INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA DO PIAUÍ
CAMPUS PICOS
ANÁLISE E DESENVOLVIMENTO DE SISTEMAS

PEDRO HENRIQUE RODRIGUES DAMACENA

REST E GRPC EM ARQUITETURAS DE MICROSERVIÇOS: DESEMPENHO, COMPLEXIDADE E ADEQUAÇÃO

PICOS, PIAUÍ
2025

PEDRO HENRIQUE RODRIGUES DAMACENA

REST E GRPC EM ARQUITETURAS DE MICROSERVIÇOS: DESEMPENHO, COMPLEXIDADE E ADEQUAÇÃO

Trabalho de Conclusão de Curso (Artigo Científico) apresentado como exigência parcial para obtenção do diploma do Curso de Tecnologia em Análise e Desenvolvimento de Sistemas do Instituto Federal de Educação, Ciência e Tecnologia do Piauí, Campus Picos.

Orientador: Prof. M^{e}. Joao Paulo Lima do Nascimento

PICOS, PIAUÍ
2025

Pedro Henrique Rodrigues Damacena[1]
Prof. Me. Joao Paulo Lima do Nascimento[2]

RESUMO

A crescente adoção de arquiteturas de microserviços reflete a busca por sistemas mais escaláveis, resilientes e flexíveis frente às demandas atuais do desenvolvimento de software. Nesse contexto, a comunicação entre serviços torna-se um fator decisivo para a eficiência e a manutenibilidade das aplicações, sendo REST e gRPC duas das tecnologias mais utilizadas e debatidas nesse cenário. Este artigo tem como objetivo realizar uma análise comparativa entre REST e gRPC, destacando suas características técnicas, vantagens, limitações e adequações a diferentes contextos de aplicação. A pesquisa foi conduzida com abordagem exploratória e comparativa, fundamentada em levantamento bibliográfico de artigos, estudos de caso e livros de referência. Foram definidos três critérios de avaliação: desempenho e latência, complexidade de implementação e adequação a cenários práticos, que orientaram a análise crítica das duas abordagens. Os resultados evidenciam que o REST se mantém como uma solução consolidada e amplamente adotada, oferecendo simplicidade, interoperabilidade e suporte robusto do ecossistema, embora apresente limitações de latência e menor eficiência em ambientes de alta demanda. Já o gRPC demonstra maior eficiência em termos de desempenho e suporte a comunicação em tempo real, destacando-se em cenários que exigem baixa latência, como plataformas financeiras e jogos online, porém com maior complexidade de implementação e curva de aprendizado mais acentuada. Conclui-se que a escolha entre REST e gRPC não é universal, devendo considerar as necessidades específicas do projeto e a maturidade técnica da equipe. Além disso, o estudo sugere a adoção de abordagens híbridas, que combinem as duas tecnologias de modo estratégico, explorando suas complementaridades.

Palavras-chaves: microserviços; REST; gRPC; protocolos de comunicação; arquitetura de software.

[1] Discente do curso de Tecnologia em Análise e Desenvolvimento de Sistemas,IFPI.
[2] Docente do curso de Tecnologia em Análise e Desenvolvimento de Sistemas,IFPI.

ABSTRACT

The increasing adoption of microservices architectures reflects the need for scalable, resilient, and flexible systems capable of meeting current demands in software development. In this context, communication between services becomes a decisive factor for efficiency and maintainability, with REST and gRPC standing out as two of the most widely used and debated technologies. This article primarily aims to perform a comparative analysis between REST and gRPC, emphasizing their technical characteristics, advantages, limitations, and suitability to different application scenarios. The research followed an exploratory and comparative approach, based on a literature review of articles, case studies, and reference books. Three evaluation criteria were defined: performance and latency, implementation complexity, and suitability to practical contexts, which guided the critical analysis of both approaches. The results show that REST remains a consolidated and widely adopted solution, offering simplicity, interoperability, and a robust ecosystem, despite higher latency and less efficiency in high-demand environments. Conversely, gRPC achieves superior performance and real-time communication support, excelling in scenarios that require low latency, such as financial platforms and online games, but it involves greater implementation complexity and a steeper learning curve. The study concludes that the choice between REST and gRPC is not universal but must be aligned with the project's specific requirements and the team's technical maturity. Furthermore, a hybrid approach is suggested, strategically combining both technologies to leverage their complementary strengths.

Keywords: microservices; REST; gRPC; communication protocols; software architecture.

Data de aprovação: 28/08/2025 (data de apresentação do TCC)

1 INTRODUÇÃO

Nas últimas décadas, a evolução das práticas de desenvolvimento de *software* tem impulsionado a busca por arquiteturas mais flexíveis, escaláveis e adaptáveis às demandas de negócios em constante mudança. Nesse contexto, a arquitetura de microserviços tem ganhado destaque como uma alternativa moderna à tradicional abordagem monolítica. Os microserviços consistem na decomposição de um sistema em múltiplos serviços menores, independentes e especializados, que se comunicam entre si para compor a funcionalidade global da aplicação. Cada microserviço é responsável por um conjunto específico de funcionalidades, podendo ser desenvolvido, implantado e escalado de forma autônoma. Entre as principais características que tornam essa arquitetura atrativa estão a autonomia, a escalabilidade horizontal, a resiliência a falhas e a agilidade no desenvolvimento e implantação contínua (Fowler (2015), Newman (2021)).

