre latência de chamadas, taxa de erro e utilização de recursos, conforme demonstrado nas figuras a seguir.

Figura 7 – Métricas de latência e vazão do microserviço Notification



Fonte: elaborada pelo autor

A Figura 7 apresenta as métricas consolidadas de latência média e transações por minuto (*throughput*) do microserviço Notification, capturadas diretamente do painel do *Elastic Observability*. O gráfico superior ilustra a variação temporal da latência média das requisições, que se manteve predominantemente abaixo de 20 ms com picos pontuais de maior processamento. O gráfico inferior exhibe a taxa de *throughput*, permitindo correlacionar períodos de maior carga com eventuais variações na latência. Essa visualização é fundamental para o acompanhamento de padrões de uso e para a identificação de gargalos ocasionais de desempenho sob carga variável.

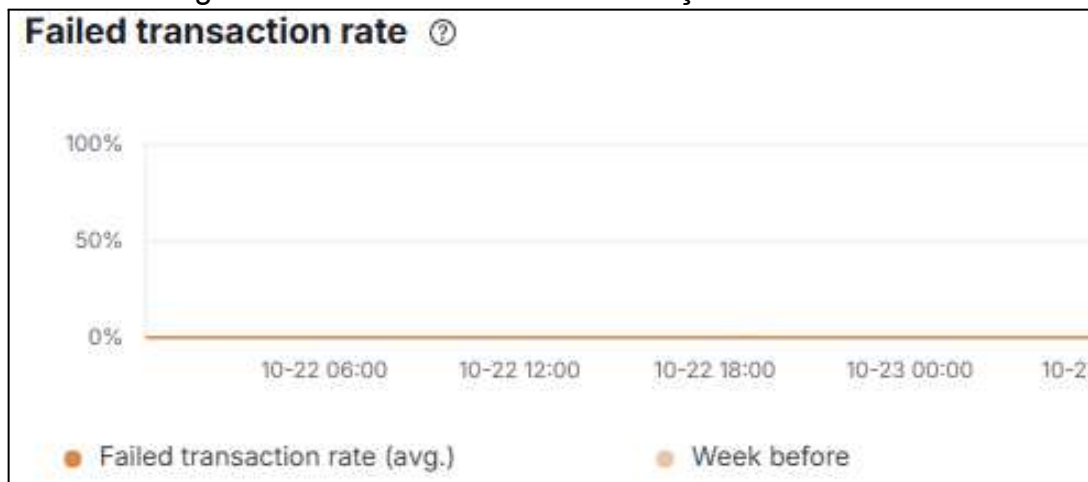
Figura 8 – Desempenho por rota e tempo médio de resposta do Notification

Transactions			
Search transactions by name			
Name ↕	Latency (avg.) ↕	Throughput ↕	Failed transacti... ↕
GET /bell-notifications	14 ms	43.4 tpm	0.0%
POST /emails/messages	411 ms	0.5 tpm	0%
OPTIONS unknown route	0.5 ms	44.3 tpm	0%
POST /aws/sns	4.8 ms	2.4 tpm	0%

Fonte: elaborada pelo autor

A Figura 8 detalha as principais rotas monitoradas no microsserviço Notification, incluindo as operações *GET /bell-notifications* e *POST /emails/messages*. Observa-se que a rota de envio de e-mails apresenta o maior tempo médio de resposta ($\approx 411\text{ ms}$), o que indica um ponto de atenção para otimização futura. Já a rota de notificações *push (/bell-notifications)* mantém latência média de 14 ms e alto volume de transações. Uma observação, tal *endpoint* está desativado, apenas enviando resposta com o *status 200 OK*, que significa sucesso, mas não retorna nenhum dado ou realiza nenhum processo. Enfim, esses dados reforçam a importância da observabilidade distribuída para identificar *endpoints* críticos e direcionar intervenções pontuais de otimização.

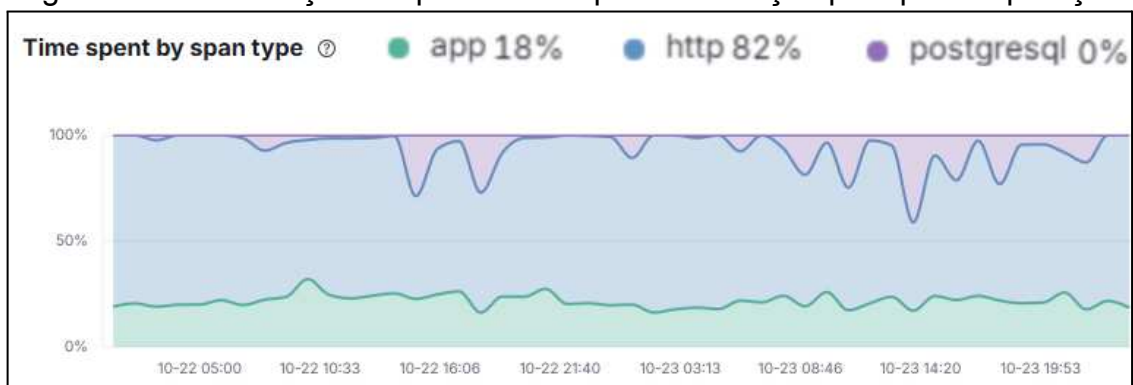
Figura 9 – Taxa de falhas em transações monitoradas



Fonte: elaborada pelo autor

A Figura 9 apresenta a taxa média de falhas das transações do microsserviço Notification durante o mesmo intervalo de observação. A ausência de variações significativas (0 %) demonstra que não ocorreram falhas no período analisado. O acompanhamento contínuo desse indicador é essencial para detectar regressões em *deploys* e avaliar a robustez do sistema sob diferentes volumes de carga.

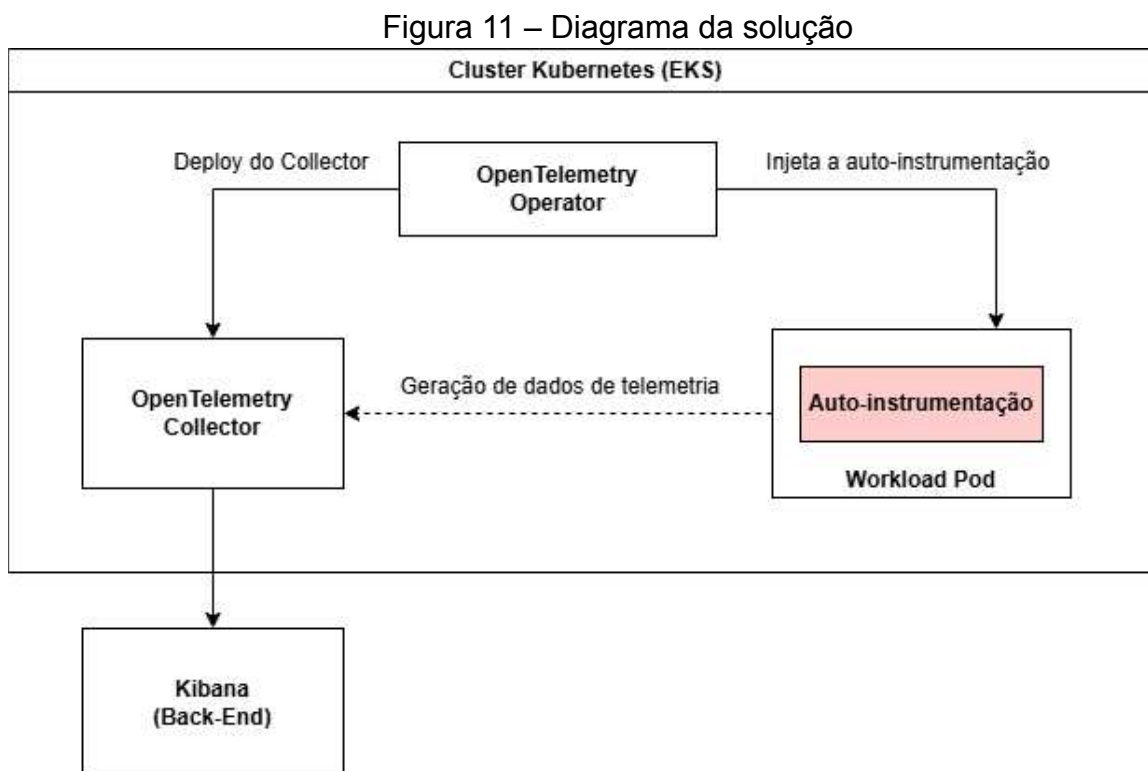
Figura 10 – Distribuição temporal do tempo de execução por tipo de operação



Fonte: elaborada pelo autor

A Figura 10 evidencia a distribuição do tempo de execução. Nota-se que aproximadamente 82% do tempo é classificado como '*http*', referente à latência de comunicação e aguardo de resposta entre serviços. Embora a métrica direta seja de rede, a análise aprofundada sugere que essa espera é majoritariamente derivada do tempo de processamento de consultas no banco de dados do serviço chamado.

Além disso, a Figura 11 apresenta o diagrama simplificado da solução final de observabilidade adotada, detalhando o fluxo de instrumentação dentro do cluster EKS da Keeps.



Fonte: elaborada pelo autor

Neste diagrama, observa-se o papel de cada componente na arquitetura de monitoramento. O *OpenTelemetry Operator* é responsável por injetar automaticamente a instrumentação nas aplicações em execução (*Workload Pods*), utilizando recursos declarativos (*Instrumentation CRDs*). Esses componentes instrumentados geram dados de telemetria (métricas, *logs* e *traces*) que são coletados pelo *OpenTelemetry Collector*, o qual centraliza e normaliza as informações antes de encaminhá-las ao *Elastic APM/Kibana*, onde são armazenadas e analisadas.

Em síntese, essa solução permitiu padronizar a coleta de dados de telemetria, consolidar a visibilidade distribuída e aprimorar o monitoramento contínuo, resultando em um ambiente mais previsível, escalável e resiliente. Essas mudanças não apenas facilitaram a análise de falhas e a correlação de eventos, mas também proporcionaram uma base sólida para o crescimento e a escalabilidade da plataforma. A integração entre *OpenTelemetry*, *Pino* e *Kibana Observability* gerou

ganhos operacionais expressivos e estabeleceu uma infraestrutura de observabilidade robusta e eficiente, fundamental para o monitoramento contínuo da aplicação.

5.1.2 Padronização da comunicação interna com gRPC

A comunicação entre microsserviços é um dos elementos mais críticos em arquiteturas distribuídas, pois influencia diretamente a eficiência, a resiliência e a manutenibilidade do sistema. Com base nos problemas identificados no capítulo anterior, especialmente o alto acoplamento entre serviços e a inconsistência nos formatos de integração, foram implementadas melhorias estruturais na camada de comunicação *service-to-service*, com o objetivo de padronizar protocolos, reduzir dependências diretas e aprimorar a confiabilidade das interações internas.

Inicialmente, observou-se que a maioria dos microsserviços da Keeps comunicava-se por meio de REST APIs síncronas, implementadas de forma heterogênea e sem padrões consolidados de versionamento, tratamento de erros ou controle de dependências. Essa abordagem, embora simples, introduz sobrecarga de rede, acoplamento temporal e maior latência entre chamadas internas, problemas amplamente descritos na literatura (NISWAR et al., 2024; AKSAKALLI et al., 2021).

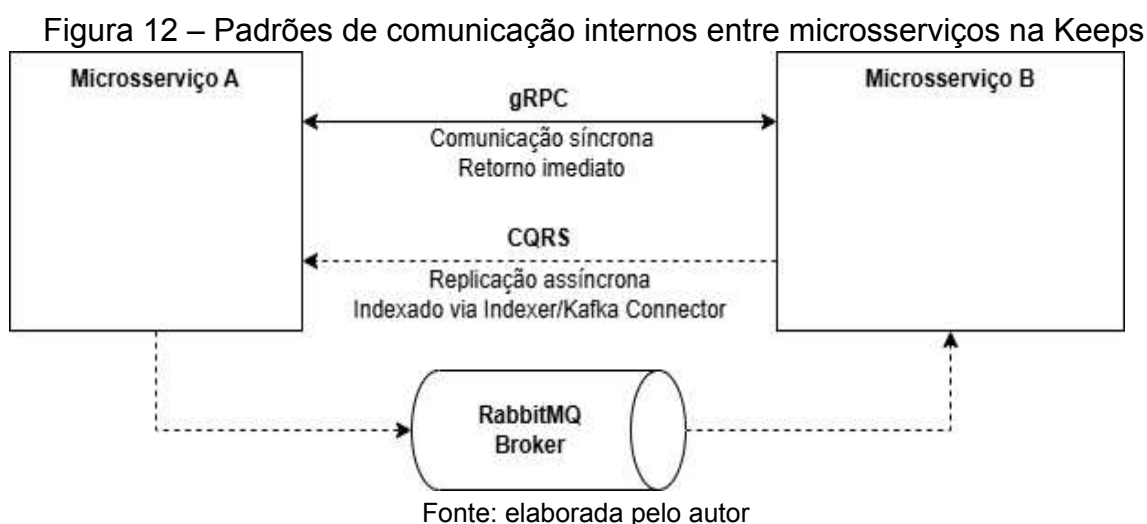
Com base nas conclusões apresentadas por Niswar et al. (2024), que demonstram o melhor desempenho do gRPC em cenários de alta concorrência, foi definida a adoção do gRPC como padrão para comunicações internas síncronas entre microsserviços. A decisão visou otimizar a serialização e o transporte de dados por meio do uso do *Protocol Buffers (Protobuf)* sobre HTTP/2, garantindo maior eficiência, menor latência e padronização contratual. Os arquivos *.proto* foram centralizados em um repositório compartilhado, mantido no *Nexus*, de modo a assegurar a consistência das definições e facilitar a integração entre serviços desenvolvidos em linguagens distintas, como *Python* e *Node.js*.

O ambiente de execução dos microsserviços permanece unificado em um único cluster Kubernetes, no qual o balanceamento de carga do *gRPC* é realizado de forma nativa via *ClusterIP*, sem a necessidade de *proxies* intermediários. Para manter o isolamento de contexto entre clientes, o cabeçalho *x-client* é enviado via *metadata*, garantindo a separação de *tenants* sem a necessidade de autenticação direta entre os serviços. Essa decisão foi tomada considerando que as camadas

externas de exposição, compostas por APIs REST e, em alguns casos, por um API Gateway, já implementam mecanismos de autenticação e controle de acesso.

Em cenários onde há dependência frequente e sensível de dados entre microsserviços, foi adotado o padrão CQRS para minimizar acoplamentos e otimizar a disponibilidade de dados. Nesse modelo, os serviços consumidores mantêm uma replicação assíncrona das informações necessárias, evitando consultas diretas e reduzindo a dependência temporal entre componentes. Essa replicação é realizada por meio de um módulo utilitário interno, que integra o *Indexer* e o *Kafka Connector*, responsáveis por processar e sincronizar eventos de atualização de dados entre as bases de origem e destino. O uso desse padrão já fazia parte da arquitetura da Keeps, mas, a partir desta reestruturação, foi formalizado como prática padrão de análise: sempre que um serviço A necessitar de dados de outro serviço B, deve-se avaliar se a comunicação será estabelecida via *gRPC* (para interações pontuais e síncronas) ou *CQRS* (para consultas recorrentes e de leitura).

Em situações nas quais o serviço solicitante apenas precisa notificar ou iniciar um processo em outro microsserviço, sem a necessidade de resposta imediata, foi definido que a comunicação deve ocorrer de forma assíncrona por mensageria. Esse modelo será detalhado posteriormente. É possível ver o esquema dos padrões definidos na Figura 12.



A padronização da comunicação interna proporcionou avanços na coerência técnica entre microsserviços, reduziu a duplicação de esforços de integração e tornou o ecossistema mais previsível e sustentável a longo prazo. Além disso, a

consolidação dos protocolos e padrões de interação entre serviços estabeleceu uma base sólida para futuras evoluções da arquitetura, facilitando tanto a manutenção quanto a expansão da plataforma da Keeps.

5.1.3 Utilização de *cache*

Conforme identificado no capítulo anterior, a Keeps já utilizava o *Redis* em alguns cenários específicos, principalmente para armazenar resultados temporários de consultas repetitivas entre microsserviços. Entretanto, durante a análise das comunicações *service-to-service*, foram detectados diversos outros pontos de dependência em que a ausência de *cache* contribuía para o aumento da latência e para a degradação da experiência do usuário final.

De acordo com Aksakalli et al. (2021), o uso adequado de *cache* em microsserviços é considerado uma das estratégias mais eficientes para mitigar gargalos de comunicação, especialmente em interações síncronas e de leitura intensiva. Com base nesse entendimento, definiu-se como padrão de desenvolvimento a análise prévia da possibilidade de utilização de *cache* em novas implementações, principalmente em cenários nos quais a aplicação dependa de dados externos para processar requisições de forma síncrona e com retorno imediato. Essa prática tem como objetivo não apenas reduzir a latência, mas também aumentar a resiliência das aplicações diante de falhas transitórias de rede ou indisponibilidade de serviços dependentes. Um dos casos mais relevantes de aplicação dessa estratégia ocorreu na comunicação entre os microsserviços e o MyAccount.

O MyAccount é o microsserviço responsável pela gestão de perfis e permissões de usuários. Antes da implementação do *cache*, a disponibilidade do MyAccount impactava diretamente todos os microsserviços dependentes, pois qualquer falha ou lentidão no serviço central resultava em indisponibilidade generalizada. Para mitigar esse risco, foi implementado um mecanismo de *cache* de permissões no *Redis*. Esse *cache* armazena temporariamente as informações de perfis e autorizações dos usuários, permitindo que as aplicações continuem operando mesmo em períodos de instabilidade do MyAccount.