A popularização dos microserviços é amplamente abordada por autores renomados como Martin Fowler e Sam Newman, cujas contribuições destacam os benefícios da fragmentação de sistemas monolíticos em componentes menores e mais gerenciáveis. Segundo esses autores, essa abordagem promove um ciclo de entrega mais rápido, maior alinhamento com estruturas organizacionais ágeis e uma maior facilidade na adoção de tecnologias heterogêneas dentro do mesmo ecossistema (Fowler (2015), Newman (2021)). Além disso, outros estudos reforçam esses argumentos ao apontar que empresas que adotam microserviços conseguem responder com maior rapidez às mudanças de mercado (Dragoni et al. (2017)).

Apesar dos benefícios, a adoção de microserviços também traz uma série de novos desafios, especialmente no que diz respeito à complexidade da infraestrutura e da comunicação entre os serviços. Um dos principais problemas enfrentados na adoção dessa arquitetura é justamente a comunicação entre os serviços distribuídos. Em contraste com sistemas monolíticos — nos quais os componentes compartilham a mesma base de código e memória — os microserviços operam em processos isolados e, frequentemente, em máquinas distintas (Chen (2015), Thönes (2015)). Isso impõe a necessidade de mecanismos de comunicação interprocessual (IPC) confiáveis e eficientes. Sem uma estratégia de comunicação bem definida, a interoperabilidade entre os serviços pode se tornar um gargalo crítico, comprometendo a coesão e o desempenho geral do sistema (Taibi, Lenarduzzi e Pahl (2018), Francesco, Malavolta e Lago (2017)).

A importância de uma comunicação eficiente é evidenciada por estudos de caso como o da SoundCloud[3], que enfrentou dificuldades de escalabilidade devido ao uso

[3] Para mais informações sobre a empresa SoundCloud, consulte o site oficial: <https://soundcloud.com/>.

excessivo de chamadas ao protocolo de transferência de hipertexto (*Hypertext Transfer Protocol* - HTTP)[4] síncronas entre serviços, gerando *cascading failures* e tempos de resposta elevados. A solução adotada foi a introdução de uma camada de mensagens assíncronas com Kafka[5], que reduziu significativamente a latência e aumentou a resiliência do sistema (Balalaie, Heydarnoori e Jamshidi (2016)). Outro exemplo é o da Netflix, que inicialmente sofria com falhas em cadeia causadas por dependências diretas entre microserviços. A empresa implementou um padrão baseado em filas e *fallback* com o uso de *circuit breakers*[6] via Hystrix[7], reforçando a importância de abordagens resilientes de comunicação (Nadareishvili et al. (2016)).

Esses casos demonstram que a escolha e o entendimento correto das tecnologias de comunicação — como REST, gRPC, mensagens assíncronas ou filas de eventos — são decisivos para o sucesso de uma arquitetura de microserviços eficiente. Essas tecnologias não apenas viabilizam a troca de informações entre serviços, mas também influenciam diretamente atributos como desempenho, tolerância a falhas e escalabilidade.

Diante desse cenário, este artigo tem como objetivo comparar duas das pricipais tecnologias e abordagens de comunicação utilizadas em arquiteturas de microserviços, com foco nos aspectos técnicos, vantagens, limitações e adequação a diferentes contextos. Pretende-se oferecer um guia introdutório, claro e prático, que auxilie estudantes, desenvolvedores iniciantes e profissionais em transição a compreender as implicações das decisões tecnológicas e a escolher, de forma mais fundamentada, as soluções mais apropriadas para suas necessidades.

O artigo está organizado da seguinte forma: na Seção 3, apresenta-se o referencial teórico, abordando os conceitos fundamentais, as principais tecnologias de comunicação e alguns trabalhos correlatos; na Seção 4, realiza-se a análise comparativa entre as abordagens selecionadas, considerando critérios técnicos e operacionais; e, por fim, na Seção 5, é apresentada a síntese comparativa, consolidando os resultados e destacando as conclusões do estudo.

2 METODOLOGIA

A pesquisa foi conduzida a partir de uma abordagem comparativa, de natureza exploratória e aplicada, com o objetivo de analisar as características técnicas e operacionais das arquiteturas de comunicação REST e gRPC em ambientes de microserviços.

[4] <https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview>.

[5] Apache Kafka é uma plataforma de streaming de eventos distribuída de código aberto. Mais informações em: <https://kafka.apache.org/>.

[6] Circuit Breakers são padrões de resiliência importantes em arquiteturas de microsserviços. Para mais detalhes, veja a documentação de projetos como o Hystrix: <https://github.com/Netflix/Hystrix/wiki>.

[7] Hystrix é uma biblioteca que ajuda a controlar as interações entre esses serviços distribuídos. Mais informações podem ser encontradas em seu repositório GitHub: <https://github.com/Netflix/Hystrix>.

Para isso, adotou-se como referência trabalhos teóricos e experimentais da literatura recente, buscando consolidar fundamentos que permitissem uma avaliação crítica dos dois modelos. A escolha desse método se justifica pela necessidade de compreender não apenas as vantagens e desvantagens de cada tecnologia, mas também sua adequação a diferentes cenários de aplicação.

O estudo foi estruturado em três etapas principais. Na primeira, realizou-se um levantamento bibliográfico em artigos, livros e estudos de caso que abordam microserviços, protocolos de comunicação e desempenho em sistemas distribuídos. Esse levantamento forneceu o embasamento necessário para compreender os conceitos fundamentais de arquitetura de software, bem como a evolução das práticas de desenvolvimento que motivaram a adoção de REST e gRPC.

Na segunda etapa, foram definidos os critérios de análise que orientaram a comparação entre as tecnologias. Os critérios selecionados foram: desempenho e latência, complexidade de implementação e adequação a casos de uso. Cada critério foi escolhido com base em sua relevância para a adoção prática das arquiteturas de comunicação, considerando tanto aspectos técnicos quanto operacionais. Essa definição garantiu uma análise consistente e alinhada ao objetivo central do trabalho.

Por fim, na terceira etapa, realizou-se a análise comparativa propriamente dita, aplicando os critérios estabelecidos às duas abordagens. A análise combinou evidências teóricas e resultados empíricos de estudos prévios, permitindo identificar padrões, pontos fortes e limitações de REST e gRPC. A partir dessa síntese, buscou-se oferecer uma visão clara e prática, de modo a orientar profissionais e estudantes na escolha mais adequada de tecnologia para diferentes contextos de microserviços.

3 FUNDAMENTAÇÃO TEÓRICA

Esta seção está organizada em três subseções. Na subseção 3.1, apresenta-se a definição de arquitetura de software e os principais estilos arquiteturais, com foco nas abordagens monolítica e de microserviços. A subseção 3.2 aborda as tecnologias e estratégias de comunicação em microserviços, destacando as arquiteturas REST e gRPC. Por fim, a subseção 3.3 apresenta os trabalhos correlatos, reunindo estudos relevantes que embasam e contextualizam a pesquisa.

3.1. CONCEITOS DE ARQUITETURA DE SOFTWARE

A arquitetura de software pode ser compreendida como a estrutura fundamental de um sistema, que define a organização de seus componentes, as relações entre eles e os princípios que orientam seu design e evolução. Martin Fowler enfatiza que a arquitetura não se restringe apenas a diagramas ou decisões iniciais, mas envolve

escolhas críticas que afetam profundamente as qualidades de um sistema, como desempenho, escalabilidade, manutenibilidade e segurança. Ela atua como um guia para o desenvolvimento, ajudando equipes a tomar decisões coerentes ao longo do ciclo de vida do software, e servindo como base para a comunicação entre desenvolvedores e demais partes interessadas.

Além disso, a arquitetura de software é um campo dinâmico, que evolui de acordo com as necessidades do negócio, avanços tecnológicos e contextos operacionais. Fowler ressalta que, embora a arquitetura imponha limites e diretrizes, ela deve manter-se flexível para acomodar mudanças e inovações. Essa perspectiva torna a arquitetura não apenas um artefato técnico, mas também um processo contínuo de adaptação e refinamento, no qual decisões sobre estilos arquiteturais — como monólitos e microserviços — impactam diretamente a forma como o sistema é desenvolvido, implantado e mantido (Fowler (2013)).

Nesta subseção, serão explorados dois dos estilos arquiteturais mais proeminentes no desenvolvimento de software: a arquitetura monolítica e a arquitetura de microserviços. A compreensão de suas características, vantagens e desvantagens é fundamental para contextualizar a discussão sobre os mecanismos de comunicação, que diferem significativamente entre esses paradigmas.

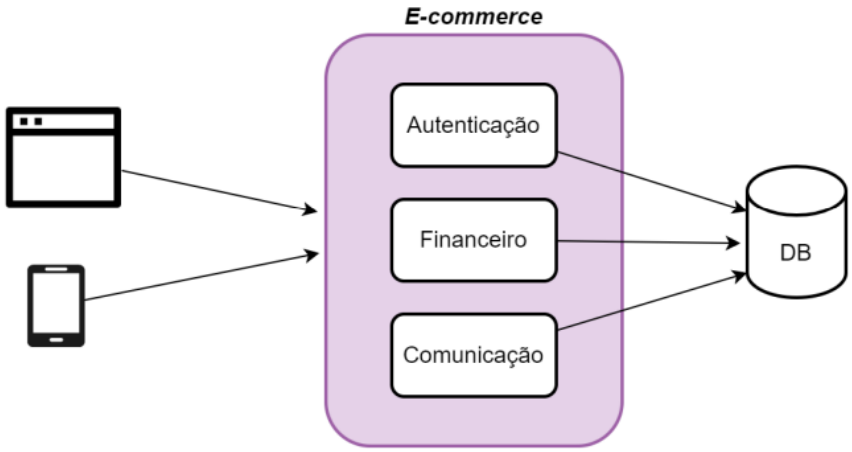
3.1.1. Arquitetura Monolítica

A arquitetura monolítica representa um modelo tradicional de desenvolvimento de software, no qual todos os componentes da aplicação — incluindo interface de usuário, lógica de negócio e acesso a dados — estão integrados em um único artefato executável. De acordo com Taibi e Lenarduzzi (2018), essa abordagem é comum em sistemas legados e favorece, inicialmente, a simplicidade de desenvolvimento, implantação e gerenciamento, especialmente em projetos pequenos ou com equipes centralizadas. Segundo Dragoni et al. (2017), a estrutura monolítica permite uma visão unificada do sistema, o que pode ser benéfico para o controle de versões e a integração de funcionalidades fortemente acopladas, reduzindo a complexidade inicial do projeto.