O funcionamento desse mecanismo foi projetado com foco em segurança e consistência: sempre que ocorre uma alteração nas permissões de um usuário,

como a revogação de acesso, o cache correspondente é automaticamente invalidado, garantindo que o sistema reflita a nova configuração em tempo real. O TTL (*Time to Live* - um tempo de vida para a chave, fazendo com que no fim desse tempo, a chave expire) foi definido de forma balanceada, assegurando que os dados armazenados sejam utilizados apenas durante o período necessário para a normalização do serviço central, evitando acessos indevidos. Essa implementação reduziu o impacto de falhas temporárias do *MyAccount* e aumentou a estabilidade dos microsserviços dependentes, sem comprometer a integridade das políticas de segurança.

Além dessa aplicação específica, o uso de cache foi ampliado para outros contextos considerados críticos dentro da plataforma. Um dos avanços mais relevantes foi a implementação de *cache* para dados de configuração e metadados de conteúdo. Também foram adotados mecanismos de *cache* para respostas de APIs de terceiros diminuindo a dependência de requisições sucessivas a sistemas fora do domínio da Keeps.

Essas práticas consolidaram um padrão arquitetural de *caching* distribuído dentro do ecossistema Keeps, equilibrando performance, resiliência e segurança. A padronização no uso do *Redis* e a definição de políticas claras de invalidação permitiram maior previsibilidade e controle sobre o ciclo de vida dos dados em *cache*, além de reduzir a sobrecarga nas camadas de comunicação síncrona e nos bancos de dados subjacentes.

Em síntese, a formalização do uso de *cache* como parte integrante do processo de desenvolvimento contribuiu significativamente para a redução da latência nas comunicações internas, a melhoria da disponibilidade dos serviços dependentes e a resiliência operacional da arquitetura da Keeps.

5.1.4 Adoção de consistência eventual

Seguindo com as melhorias implementadas, o conceito de consistência eventual foi introduzido em pontos estratégicos da arquitetura, um aspecto que não estava previsto anteriormente nas interações entre microsserviços que utilizam replicações de dados via CQRS. A Keeps não adota transações distribuídas complexas nem padrões como SAGA (ou outros como Orquestração de Transações, conforme discutido na Seção 2.4.6), uma vez que tais soluções não são necessárias

no contexto da plataforma. Em vez disso, foram adotadas implementações simples e pontuais, desenvolvidas manualmente, eficazes para resolver problemas específicos de comunicação.

Um exemplo concreto dessa melhoria ocorreu no microserviço Notification, responsável por enviar notificações aos usuários via email, WhatsApp e outros canais. O serviço MyAccount, responsável pelo gerenciamento de autenticação e permissões dos usuários, replica os dados dos usuários para o Notification via CQRS. No entanto, devido ao tempo necessário para a replicação desses dados, em alguns casos as notificações não eram enviadas. Isso acontecia porque o microserviço Notification dependia dos dados atualizados do usuário para processar a notificação. Esse atraso na replicação impedia o envio das notificações e afetava outras funções críticas, como a autenticação de login, que também dependia desses dados replicados.

Para resolver esse problema, foi implementada a consistência eventual, permitindo que o microserviço Notification fosse menos dependente da disponibilidade imediata dos dados de MyAccount. O código foi alterado para tratar a ausência temporária de dados do usuário. Em vez de falhar completamente ou esperar pela replicação, o processo de envio da notificação foi reconfigurado para ser agendado até que os dados estivessem disponíveis na base local. Assim que a replicação for completada, a notificação será enviada automaticamente. O código a seguir ilustra como as notificações são tratadas de forma assíncrona e como o reprocessamento é agendado quando há falha na replicação de dados.

Figura 13 – Código para notificação de usuário com consistência eventual

```

async create(message: CreateEmailMessageDto, workspace: Workspace,
recipient?: Recipient): Promise<Message> {
  let recipientNotFoundEmail: string;
  if (!recipient) {
    // Consistência eventual: armazena email mesmo sem dados do
usuário replicados
    recipientNotFoundEmail = message.receiverMail;
  }
  const createdMessage = this.repository.create({
    ...message,
    recipient: recipient,
    recipientNotFoundEmail, // Permite processamento posterior
quando dados estiverem disponíveis
    workspace,
  });
  // . . .
}

@Cron(CronExpression.EVERY_MINUTE)
async retrySendMessageWithRecipientNotFoundCron() {
  const messages = await this.repository.find({
    where: {
      status: MESSAGE_STATUS.RECIPIENT_NOT_FOUND,
      createdAt: MoreThan(someTimeBefore),
    },
  });

  messages?.forEach((message) => {
    // Tenta reenviar quando dados do usuário estiverem disponíveis
    this.retrySendMessageWithRecipientNotFound(message);
  });
}

```

Fonte: elaborada pelo autor

Quando o sistema não consegue encontrar os dados do destinatário de uma notificação, a mensagem (*message*) é armazenada com a informação de que o destinatário não foi encontrado (*recipientNotFoundEmail*). Esse armazenamento permite que a mensagem seja processada posteriormente, assim que as informações do destinatário forem replicadas e atualizadas.

O processo de reenvio das mensagens é feito por meio de uma tarefa agendada (@Cron), que é executada a cada minuto. Nessa tarefa, o sistema busca as mensagens não enviadas devido à falta de dados do destinatário e tenta processá-las novamente para envio, assim que os dados do usuário se tornem disponíveis. Essa abordagem garante que as notificações sejam enviadas de forma resiliente e que a integridade das operações seja mantida, mesmo em cenários de dados inconsistentes temporários.

A introdução de consistência eventual trouxe ganhos na eficiência e na experiência do usuário, resolvendo o problema da não entrega das notificações e eliminando falhas de comunicação que comprometem a continuidade do sistema. Esse mecanismo de consistência eventual mostrou-se altamente eficaz em resolver problemas pontuais, sem a necessidade de transações complexas entre microsserviços.

Além disso, a flexibilidade dessa abordagem foi essencial para manter a resiliência e disponibilidade do sistema, sem aumentar a complexidade do modelo. A implementação de soluções manuais e simples, quando aplicadas de forma estratégica, garantiu a alta performance da arquitetura, sem comprometer a integridade ou escalabilidade do ambiente da Keeps.

5.1.5 Reestruturação e ampliação do uso de mensageria

Na Keeps, a mensageria já era utilizada de forma pontual por meio do RabbitMQ, porém, durante o processo de reestruturação da arquitetura, observou-se a necessidade de expandir e aprimorar sua aplicação, consolidando-a como o principal meio de comunicação assíncrona entre serviços.

O RabbitMQ, que implementa o protocolo AMQP, foi mantido como o message broker padrão, mas passou a contar com um conjunto de melhorias estruturais e operacionais. Dentre as principais intervenções realizadas destacam-se:

- 1) Criação de filas dedicadas por domínio, organizadas conforme os bounded contexts dos serviços;
- 2) Implementação de DLQ para o armazenamento temporário de mensagens com falha de entrega;

- 3) Adoção de políticas de *retries automáticos* e *backoff exponencial*, reduzindo a perda de mensagens em cenários de instabilidade;
- 4) Integração de métricas e *logs* centralizados via OpenTelemetry, viabilizando a rastreabilidade de cada mensagem durante seu ciclo de vida.

Relembra-se que eram frequentes os casos de perda de mensagens e duplicações em períodos de pico de carga, ocasionadas pela ausência de confirmações manuais (*acknowledgments*) e configurações inadequadas nas filas e nos *workers*. Após a reestruturação, a arquitetura de mensageria passou a apresentar melhor desempenho e resiliência. O uso de DLQ permitiu identificar padrões recorrentes de falha e possibilitou o reprocessamento de eventos sem perda de dados.

Em termos de desempenho, espera-se que a ampliação do uso da mensageria resulte em uma redução significativa das falhas de comunicação e na diminuição do tempo médio de reprocessamento de eventos. Esses resultados esperados são baseados no que Niswar et al. (2024) observaram em seu estudo experimental: sistemas que adotam protocolos otimizados e estratégias de comunicação assíncrona obtêm melhor aproveitamento de CPU e menor latência em ambientes de alta concorrência.

5.1.6 Implementação do *API Gateway*

Sobre a implementação da *API Gateway* na Keeps, o objetivo é centralizar o tráfego de comunicação entre os microsserviços internos e os clientes externos. Neste novo modelo, o *API Gateway* funciona como um portal de entrada, permitindo que todas as requisições dos usuários sejam gerenciadas de forma centralizada. Com isso, o *gateway* assume responsabilidades cruciais, como o controle de autorização, validando as permissões para o acesso aos serviços, além de permitir a escala de segurança necessária para ambientes distribuídos.

No momento, o único microsserviço operacional junto ao *API Gateway* é o Image Generator, um serviço responsável pela geração de imagens por meio de Inteligência Artificial (IA). Esse padrão de comunicação foi adotado para garantir consistência e interoperabilidade entre os microsserviços, simplificando o desenvolvimento e a manutenção dos sistemas de integração.

A arquitetura implementada foi projetada para ser escalável, ou seja, a integração do API Gateway com o Image Generator não se limita a esse microserviço. O plano futuro é que todos os novos microserviços da Keeps adotem essa mesma abordagem de comunicação, utilizando o *API Gateway* para gerenciar o tráfego externo e interações internas. Essa implementação facilita a segurança, visto que o *gateway* centraliza os pontos de controle de acesso e autenticação, permitindo um monitoramento mais eficiente e reforçando as políticas de segurança.

O fortalecimento da segurança é um dos principais objetivos dessa implementação. Como o *API Gateway* centraliza as chamadas externas, ele se torna o ponto único de validação e autorização de acesso, facilitando a aplicação de políticas de segurança, como controle de tráfego e prevenção contra ataques distribuídos, como *Distributed Denial of Service* (DDoS). Este modelo de comunicação integrado com o *API Gateway* não só fortalece a segurança, mas também melhora a escalabilidade e flexibilidade da arquitetura.

5.2 RESULTADOS EXPERIMENTAIS E MÉTRICAS DE DESEMPENHO

Com o objetivo de validar o impacto técnico das melhorias implementadas na arquitetura da Keeps, foram conduzidos testes experimentais em ambiente controlado, comparando o comportamento das versões anteriores e posteriores às otimizações descritas no capítulo anterior. As métricas avaliadas refletem três dimensões centrais de desempenho: tempo de resposta, *throughput* e utilização de CPU. Permitindo mensurar de forma objetiva os ganhos decorrentes das mudanças arquitetônicas aplicadas.

Os experimentos foram realizados em ambiente local de desenvolvimento, configurado para simular o comportamento real dos microserviços em execução no cluster Kubernetes da Keeps. Todas as execuções foram monitoradas através do OpenTelemetry, com análise e visualização dos resultados no Elastic Kibana Observability, garantindo a correlação temporal entre métricas de aplicação e de infraestrutura.

Embora diversos fluxos da plataforma tenham sido afetados pelas melhorias de comunicação e cache, o primeiro cenário experimental selecionado concentra-se no processamento de tracking do microserviço Konquest, pela sua relevância

operacional e alto volume de requisições. Outros fluxos impactados serão incluídos posteriormente, utilizando a mesma metodologia descrita nesta seção.

5.2.1 Metodologia dos testes

A metodologia experimental foi estruturada para avaliar o impacto direto das melhorias implementadas nos fluxos de comunicação e processamento dos microserviços, considerando o comportamento real das aplicações em operação. As análises concentraram-se em transações específicas que sofreram alterações significativas decorrentes das melhorias descritas no Capítulo 5.1, como a substituição de comunicações internas HTTP REST por gRPC, a introdução de cache defensivo em Redis e a otimização de dependências entre serviços.

Cada transação foi avaliada em duas fases experimentais distintas:

- 1) Fase 1 – Baseline: execução da versão anterior do código, sem as melhorias.
- 2) Fase 2 – Versão aprimorada: execução da mesma transação após a implementação das melhorias.

A geração de carga foi realizada utilizando a ferramenta k6, simulando requisições HTTP REST equivalentes às chamadas feitas por usuários reais. Essa abordagem foi escolhida por representar de forma mais fidedigna o ponto de entrada do sistema comparado ao um *script* manual, tentando manter o comportamento natural de chamadas cliente-servidor, enquanto as comunicações internas operam de forma transparente na camada de backend.

O ambiente experimental foi composto por contêineres Docker configurados com 2 vCPUs e 1 GB de memória por serviço. Todas as execuções foram realizadas com 50 usuários virtuais simultâneos ao longo de dez minutos, com sessenta segundos iniciais de aquecimento descartados da análise. As execuções foram repetidas três vezes por fase, permitindo calcular médias e reduzir interferências pontuais.

Essa metodologia assegurou que as diferenças observadas entre as fases refletissem exclusivamente o impacto das melhorias, sem influência de variações externas do ambiente.

5.2.2 Cenário experimental: *Tracking* no Konquest

O cenário experimental selecionado corresponde ao processamento de *tracking* de consumo de conteúdo, executado pelo microsserviço Konquest. Esse fluxo foi escolhido por representar uma das transações de maior volume e importância dentro do ecossistema da Keeps, pois é responsável por registrar o tempo de consumo real dos usuários em cada conteúdo de aprendizagem, servindo de base para o cálculo da nota final e dos indicadores de desempenho individual e corporativo. O endpoint avaliado é o seguinte: *PATCH konquest/users/learn-activities/{id}*.

Na versão anterior da aplicação das melhorias, esse processamento realizava a validação de permissões do usuário diretamente no serviço MyAccount por meio de uma chamada síncrona *HTTP REST*, sem mecanismos de contingência. Essa abordagem gerava acoplamento temporal e aumento de latência em períodos de alta carga ou instabilidade no MyAccount.

Na versão aprimorada, a comunicação interna entre o Konquest e o MyAccount foi reestruturada para utilizar o protocolo *gRPC*, com *Protocol Buffers* como formato de serialização binária. Além disso, foi introduzido um *cache* defensivo Redis responsável por armazenar localmente as permissões dos usuários e validar as requisições subsequentes sem depender da disponibilidade imediata do serviço remoto. Essa combinação reduziu significativamente o tempo de validação das permissões e eliminou a necessidade de re-conexões redundantes.

O teste foi realizado submetendo o endpoint a um volume constante de requisições simuladas, refletindo o padrão real de uso da aplicação. Durante os testes, foram monitoradas as métricas de tempo médio de resposta (p50), percentis superiores (p95 e p99), *throughput* médio e utilização de CPU por contêiner, permitindo uma visão abrangente da performance do fluxo antes e depois das otimizações internas.

5.2.3 Execução e coleta de métricas

A execução dos experimentos foi conduzida de forma controlada, respeitando o fluxo real de autenticação e processamento adotado pela arquitetura da Keeps. O endpoint avaliado, responsável pelo registro do consumo de conteúdo no

microserviço Konquest, requer autenticação por meio de *token* JWT emitido pelo *Keycloak*, bem como o envio do cabeçalho *x-client*, que identifica a *workspace* do usuário solicitante.

Para garantir que os testes reproduzissem fielmente as condições de produção, foi implementado um processo completo de autenticação e autorização. Antes de cada rodada de testes, o *token* JWT foi obtido diretamente do *Keycloak* através do endpoint padrão de autenticação OAuth2, conforme o exemplo abaixo na Figura 14. Com os *tokens* já carregados, foram mapeados os usuários criados exclusivamente para os testes e executado o *script* utilizando o comando da Figura 15, passando como parâmetro os valores já citados. Na Figura 16 está representado um script conceitual e resumido, o *script* original possui um quantidade maior de código para implementar o cenário de testes e coleta.

Figura 14 – Comando para carregar o *Token*

```
curl --location --request POST
'https://keycloak.keepsdev.com/realms/keeps/protocol/openid-connect
/token' \
  --header 'Content-Type: application/x-www-form-urlencoded' \
  --data-urlencode 'grant_type=password' \
  --data-urlencode 'client_id=keeps-client' \
  --data-urlencode 'username=test_user' \
  --data-urlencode 'password=test_password'
```

Fonte: elaborada pelo autor

Figura 15 – Comando do k6 utilizado

```
k6 run --vus 50 --duration 10m
--summary-trend-stats="avg,p(50),p(95),p(99)"
konquest_tracking_rest.js
```

Fonte: elaborada pelo autor

Figura 16 – Script resumido em javascript *konquest_tracking_rest.js*

```
import http from 'k6/http';
import { sleep } from 'k6';

export default function () {
  const body = JSON.stringify({ progress: 5 });
  const res =
    http.patch(`${__ENV.API_URL}/konquest/users/learn-activities/4a86ce
81-7892-4eaf-816a-7ba1648fc6b4`, body, {
      headers: { 'Content-Type': 'application/json' },
    });
  sleep(0.5);
}
```

Fonte: elaborada pelo autor

Além do *script*, a CPU foi monitorada constantemente. Os dados de latência e *throughput* foram exportados a partir dos relatórios gerados pelo k6, permitindo a criação das tabelas comparativas apresentadas na próxima seção.

Os indicadores analisados foram definidos com base na leitura dos trabalhos relacionados:

- 1) Tempo de resposta (ms): medido pelos percentis p50, p95 e p99, representando a mediana, as caudas superiores e os piores tempos observados.
- 2) *Throughput* (req/s): taxa média de requisições processadas por segundo durante o período estável de carga.
- 3) Pico de CPU (%): pico de uso dos contêineres durante as janelas de teste, representando a eficiência computacional sob carga.

5.2.4 Resultados obtidos

Os resultados obtidos permitem avaliar quantitativamente o impacto das melhorias implementadas na comunicação entre os microsserviços Konquest e MyAccount. A Tabela 2 apresenta as métricas coletadas durante as execuções da versão original (v1) e da versão otimizada (v2), ambas submetidas à mesma carga de usuários e condições de ambiente.