Entre as principais vantagens da arquitetura monolítica, destacam-se a menor sobrecarga operacional e a facilidade de depuração em ambientes homogêneos. Kamisetty et al. (2023) observam que, por utilizarem chamadas internas e locais, os monólitos eliminam a complexidade da comunicação em rede e reduzem significativamente a latência entre módulos. Da mesma forma, Balalaie et al. (2018) reforçam que, em aplicações com lógica de negócio simples ou com baixos requisitos de escalabilidade independente, a manutenção de um único deploy pode ser mais eficiente em termos de tempo e custo em relação a uma arquitetura distribuída complexa.

A Figura 1 ilustra a arquitetura de um sistema monolítico de E-commerce, onde um cliente interage através de um navegador ou dispositivo móvel com um bloco central denominado "E-commerce". Este bloco monolítico integra três componentes principais: "Autenticação", responsável pela verificação da identidade do usuário; "Financeiro", encarregado do processamento de pagamentos e transações; e "Comunicação", que gerencia a interação com o cliente. Todos esses componentes monolíticos acessam e compartilham um único banco de dados ("DB"), demonstrando a natureza interligada e a dependência de uma infraestrutura de dados unificada em sistemas monolíticos.

Figura 1 – Exemplo de Arquitetura Monolítica



Fonte: Autoria própria

Por outro lado, a escalabilidade limitada e o forte acoplamento entre os módulos tornam-se desvantagens críticas à medida que o sistema cresce. Taibi, Lenarduzzi e Pahl (2018) destacam que qualquer alteração em um módulo pode exigir uma refatoração de toda a aplicação, o que pode introduzir riscos operacionais e aumentar o tempo de entrega. Já Katal et al. (2025) observam que o crescimento contínuo do código-fonte em arquiteturas monolíticas resulta em dependências rígidas e dificuldades na introdução de novas tecnologias, comprometendo a evolução arquitetural e a flexibilidade em ambientes que demandam mudanças frequentes.

3.1.2. Arquitetura de Microserviços

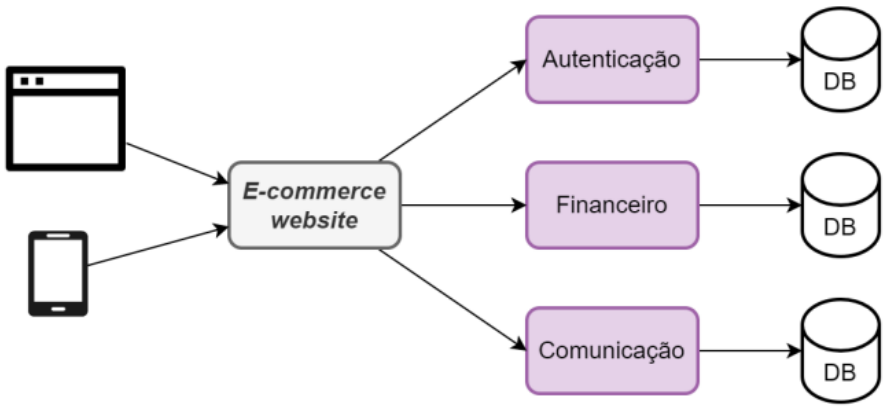
A arquitetura de microserviços é uma abordagem de design que promove a separação de aplicações em componentes independentes e distribuídos. Segundo Kondreddy e Rastogi (2025), essa arquitetura é caracterizada pela autonomia de desenvolvimento, escalabilidade granular, resiliência frente a falhas e maior agilidade organizacional. Essa independência permite que equipes desenvolvam, testem e implantem serviços separadamente, otimizando os ciclos de entrega e facilitando a manutenção contínua. De acordo com Li et al. (2021), os microserviços também favorecem a flexibilidade

arquitetural em ambientes distribuídos, proporcionando maior capacidade de adaptação a mudanças rápidas do mercado.

Entre os principais benefícios dos microserviços, destaca-se a entrega contínua e rápida de software. Segundo SAID et al. (2025), a adoção dessa arquitetura contribui para ciclos de desenvolvimento mais curtos e maior reutilização de componentes, o que reduz custos e tempo de produção. Além disso, Garimilla (2024) enfatiza que os microserviços se alinham naturalmente a práticas ágeis, permitindo uma segmentação técnica e organizacional que apoia o uso de diferentes tecnologias — ou seja, a heterogeneidade tecnológica — conforme as necessidades específicas de cada serviço, sem comprometer a coesão do sistema como um todo.

A Figura 2 apresenta um exemplo de arquitetura de microsserviços. Em contraste com a abordagem monolítica, o sistema de E-commerce é decomposto em serviços menores e independentes. A partir do website, as requisições são direcionadas para serviços específicos: o de Autenticação, o Financeiro e o de Comunicação. Uma característica fundamental desta arquitetura, como ilustrado, é que cada um desses microsserviços possui e gerencia seu próprio banco de dados (DB), promovendo autonomia e isolamento entre os componentes do sistema.

Figura 2 – Exemplo de Arquitetura de Microsserviços



Fonte: Autoria própria

É importante ressaltar, porém, que os microserviços também trazem desafios técnicos significativos. Rodríguez et al. (2021) apontam que a implantação e manutenção de uma arquitetura baseada em serviços exige uma infraestrutura complexa, que envolva orquestração de *containers*, *pipelines* de Integração Contínua e Entrega Contínua (CI/CD), monitoramento distribuído e automação de testes. Já Suleiman e Murtaza (2024) observam que, além da infraestrutura, a comunicação entre os serviços se torna um dos pontos mais sensíveis, pois envolve chamadas em rede, controle de latência, tolerância a falhas e eventual consistência, especialmente em ambientes com alta taxa de requisições e dependência entre serviços.

3.2. MECANISMOS DE COMUNICAÇÃO EM MICROSERVIÇOS

A comunicação entre serviços é um elemento crucial em arquiteturas de microserviços, influenciando diretamente o desempenho, a escalabilidade e a resiliência do sistema. Nesse contexto, a comparação entre diferentes abordagens torna-se fundamental para compreender seus impactos técnicos e operacionais, orientando a escolha da solução mais adequada. Segundo Hedelin (2024), a escolha do protocolo adequado deve levar em conta não apenas métricas de desempenho, mas também aspectos operacionais e contextuais, de modo a garantir maior eficiência e robustez ao sistema.

3.2.1. REST (Representational State Transfer)

A arquitetura REST (Representational State Transfer) foi proposta por Fielding (2000) em sua tese de doutorado na Universidade da Califórnia, Irvine, em 2000, como um estilo arquitetural para sistemas distribuídos, especialmente na web. O REST define um conjunto de restrições arquiteturais que orientam o desenvolvimento de aplicações escaláveis e interoperáveis, baseando-se em uma comunicação entre cliente e servidor orientada a recursos. Esses recursos são identificados por URIs (Uniform Resource Identifiers), e sua manipulação é feita por meio de representações, geralmente em formatos como *Javascript Object Notation*(JSON) ou *Extensible Markup Language*(XML). Essa abordagem valoriza a simplicidade e o uso uniforme de protocolos amplamente adotados, como o HTTP (Richardson, Amundsen e Ruby (2013)).

Um dos pilares do REST é o conceito de uniform interface, que estabelece a padronização da comunicação entre os componentes do sistema. Essa padronização se dá através do uso semântico dos métodos HTTP, como GET (recuperação de dados), POST (criação de recursos), PUT (atualização) e DELETE (remoção) (Fielding (2000)). Ao tratar recursos como entidades abstratas acessíveis via URI, REST promove uma separação clara entre cliente e servidor, permitindo que cada um evolua de forma independente. Além disso, outras restrições como a ausência de estado no servidor (*statelessness*) e a capacidade de cache contribuem para a escalabilidade e o desempenho da arquitetura (Wilde e Pautasso (2011)).

As vantagens técnicas do REST são amplamente reconhecidas. Sua simplicidade reduz a complexidade do desenvolvimento e da manutenção de sistemas distribuídos (Richardson, Amundsen e Ruby (2013)). Por ser baseado no protocolo HTTP, REST herda sua ampla compatibilidade com diversas plataformas, linguagens e ferramentas. Essa interoperabilidade facilita a integração entre sistemas heterogêneos. A escalabilidade é favorecida pela restrição stateless, que simplifica o balanceamento de carga e a replicação de servidores. Além disso, a adoção generalizada do REST contribui para um ecossistema rico em bibliotecas e boas práticas consolidadas (Newman (2021)).

No entanto, REST não está isento de críticas. Uma das principais limitações é a ausência de contratos formais de interface, o que pode dificultar a validação e a evolução segura das APIs como aponta Masse (2011). Além disso, a natureza stateless implica que cada requisição deve conter todas as informações necessárias, o que pode resultar em sobrecarga de rede, especialmente em contextos com alta latência. REST também não é a abordagem mais adequada para aplicações que requerem comunicação em tempo real, como chats ou jogos online, onde soluções baseadas em WebSockets ou protocolos assíncronos podem oferecer maior eficiência (Wilde e Pautasso (2011)).

3.2.2. gRPC (Google Remote Procedure Call)

A arquitetura gRPC (Google Remote Procedure Call) representa uma evolução moderna dos paradigmas clássicos de chamada de procedimento remoto (Remote Procedure Call, RPC), os quais visam abstrair a complexidade da comunicação entre processos em redes distribuídas. Criado pelo Google, o gRPC surgiu da necessidade de oferecer uma alternativa eficiente, interoperável e extensível para integrar sistemas baseados em microserviços, especialmente diante das limitações do REST em cenários de comunicação de baixa latência e alto desempenho. Com suporte nativo ao protocolo HTTP/2 e ao mecanismo de serialização Protocol Buffers (protobuf), o gRPC é ideal para ambientes altamente escaláveis e heterogêneos (Carthen et al. (2024), Zaniewski e Roszczyk (2024)).

O gRPC baseia-se em contratos de serviço fortemente tipados definidos em arquivos .proto, nos quais as mensagens e métodos são especificados de forma abstrata, gerando automaticamente código cliente e servidor em diversas linguagens de programação como Go, Java, Python e C. A utilização do HTTP/2 permite comunicação multiplexada, menor sobrecarga e suporte nativo a streaming bidirecional — características que ampliam significativamente as possibilidades de comunicação síncrona e assíncrona (Chen et al. (2023)). Essa abordagem viabiliza desde chamadas simples (unárias) até fluxos contínuos de dados em tempo real, atendendo demandas típicas de sistemas distribuídos modernos.