Tabela 2 – Métricas de desempenho da API Konquest *Tracking*

Versão	p50 (ms)	p95 (ms)	p99 (ms)	CPU Média (%)	Throughput (req/s)
v1	680	972	1791	93	78
v2	337	758	974	84	167

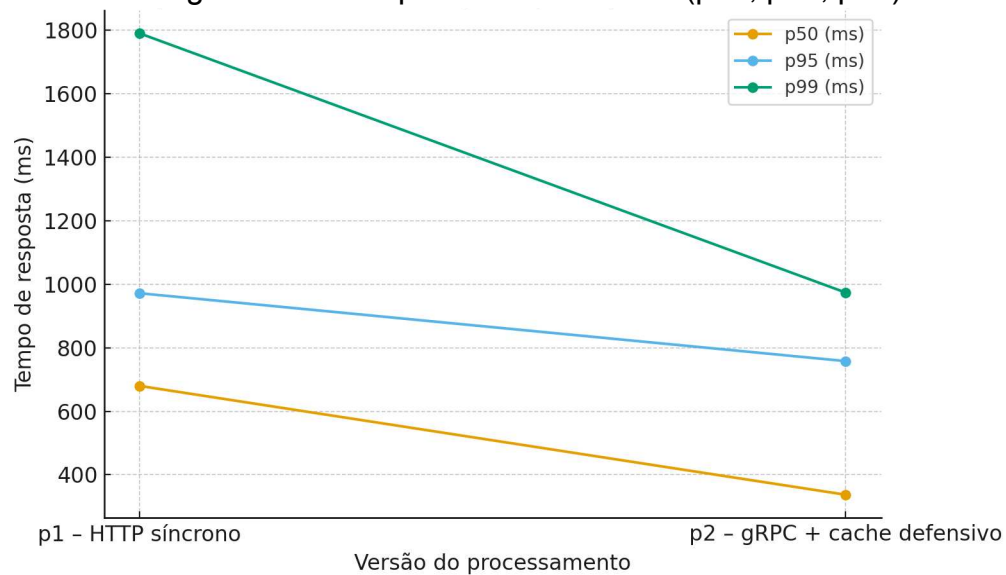
Fonte: elaborado pelo autor.

Na versão v1, a latência média (p50) manteve-se em torno de 680 ms, com picos de até 1791 ms no percentil p99. O consumo médio de CPU foi de 93%, e o *throughput* estabilizou em aproximadamente 78 requisições por segundo, refletindo as limitações inerentes à comunicação síncrona usando HTTP, caracterizada por maior overhead de serialização e ausência de cache defensivo.

Já na versão otimizada (v2), observou-se uma redução expressiva em todos os tempos de resposta, com a mediana (p50) reduzida para 337 ms e o percentil p99 para 974 ms. Além de uma diminuição no consumo médio de CPU para 84%. Paralelamente, o *throughput* aumentou para 167 requisições por segundo, demonstrando uma melhora significativa na capacidade de processamento sob as mesmas condições experimentais.

A Figura 17 apresenta a comparação dos tempos de resposta entre as versões v1 e v2, evidenciando a redução significativa nas latências médias e superiores após a substituição do protocolo HTTP por gRPC e a implementação do cache Redis defensivo. Uma expressiva melhora de desempenho na versão otimizada (v2), com redução de aproximadamente 50% na latência média (p50) e 45% no percentil p99, refletindo maior estabilidade e previsibilidade sob carga.

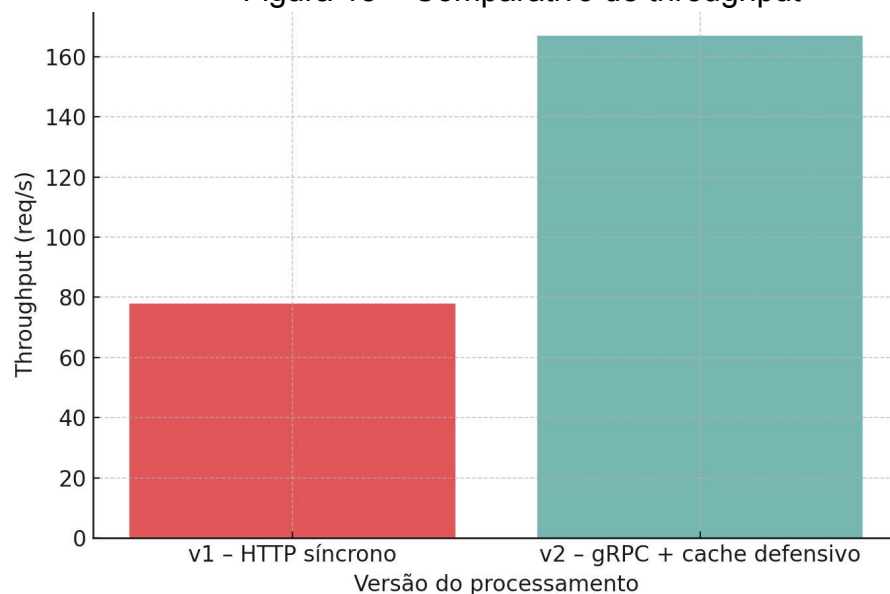
Figura 17 – Comparativo de latência (p50, p95, p99)



Fonte: elaborada pelo autor

A Figura 18 ilustra o aumento de vazão na mesma comparação, evidenciando o ganho de eficiência decorrente da adoção de protocolos binários e da eliminação de dependências temporais entre serviços.

Figura 18 – Comparativo de throughput



Fonte: elaborada pelo autor

A análise conjunta dos indicadores mostra que a arquitetura revisada (v2) apresenta respostas mais rápidas e estáveis, com melhor aproveitamento de recursos computacionais. O aumento do *throughput* (de 78 para 167 req/s) confirma o ganho de capacidade de processamento, enquanto a redução de CPU (de 93%

para 84%) comprova a eficiência do novo modelo de comunicação e o efeito positivo do cache defensivo.

Esses resultados validam empiricamente as melhorias descritas no Capítulo 5.1 e reforçam a importância da padronização de comunicação e da observabilidade distribuída como pilares para o aprimoramento contínuo da arquitetura de microsserviços.

5.2.5 Discussão dos resultados

A análise integrada das métricas de desempenho evidencia de forma clara os ganhos proporcionados pelas melhorias implementadas na comunicação entre os microsserviços Konquest e MyAccount. Os resultados empíricos confirmam que a substituição do protocolo HTTP síncrono por gRPC e a introdução do cache defensivo com Redis impactaram positivamente todos os indicadores observados (latência, *throughput* e utilização de CPU).

Conforme demonstrado na Figura 17, a latência média (p50) apresentou redução de 680 ms para 337 ms, representando uma melhoria de aproximadamente 50%. As latências superiores também foram significativamente reduzidas, com o percentil p95 caindo de 972 ms para 758 ms e o p99 de 1791 ms para 974 ms, reduções de 22% e 45,6%, respectivamente. Essa diminuição nas caudas da distribuição confirma que o sistema passou a responder de forma mais previsível sob condições de carga elevada, característica essencial para aplicações distribuídas com múltiplos microsserviços interdependentes.

Observou-se também um aumento de 114% no *throughput* médio, que passou de 78 para 167 requisições por segundo. Esse ganho está diretamente associado à adoção do protocolo gRPC, reduzindo o *overhead* de serialização e tornando a comunicação entre serviços mais eficiente. Além disso, a inclusão do cache Redis defensivo reduziu a frequência de chamadas diretas ao serviço MyAccount, diminuindo a dependência temporal e eliminando gargalos causados pela latência de rede. Essa estratégia resultou em maior estabilidade transacional e em menor variação de tempo de resposta sob picos de carga.

A redução do consumo médio de CPU também se mostrou um indicador consistente do aumento da eficiência computacional. O uso médio caiu de 93% na versão v1 para 84% na versão v2, representando uma diminuição aproximada de

9,7%. Esse resultado decorre da menor carga de processamento para serialização e desserialização de mensagens e da eliminação de chamadas redundantes, proporcionada pelo cache local. O sistema passou a processar mais requisições simultâneas com menor utilização de recursos, demonstrando não apenas ganho de desempenho, mas também uma melhoria concreta de escalabilidade.

Esses resultados validam a hipótese central deste trabalho, segundo a qual o conjunto de melhorias implementadas (gRPC e Redis) contribui de forma significativa para aumentar a eficiência e a resiliência da arquitetura da Keeps. O modelo revisado se mostrou capaz de reduzir simultaneamente a latência e o consumo de CPU, enquanto aumentou a previsibilidade e a estabilidade do comportamento sistêmico, aspectos fundamentais para a manutenção de níveis adequados de desempenho em arquiteturas distribuídas.