Dentre suas vantagens técnicas, o gRPC se destaca por sua eficiência na serialização de mensagens (graças ao Protobuf), sua baixa latência e sua capacidade de integração com arquiteturas orientadas a eventos e serviços. Estudos empíricos demonstram superioridade do gRPC em termos de tempo de resposta e uso de rede quando comparado a REST e SOAP, sobretudo em contextos de aplicações com alto volume de chamadas e requisitos de comunicação em tempo real (HEDELIN, 2024). Adicionalmente, sua adoção tem se consolidado em ecossistemas de computação em nuvem e 5G, onde o desempenho e a padronização são essenciais.

Apesar de seus méritos, o gRPC apresenta algumas limitações notáveis. Sua

compatibilidade direta com navegadores é restrita, exigindo o uso de gRPC-Web como camada intermediária, o que pode comprometer parte de sua performance. Ademais, a adoção do Protobuf — embora eficiente — impõe uma curva de aprendizado acentuada para desenvolvedores acostumados a formatos como JSON. A interoperabilidade com clientes RESTful legados também requer soluções adaptativas ou gateways, dificultando sua integração em arquiteturas híbridas.

3.3. ESTUDOS RELACIONADOS

A comunicação eficiente entre serviços é um dos pilares para o sucesso de arquiteturas baseadas em microserviços. Com a crescente complexidade e adoção dessa abordagem arquitetural, diversos estudos vêm se dedicando a investigar os impactos e as alternativas tecnológicas relacionadas à comunicação interserviços. Nesse contexto, alguns trabalhos se destacam por oferecer contribuições relevantes para o entendimento e a evolução desse domínio, ainda que com escopos distintos do objetivo deste artigo.

Weerasinghe e Perera (2024) realizaram uma análise experimental sobre como diferentes estratégias de comunicação afetam o desempenho de sistemas microserviço-orientados. O estudo avalia cenários de decomposição de sistemas monolíticos, testando tecnologias como REST e gRPC, e propõe uma estratégia otimizada baseada em comunicação assíncrona. Embora tecnicamente robusto, o foco do trabalho está em otimizações de desempenho específicas para sistemas já estabelecidos, e não aborda de forma introdutória os conceitos e critérios para escolha de tecnologias — o que torna sua aplicabilidade limitada para leitores iniciantes.

Cruz (2021) realizou uma modelagem prática de microsserviços e arquiteturas de APIs, utilizando diferentes estilos de comunicação, como REST, gRPC, SOAP[8] e GraphQL[9], para a construção de aplicações distribuídas. O autor destaca que o conhecimento sobre esses padrões é essencial para que os desenvolvedores saibam onde e como aplicá-los, visto que cada abordagem apresenta características distintas que favorecem diferentes cenários. Em sua análise, o autor conclui que arquiteturas baseadas em microserviços trazem benefícios significativos para aplicações de grande porte, como a divisão eficiente de equipes, aumento da produtividade e maior resiliência a falhas. Além disso, ele ressalta que, embora padrões modernos como REST e gRPC ofereçam maior desempenho na comunicação entre serviços, padrões mais antigos como SOAP apresentam limitações nesse aspecto. No entanto, o estudo não promove uma comparação sistemática entre essas abordagens nem se destina a iniciantes.

[8] SOAP (Simple Object Access Protocol) é um protocolo baseado em XML para acessar serviços web. Ele é usado para trocar informações estruturadas em um ambiente descentralizado e distribuído.

[9] GraphQL é uma linguagem de consulta para APIs e um tempo de execução para executar essas consultas com seus dados existentes. Mais informações em: <https://graphql.org/>.

Outro estudo relevante é o de Hedelin (2024), que realiza uma análise comparativa entre os protocolos de comunicação REST, gRPC e SOAP, com foco em microserviços. O trabalho oferece uma visão abrangente do desempenho e da latência dessas tecnologias em cenários reais, contribuindo significativamente para a escolha adequada de protocolo conforme o contexto da aplicação. Em contraste com estudos mais limitados a um único ecossistema, como Java/Spring, este trabalho aborda múltiplas abordagens com base em métricas objetivas de desempenho.

Em contraste com os trabalhos mencionados, este artigo propõe uma análise comparativa, acessível e orientada à prática de duas das principais tecnologias de comunicação utilizadas em arquiteturas de microserviços — REST e gRPC. Seu objetivo é preencher uma lacuna ainda presente na literatura: a falta de um guia introdutório e didático que auxilie profissionais iniciantes a compreenderem as implicações técnicas e estratégicas de cada abordagem e, assim, tomarem decisões mais fundamentadas no design e evolução de sistemas distribuídos.

4 ANÁLISE COMPARATIVA ENTRE REST E GRPC

Esta seção apresenta uma análise comparativa das principais tecnologias e abordagens de comunicação em arquiteturas de microserviços. Está estruturada em duas subseções: a subseção 4.1 descreve os critérios utilizados para a comparação, enquanto a subseção 4.2 realiza a análise detalhada das abordagens de comunicação selecionadas.

4.1. CRITÉRIOS DE AVALIAÇÃO

Para a análise, foram estabelecidos os seguintes critérios, considerados fundamentais para avaliar a adequação das diferentes tecnologias de comunicação em diversos contextos de microserviços:

4.1.1. Desempenho e Latência

Este critério avalia a velocidade de troca de informações entre os microserviços e o tempo de resposta geral do sistema. Engloba aspectos como a sobrecarga de protocolo, a eficiência na serialização/desserialização de dados e a capacidade de processar um alto volume de requisições. Como evidenciado por Maltsev e Amin (2024), o formato de serialização impacta diretamente a latência inter-serviços e o *throughput* geral em ambientes distribuídos, sendo decisivo na escolha da tecnologia de comunicação mais adequada para sistemas de alto desempenho.