Por fim, é importante ressaltar que os testes foram conduzidos em ambiente local controlado, com rede de baixa latência e recursos limitados a dois vCPUs por serviço. Assim, embora os resultados reflitam ganhos reais de eficiência, espera-se que, em ambiente produtivo com maior variabilidade de rede e volume de requisições, o impacto positivo das otimizações seja ainda mais perceptível, especialmente na redução de erros por timeouts e na estabilização de picos de carga.

Em síntese, a versão v2 apresentou melhorias substanciais em todos os aspectos analisados, menor tempo de resposta, maior *throughput* e menor custo computacional, confirmando a eficácia das mudanças arquitetônicas propostas e consolidando as bases para a evolução contínua da plataforma Keeps.

6 CONCLUSÃO

6.1 SÍNTESE DOS RESULTADOS

Os resultados obtidos neste trabalho evidenciam que a otimização da comunicação entre microsserviços na Keeps contribuiu para o aprimoramento da performance, da estabilidade e da escalabilidade da plataforma. As melhorias implementadas, em especial a migração parcial das comunicações para gRPC e o uso de cache defensivo com Redis, demonstraram reduções na latência média e nos percentis superiores, além de ganhos de throughput e eficiência no uso de CPU.

Esses avanços consolidam o gRPC como protocolo preferencial para cenários de comunicação síncrona no contexto da Keeps, devido à sua natureza binária, suporte a HTTP/2 e baixo overhead de serialização. Em paralelo, a utilização do RabbitMQ como solução de mensageria assíncrona reforçou o potencial de desacoplamento entre serviços, garantindo maior resiliência e continuidade operacional em situações de carga elevada ou falhas temporárias.

Os testes realizados demonstraram empiricamente que a padronização da comunicação interna e o uso de mecanismos de cache não apenas reduzem o tempo de resposta das aplicações, mas também diminuem o custo computacional e ampliam a previsibilidade do comportamento sistêmico.

6.2 LIMITAÇÕES DO ESTUDO

Embora os resultados tenham sido satisfatórios, é importante reconhecer algumas limitações inerentes ao escopo e à metodologia adotados neste estudo. Os experimentos foram conduzidos em ambiente local controlado, e não em ambiente de produção, o que implica uma rede de baixa latência e ausência de variabilidade externa, como picos de tráfego reais e interferências de infraestrutura.

Além disso, nem todos os microsserviços da Keeps foram convertidos para gRPC, de modo que parte da comunicação ainda ocorre via REST. Isso limita a avaliação de impacto em escala global e restringe a análise a cenários específicos de integração entre Konquest e MyAccount.

Outro ponto de atenção é que a instrumentação via OpenTelemetry ainda exige ajustes manuais em determinados módulos legados, impedindo a coleta completa de métricas de rastreamento distribuído. Por fim, o estudo concentrou-se

prioritariamente em métricas de desempenho (tempo de resposta, *throughput* e CPU), não abordando de forma detalhada custos computacionais de longo prazo ou consumo de recursos em nuvem, fatores relevantes para estudos futuros sobre eficiência econômica da arquitetura.

6.3 TRABALHOS FUTUROS

Os resultados apresentados nesta pesquisa demonstraram o impacto positivo da padronização de comunicação e da introdução de mecanismos de cache na arquitetura de microsserviços da Keeps. No entanto, as melhorias implementadas representam apenas uma etapa intermediária no processo de evolução tecnológica da plataforma. Com base nas observações obtidas durante os experimentos e nas práticas consolidadas da literatura sobre sistemas distribuídos, foram identificadas oportunidades adicionais de aprimoramento, que compõem o plano de propostas futuras.

A primeira proposta consiste na automação dos testes de desempenho e regressão. Os experimentos realizados nesta pesquisa foram conduzidos manualmente, com scripts e configurações específicas. A integração dessas execuções em *pipelines* de CI/CD (*Continuous Integration / Continuous Deployment*), utilizando ferramentas como *GitHub Actions* ou *GitLab CI*, permitiria validar continuamente o impacto de novas alterações no desempenho e na estabilidade do sistema. Essa automação garantiria que futuras otimizações de código não introduzem regressões ou aumentos inesperados de latência.

Outra proposta de avanço consiste na ampliação da observabilidade distribuída já estruturada com o OpenTelemetry e Elastic APM. Atualmente, as métricas e *traces* são coletadas e armazenadas de forma centralizada, mas sem o uso de alertas automatizados e correlação entre *logs* e eventos de infraestrutura. A implementação de dashboards dinâmicos, com alertas baseados em anomalias e SLOs (*Service Level Objectives*), permitiria detectar degradações de desempenho em tempo real e responder prontamente a incidentes.

Outro avanço relevante a ser explorado é o estudo da implementação de um Service Mesh. O uso de soluções como Istio, Linkerd ou Kuma possibilitaria padronizar políticas de tráfego, autenticação mTLS, retries, circuit breakers e rate limiting sem necessidade de intervenção direta no código das aplicações. Essa

abordagem elevaria o nível de segurança e observabilidade, ao mesmo tempo em que simplificaria a gestão operacional da comunicação interna. Além disso, o Service Mesh facilitaria a adoção de práticas avançadas como canary releases, aprimorando o controle sobre o ciclo de vida das releases e reduzindo o risco de indisponibilidade em atualizações. Estudos futuros também podem abordar a decomposição do serviço *Konquest* em microsserviços menores e mais granulares para reduzir sua responsabilidade monolítica.

Essas propostas complementam o conjunto de melhorias abordadas nesta pesquisa e apontam o caminho para uma arquitetura de microsserviços mais resiliente, observável, escalável e segura, alinhada às práticas de engenharia modernas e às necessidades de crescimento da plataforma Keeps. Sua implementação progressiva permitirá consolidar a base técnica construída nesta etapa, ampliando os ganhos de desempenho e estabelecendo um ambiente mais preparado para suportar novas funcionalidades e volumes crescentes de uso.

6.4 CONSIDERAÇÕES FINAIS

A partir da experiência prática na Keeps e da análise comparativa entre protocolos, ferramentas e estratégias de comunicação, conclui-se que a evolução arquitetural baseada em microsserviços transcende a dimensão tecnológica, configurando-se como um processo de transformação organizacional e cultural.

A consolidação de práticas como observabilidade distribuída, mensageria assíncrona, cache defensivo e automação contínua evidencia que a maturidade de uma arquitetura não se limita à sua estrutura técnica, mas à capacidade da equipe em operar, monitorar e aprimorar continuamente o ecossistema.

O presente trabalho, portanto, não apenas comprova os ganhos de desempenho obtidos com a modernização da comunicação entre microsserviços, mas também oferece um modelo de referência para iniciativas futuras de evolução arquitetural dentro e fora da Keeps. Os resultados alcançados e as propostas apresentadas constituem um passo importante rumo a uma plataforma mais eficiente, escalável e sustentável, preparada para os desafios de crescimento e inovação que caracterizam o cenário corporativo atual.

Para a realização deste Trabalho de Conclusão de Curso, diversas disciplinas foram fundamentais, tanto para o embasamento teórico quanto para a aplicação

prática dos conceitos desenvolvidos. As disciplinas de Engenharia de Software, Desenvolvimento de Sistemas Orientados a Objetos e Programação para Web forneceram os fundamentos necessários para a construção e organização de sistemas complexos, permitindo compreender os princípios de modularização, reuso e integração entre componentes, aspectos centrais na arquitetura de microsserviços. Computação Distribuída e Redes de Computadores foram essenciais para o domínio dos protocolos de comunicação, gerenciamento de conexões e análise de latência, conhecimentos aplicados na comparação entre REST e gRPC. Dessa forma, a integração entre teoria e prática proporcionada por essas disciplinas foi decisiva para a condução e a finalização deste trabalho.

REFERÊNCIAS

FOWLER, Susan J. **Microserviços prontos para a produção: Construindo sistemas padronizados em uma organização de engenharia de software**. Tradução: Claudio Adas. Rio de Janeiro: Alta Books, 2017.

AKSAKALLI, I. K. et al. **Deployment and communication patterns in microservice architectures: A systematic literature review**. Journal of Systems and Software, 2021. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0164121221001114>. Acesso em: 7 maio 2025.

NISWAR, M. et al. **Performance evaluation of microservices communication with REST, GraphQL, and gRPC**. International Journal of Electronics and Telecommunications, v. 70, n. 2, 2024. Disponível em: <https://ijet.ise.pw.edu.pl/index.php/ijet/article/view/10.24425-ijet.2024.149562>. Acesso em: 12 abril 2025.

GADELHA, Pedro Henrique Costa. **Projeto Dirigido por Domínio e Arquitetura de Microserviços: Um estudo de caso para um Sistema de Gerência de Pavimentos**. 2023. 70 f. Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação) – Universidade Federal da Paraíba, João Pessoa, 2023. Disponível em: https://repositorio.ufpb.br/jspui/bitstream/123456789/31687/1/Pedro%20Henrique%20Costa%20Gadelha_TCC.pdf. Acesso em: 20 abril 2025.

LERCHER, T.; LEOPOLD, H.; LI, C.; KELLER, M. **Microservice API evolution in practice: A study on strategies and challenges**. Journal of Systems and Software, v. 208, 111809, 2024. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0164121224001559>. Acesso em: 28 maio 2025

RODRIGUES, H.; RITO SILVA, A.; AVRITZER, A. **Assessment of performance and its scalability in microservice architectures: systematic literature review**. Journal of Systems and Software, v. 208, p. 111809, 19 maio 2025. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0164121225001682>. Acesso em: 3 julho 2025

LENARDUZZI, Valentina et al. **Does migrating a monolithic system to microservices decrease the technical debt?**. Journal of Systems & Software, v. 169, 110710, 2020. Disponível em:

<https://www.sciencedirect.com/science/article/pii/S0164121220301539>. Acesso em: 5 julho 2025

KHAN, Muhammad Umair; HASSAN, Sidra; IQBAL, Maham; ABDULLAH, Muhammad. **Monitoring tools for DevOps and microservices: A systematic grey literature review**. Journal of Systems and Software, v. 203, 111614, 2023. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0164121223003011>. Acesso em: 5 julho 2025

RABBITMQ. **RabbitMQ 4.1 Documentation**. 2025. Disponível em: <https://www.rabbitmq.com/docs>. Acesso em: 24 julho 2025

CONFLUENT. **Confluent – Apache Kafka for the modern enterprise**. 2025. Disponível em: <https://www.confluent.io>. Acesso em: 24 julho 2025

REDISLABS. **Redis – The real-time data platform**. 2025. Disponível em: <https://redis.io/>. Acesso em: 24 julho 2025

CNCF. *OpenTelemetry Documentation*. Cloud Native Computing Foundation, 2025. Disponível em: <https://opentelemetry.io>. Acesso em: 02 ago. 2025.

ELASTIC. *Elastic APM and OpenTelemetry Integration Guide*. Elastic NV, 2025. Disponível em: <https://www.elastic.co/apm>. Acesso em: 02 ago. 2025.

APÊNDICE A – ARTIGO MODELO SBC

Desafios e Estratégias na Implementação da Comunicação entre Microsserviços em Arquiteturas Escaláveis: Um Estudo de Caso na Keeps

Gabriel Rodrigues dos Santos Mouta¹

¹LaPeSD - Laboratório de Pesquisa em Sistemas Distribuídos – Universidade Federal de Santa Catarina (UFSC)

Florianópolis – SC – Brazil

`gabriel.mouta@grad.ufsc.br`

Abstract. *This project investigated the challenges and strategies for achieving efficient communication between microservices in scalable architectures, focusing on a case study of the company Keeps. The study analyzed the difficulties encountered during the migration from a Service-Oriented Architecture (SOA) to a microservices-based architecture, addressing issues such as latency, resource consumption, and integration failures. Additionally, communication protocols and tools such as REST, gRPC, RabbitMQ, API Gateway, and Kubernetes were evaluated. The research sought to understand the factors that impact the performance and scalability of microservices communication, presenting recommendations and strategies validated in practice that may assist other companies undergoing similar modernization processes.*

Resumo. *Este trabalho investigou os desafios e as estratégias para uma comunicação eficiente entre microsserviços em arquiteturas escaláveis, com foco no estudo de caso da empresa Keeps. O estudo analisou as dificuldades enfrentadas durante a migração de uma arquitetura SOA para microsserviços, abordando temas como latência, consumo de recursos e falhas de integração. Além disso, foram avaliados protocolos e ferramentas como REST, gRPC, RabbitMQ, API Gateway e Kubernetes. A pesquisa buscou compreender os fatores que impactam o desempenho e a escalabilidade da comunicação, apresentando recomendações e estratégias validadas na prática que podem auxiliar outras empresas em processos semelhantes de modernização arquitetural.*

1. Introdução

A arquitetura de microsserviços consolidou-se como uma abordagem eficaz para o desenvolvimento de sistemas escaláveis e altamente disponíveis. Ao promover a decomposição de aplicações em serviços independentes e especializados, esse modelo supera diversas limitações das arquiteturas monolíticas e orientadas a serviços (SOA), possibilitando maior autonomia de desenvolvimento, melhor implantação contínua e resiliência operacional [3]. Contudo, a adoção dessa arquitetura introduz novas complexidades, tornando a comunicação entre serviços, distribuída, heterogênea e sujeita à latência, um dos aspectos mais críticos a serem otimizados [1].

Este artigo apresenta um estudo de caso na Keeps, uma *EdTech* que oferece soluções voltadas à gestão e à experiência de aprendizado corporativo (*LMS/LXP*) no modelo *Software as a Service* (*SaaS*). O ecossistema da empresa, que atende mais de 500 mil colaboradores, exige alta disponibilidade e eficiência. Inicialmente operando em uma arquitetura SOA hospedada no provedor de computação em nuvem da AWS, a organização iniciou em 2021 a migração para microsserviços visando maior escalabilidade.

Entretanto, após a transição, surgiram desafios significativos relacionados à comunicação entre os serviços, como alta latência acumulada, falhas de integração, dificuldades de monitoramento e uso ineficiente de recursos computacionais. Esses fatores afetaram diretamente a performance dos sistemas e a qualidade entregue aos clientes.

O objetivo deste trabalho foi analisar e implementar estratégias para otimizar essa comunicação, focando em eficiência, resiliência e escalabilidade. Através de uma abordagem prática e experimental, avaliou-se a substituição de tecnologias de comunicação, *HTTP (Hypertext Transfer Protocol)/REST (Representational State Transfer)* por *gRPC*, a implementação de padrões de resiliência, como cache defensivo e *dead-letter queues - DLQ* e a melhoria da observabilidade com *OpenTelemetry*.

2. Fundamentação Teórica

A comunicação em sistemas distribuídos é fundamental para a integridade da arquitetura. A literatura classifica os padrões mais recorrentes em comunicação síncrona e assíncrona.

A escolha do protocolo impacta diretamente o desempenho. O padrão REST (sobre HTTP/1.1) é amplamente utilizado pela sua simplicidade, mas pode apresentar latência elevada devido ao overhead de cabeçalhos e formato textual, *JSON (JavaScript Object Notation)*. Em contrapartida, o *gRPC*, desenvolvido pelo Google, utiliza HTTP/2 e *Protocol Buffers* (binário), oferecendo menor latência, multiplexação de chamadas e contratos fortemente tipados, sendo ideal para comunicação interna entre serviços [6].

Para comunicação assíncrona, o desacoplamento temporal entre serviços é sua principal característica e ferramentas como *RabbitMQ* são utilizadas para implementação desse padrão. A confiabilidade nesse modelo depende da implementação de padrões como DLQ para mensagens não processadas e políticas de *retry* com *backoff* exponencial para lidar com falhas transientes [7].

Em ambientes distribuídos, a correlação de eventos é complexa. O *OpenTelemetry* surge como um padrão da *CNCF (Cloud Native Computing Foundation)* para instrumentação, gerando métricas, *logs* e *traces* que permitem visualizar o ciclo de vida completo de uma requisição através de múltiplos serviços [2].

3. Trabalhos Relacionados

A literatura apresenta diversos estudos sobre o desempenho de microsserviços. Niswar et al. [6] realizaram uma avaliação experimental controlada comparando REST, GraphQL e *gRPC*. Os resultados demonstraram que o *gRPC* apresenta desempenho superior em cenários de alta carga e dados aninhados, tanto em tempo de resposta quanto em consumo de CPU.

Aksakalli et al. [1] conduziram uma revisão sistemática sobre padrões de comunicação, destacando que a escolha inadequada entre síncrono e assíncrono é a principal causa de acoplamento excessivo e falhas em cascata. Lercher et al. [5] discutem os desafios da evolução de APIs, apontando que a falta de governança em contratos de serviço (como os definidos via Protobuf) gera quebras de compatibilidade em produção.

Este trabalho se diferencia ao aplicar e validar essas premissas teóricas em um cenário industrial real, enfrentando as restrições de um sistema rodando em produção e utilizando dados de tráfego real para validar as otimizações, além da experiência de um desenvolvedor do time.

4. Estudo de Caso: Arquitetura da Keeps

A plataforma da Keeps é hospedada na AWS e orquestrada via *Kubernetes*, utilizando o *Amazon EKS*. O ecossistema é composto por diversas aplicações, com destaque para o Konquest (*LMS/LXP*) e o MyAccount (Gestão de Identidade).

Antes das intervenções, a comunicação entre os serviços era predominantemente síncrona via REST. A análise técnica identificou fragilidades críticas:

- a) Acoplamento Temporal e Latência: O serviço Konquest realizava chamadas síncronas ao MyAccount para validar permissões a cada requisição. Em picos de acesso, a latência de rede somada ao tempo de processamento gerava gargalos, elevando o tempo de resposta final (P99) para valores inaceitáveis. O Konquest, devido à sua centralização de regras, atuava tecnicamente como um "macro-serviço", dificultando o escalonamento isolado.
- b) Perda de Mensagens: A configuração inicial do RabbitMQ não utilizava DLQ. Mensagens que falhavam no consumo (por exemplo, devido a erros de banco de dados momentâneos) eram descartadas silenciosamente, gerando inconsistência de dados entre serviços.
- c) Observabilidade Fragmentada: A falta de rastreamento distribuído impedia a identificação da causa raiz. Logs eram isolados por pod, tornando impossível rastrear uma transação que atravessasse múltiplos serviços.