4.1.2. Complexidade de Implementação

Refere-se ao esforço necessário para projetar, desenvolver e manter a comunicação entre os serviços. Considera a curva de aprendizado da tecnologia, a disponibilidade de bibliotecas e frameworks, e a complexidade da configuração e do gerenciamento. Segundo Weerasinghe e Perera (2023), embora microserviços aumentem a modularidade, eles também impõem desafios significativos de orquestração e controle de dependências, especialmente quando envolvem tecnologias com contratos binários e requisitos mais rígidos, como o gRPC.

4.1.3. Adequação e Casos de Uso

Este critério avalia em quais tipos de aplicações e requisitos específicos cada tecnologia se sobressai, considerando sua maturidade no mercado, a disponibilidade de ferramentas de suporte e a relevância de sua comunidade para diferentes cenários de aplicação. Aspectos como a escalabilidade e a resiliência também serão considerados dentro da adequação para cenários específicos. De acordo com Tusa et al. (2024), em contextos como computação em borda, IoT e sistemas de análise em tempo real, tecnologias como o gRPC se destacam por sua escalabilidade, suporte a streaming bidirecional e menor consumo de recursos, o que é essencial em aplicações com forte sensibilidade à latência.

4.2. COMPARAÇÃO DAS TECNOLOGIAS

Nesta seção, cada abordagem de comunicação será analisada individualmente, aplicando os critérios de comparação definidos anteriormente para destacar suas características, vantagens e limitações.

4.2.1. REST em Microserviços

A arquitetura REST é amplamente adotada em ambientes de microserviços devido à sua simplicidade, adesão aos padrões da Web e interoperabilidade baseada no protocolo HTTP. Contudo, seu desempenho apresenta limitações relevantes quando se consideram métricas de latência e *throughput*. A comunicação REST envolve overheads substanciais decorrentes da serialização de dados em JSON, o que impacta o tempo de resposta em ambientes de alta concorrência. Detti, Funari e Petrucci (2023) demonstram, por meio da plataforma µBench, que sistemas RESTful exibem maior latência média do que aplicações monolíticas devido à fragmentação da lógica e à troca frequente de mensagens entre serviços. Essa limitação torna-se mais evidente em arquiteturas com alto número de chamadas síncronas, como evidenciado por Tam et al.

(2023), que apontam instabilidades de latência em fluxos REST sob cargas variáveis, especialmente em topologias com dependências profundas entre serviços.

Apesar de sua popularidade, a implementação de microserviços com REST exige atenção a uma série de desafios que afetam a complexidade do desenvolvimento. Luntovskyy e Shubyn (2020) destacam que, embora a introdução inicial de REST seja facilitada pela ampla disponibilidade de frameworks e bibliotecas (como Spring Boot e Express.js), a ausência de contratos fortemente tipados pode dificultar a automação de validações, integração contínua e testes. Isso se agrava em contextos onde a evolução dos serviços ocorre de forma descentralizada. Papapanagiotou e Chella (2018) também relatam que a latência acumulada em chamadas síncronas REST prejudica a adaptabilidade e dificulta a aplicação de estratégias de tolerância a falhas, exigindo ferramentas externas para *circuit breaking*, *retries* e monitoramento.

Ainda assim, REST continua sendo uma escolha sólida em uma ampla gama de cenários, especialmente aqueles em que a interoperabilidade, simplicidade e adoção massiva de ferramentas são determinantes. Usman et al. (2022) indicam que REST é amplamente suportado por ferramentas de observabilidade, *logging* e segurança, sendo amplamente utilizado em ambientes híbridos, edge e cloud, onde APIs públicas ou integráveis são comuns. Grambow et al. (2020) destacam que a modularidade e previsibilidade dos padrões REST favorecem o uso em portais externos, integrações com terceiros e aplicações voltadas à exposição de serviços de forma padronizada.

Em síntese, a arquitetura REST representa uma solução madura e consolidada para a construção de sistemas distribuídos baseados em microserviços. Sua adoção estratégica deve considerar as limitações técnicas impostas por seu modelo de comunicação, em especial no que tange à latência e à sobrecarga de mensagens. Entretanto, sua ampla aceitação na indústria, robusto ecossistema de suporte e curva de aprendizado reduzida tornam REST uma opção atraente quando requisitos de interoperabilidade, manutenção e padronização se sobrepõem a exigências extremas de desempenho.

4.2.2. gRPC em Microserviços

A arquitetura gRPC destaca-se por seu alto desempenho e baixa latência na comunicação entre microserviços, principalmente em ambientes de alta demanda. Diferente do modelo REST, que depende de comunicação textual via HTTP/1.1 e utiliza JSON para serialização, o gRPC opera sobre HTTP/2 e utiliza Protobuf (Protocol Buffers), que oferece compactação binária eficiente. Isso resulta em uma redução significativa no tempo de serialização e no volume de dados trafegados, contribuindo para uma comunicação até 7 vezes mais rápida que REST em cenários intensivos (Çiftçi e Çiloğlugil (2025)). Estudos recentes demonstram que gRPC atinge tempos de latência inferiores a 30ms em chamadas de ida e volta mesmo sob carga pesada, com excelente estabilidade de throughput (POPOVIC (2024)), o que o torna altamente

apropriado para arquiteturas distribuídas e sistemas de tempo real.