5. Melhorias Implementadas

Para mitigar os problemas citados, foi executado um plano de reestruturação focado em três pilares: Padronização da Comunicação, Resiliência e Observabilidade.

5.1. Padronização com gRPC e RPC Assíncrono

Baseado nos resultados de Niswar et al. [6], o protocolo gRPC foi adotado como padrão para a comunicação interna síncrona, substituindo o REST. Isso reduziu o tamanho do *payload* e o custo de serialização/desserialização.

Além disso, para a comunicação síncrona via mensageria, utilizando o *broker* RabbitMQ, foi padronizado o uso de *callback queues* exclusivas e identificadores de correlação. Isso permitiu que os serviços solicitantes correlacionassem respostas a requisições específicas de forma assíncrona, mantendo o desacoplamento e evitando o bloqueio de processos.

5.2. Estratégia de Cache Defensivo

Para resolver o acoplamento temporal com o MyAccount, foi implementado o padrão de Cache Defensivo utilizando Redis [8]. Dados de alta frequência de leitura e baixa mutabilidade, como permissões de usuários, passaram a ser armazenados temporariamente no serviço consumidor, exemplo Konquest. Isso eliminou chamadas

redundantes e garantiu que o sistema continuasse operando mesmo em instabilidades momentâneas do serviço provedor de identidade.

5.3. Resiliência e Consistência Eventual

A arquitetura de mensageria foi melhorada com a configuração de DLQs para capturar falhas. Adicionalmente, implementou-se uma estratégia de Consistência Eventual via código: em casos de falha na replicação de dados de usuários via *CQRS* (*Command Query Responsibility Segregation*), a requisição não é perdida; ela é armazenada com um estado de "pendente" e uma execução agendada reprocessa o envio assim que os dados de dependência são replicados e consistentes.

5.4. Observabilidade Unificada

Foi configurado o *OpenTelemetry* [2], utilizando o *OpenTelemetry Operator* no *Kubernetes* para injeção automática de instrumentação. Isso permitiu a correlação de métricas, *logs* e *traces*. A análise dos *traces* no Elastic APM revelou, que em um certo *endpoint*, 82% do tempo de resposta classificado como "HTTP" era, na verdade, derivado de latência em operações de banco de dados nos serviços dependentes, e não lentidão do protocolo em si. Neste caso, dependência com o Myaccount e serviços externos.

6. Resultados e Discussão

Para validar as melhorias, foram realizados testes de carga utilizando a ferramenta *k6* em ambiente controlado, simulando o fluxo crítico de "*Tracking de Conteúdo*" no serviço Konquest. O cenário comparou a versão original (v1: HTTP Síncrono sem *cache*) com a versão otimizada (v2: gRPC + *Cache Redis*).

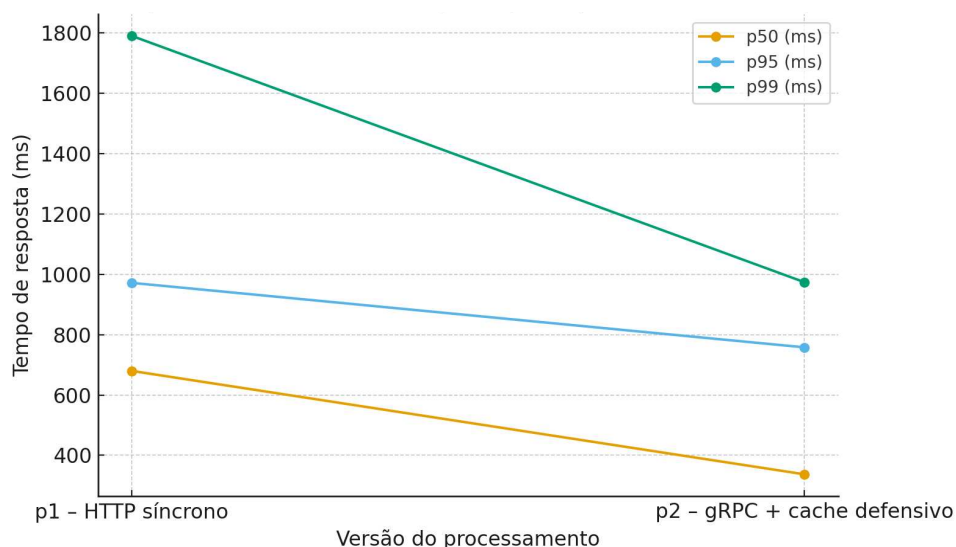
Os testes simularam 50 usuários virtuais simultâneos durante 10 minutos. A Tabela 1 resume os resultados obtidos.

Tabela 1. Comparativo de desempenho antes e depois das otimizações

Versão	p50 (ms)	p95 (ms)	p99 (ms)	CPU Média (%)	Throughput (req/s)
v1	680	972	1791	93	78
v2	337	758	974	84	167

A latência média (P50) foi reduzida em aproximadamente 50% (de 680ms para 337ms). Mais impactante para a estabilidade do sistema foi a redução na latência de cauda (P99), que caiu 45%, indicando um comportamento muito mais previsível sob carga. A vazão do sistema mais que dobrou, passando de 78 para 167 requisições por segundo, demonstrando um grande aumento na capacidade de processamento.

Figura 1. Comparativo de tempo de resposta entre as abordagens



Observou-se uma redução no consumo médio de CPU de 93% para 84% na versão otimizada, mesmo com o dobro de requisições processadas. Isso está relacionado à eficiência do protocolo binário gRPC frente ao processamento textual do JSON e também ao uso do *cache*.

Vale ressaltar que, em análises de carga máxima (como nos estudos de Niswar et al. [6]), a utilização de CPU pode ultrapassar 100%. Isso ocorre porque a métrica em ambientes de contêineres (*multi-core*) contabiliza a soma da utilização de cada núcleo disponível. Valores acima de 100% indicam que a aplicação, através de processamento paralelo eficiente, conseguiu utilizar múltiplos núcleos simultaneamente para atender à alta demanda, algo que a versão bloqueante anterior tinha dificuldade em realizar.

7. Conclusão

Este trabalho demonstrou que a otimização da comunicação entre microserviços é um requisito vital para a escalabilidade do ambiente. A aplicação prática de gRPC e Redis na arquitetura da Keeps resultou não apenas em ganhos quantitativos, redução de 50% na latência e aumento de 114% na vazão (*throughput*), mas também em maior estabilidade operacional.

A implementação de observabilidade com *OpenTelemetry* elevou a maturidade da equipe, permitindo diagnósticos precisos baseados em dados e não em suposições. Conclui-se que, para ambientes de alta demanda, a transição de REST para gRPC em comunicações internas, aliada a estratégias de *cache* defensivo, oferece um retorno técnico imediato.

Como trabalhos futuros, sugere-se a decomposição do serviço Konquest, identificado neste estudo como um gargalo devido à sua natureza de "macro-serviço", em microsserviços menores, e a avaliação da implementação de um Service Mesh para gestão avançada de tráfego e segurança mTLS. Além de melhorias na governança e lançamento de novas versões utilizando estratégias mais inteligentes, por fim, melhorias na configuração e estudo de alertas no campo da observabilidade.

Referências

- [1] AKSAKALLI, I. K. et al. Deployment and communication patterns in microservice architectures: A systematic literature review. *Journal of Systems and Software*, 2021.
- [2] CNCF. OpenTelemetry Documentation. Cloud Native Computing Foundation, 2025. Disponível em: <https://opentelemetry.io>.
- [3] FOWLER, S. J. Microsserviços prontos para a produção. Rio de Janeiro: Alta Books, 2017.
- [4] KHAN, M. U. et al. Monitoring tools for DevOps and microservices: A systematic grey literature review. *Journal of Systems and Software*, v. 203, 2023.
- [5] LERCHER, T.; LEOPOLD, H.; LI, C. Microservice API evolution in practice: A study on strategies and challenges. *Journal of Systems and Software*, v. 208, 2024.
- [6] NISWAR, M. et al. Performance evaluation of microservices communication with REST, GraphQL, and gRPC. *International Journal of Electronics and Telecommunications*, v. 70, n. 2, 2024.
- [7] RABBITMQ. RabbitMQ Documentation. 2025. Disponível em: <https://www.rabbitmq.com/docs>.
- [8] REDIS LABS. Redis: The real-time data platform. 2025. Disponível em: <https://redis.io/>.
- [9] RODRIGUES, H.; RITO SILVA, A.; AVRITZER, A. Assessment of performance and its scalability in microservice architectures. *Journal of Systems and Software*, v. 208, 2025.