No entanto, essa eficiência técnica vem acompanhada de uma maior complexidade de implementação. O desenvolvimento com gRPC requer o uso do sistema de definição de interfaces .proto, cuja sintaxe e semântica são distintas das abordagens RESTful tradicionais. Isso impõe uma curva de aprendizado mais acentuada, especialmente para equipes acostumadas a arquiteturas baseadas em HTTP e JSON (Hedelin (2024)). A configuração do ecossistema gRPC também demanda atenção redobrada quanto à geração de código cliente-servidor em múltiplas linguagens e ao uso de bibliotecas específicas para suporte a comunicação multiplexada via HTTP/2. Embora ferramentas modernas e frameworks como Istio e Kubernetes ofereçam suporte nativo ao gRPC, sua integração e monitoramento exigem maior domínio técnico e controle sobre a infraestrutura (Rahman (2025)).

Quanto à adequação e aplicabilidade, o gRPC se sobressai em sistemas distribuídos com requisitos rigorosos de desempenho e escalabilidade, como plataformas financeiras, sistemas de comunicação em tempo real, jogos online, e serviços de machine learning em produção. Sua capacidade de suportar chamadas bidirecionais e streaming de dados em tempo real o diferencia positivamente em aplicações sensíveis à latência (Zeng et al. (2025)). Apesar de relativamente mais jovem que REST, o gRPC vem ganhando sólida maturidade no mercado, com ampla adoção em ambientes corporativos e suporte ativo da comunidade open source. Ferramentas modernas como Envoy e Istio oferecem integração nativa com gRPC, enquanto soluções como Prometheus e Jaeger permitem instrumentação eficiente para monitoramento e tracing distribuído (Rahman (2025)). Isso reforça sua adequação como uma solução resiliente e altamente escalável em arquiteturas de microserviços.

Em síntese, a adoção de gRPC em arquiteturas de microserviços deve considerar um balanço criterioso entre seus ganhos de desempenho e a complexidade de sua implementação. Seu uso é fortemente indicado quando a eficiência de comunicação entre serviços é um fator crítico, principalmente em contextos de alta concorrência e interatividade. Contudo, para organizações com menor maturidade técnica ou requisitos de interoperabilidade ampla com sistemas legados, a curva de adoção do gRPC pode representar um desafio operacional considerável (Ain et al. (2025)).

5 DISCUSSÃO E SÍNTESE COMPARATIVA

Nesta seção, são reunidas e sintetizadas as principais observações obtidas a partir da análise comparativa entre REST e gRPC. O objetivo é consolidar os pontos discutidos anteriormente, destacando de forma clara as vantagens, limitações e cenários de aplicação mais adequados para cada tecnologia. A síntese é apresentada por meio de um quadro comparativo, que facilita a visualização das diferenças e semelhanças

entre as abordagens, servindo como um recurso de apoio para a tomada de decisão em projetos que envolvem arquiteturas de microserviços.

5.1. QUADRO DE COMPARAÇÃO ENTRE REST E GRPC

Para consolidar as informações apresentadas, o quadro 1 oferece um resumo visual das principais características de cada tecnologia em relação aos critérios de comparação, servindo como uma referência rápida para o leitor.

Quadro 1 – Resumo Comparativo das Abordagens de Comunicação em Microserviços

Critério	**REST**	**gRPC**
Desempenho e Latência	Apresenta maior latência devido à sobrecarga do protocolo HTTP e da serialização de dados em JSON. O desempenho pode ser instável sob cargas variáveis e em topologias com muitas chamadas síncronas	Oferece alto desempenho e baixa latência, utilizando HTTP/2 e serialização binária com Protobuf, o que reduz o volume de dados trafegados. É significativamente mais rápido que REST em cenários intensivos.
Complexidade de Implementação	A implementação inicial é simples devido à vasta disponibilidade de bibliotecas e frameworks. No entanto, a ausência de contratos formais pode complicar a validação e a evolução segura das APIs	Exige uma curva de aprendizado mais acentuada devido à necessidade de definir contratos de interface em arquivos '.proto' e à complexidade do ecossistema. A integração e o monitoramento demandam maior domínio técnico.
Casos de Uso e Adequação	Ideal para cenários que exigem alta interoperabilidade, como APIs públicas e integrações com terceiros. Sua simplicidade e o amplo suporte de ferramentas o tornam uma escolha sólida para aplicações web e sistemas híbridos	Excelente para sistemas com requisitos rigorosos de desempenho e baixa latência, como plataformas financeiras, jogos online e comunicação em tempo real. Suporta streaming bidirecional, sendo adequado para aplicações sensíveis à latência.

5.1.1. Discussão dos Resultados

A escolha de uma tecnologia de comunicação em arquiteturas de microserviços é uma decisão estratégica que impacta diretamente o desempenho, a escalabilidade e a manutenibilidade do sistema. A análise comparativa entre REST e gRPC revela que não há uma solução universal, mas sim abordagens mais adequadas a contextos específicos.

5.1.1.1. Cenários Indicados para REST

A arquitetura REST se mantém como uma escolha pragmática e robusta em cenários onde a interoperabilidade e a simplicidade são mais críticas que o desempenho extremo. É a opção ideal para APIs públicas, que precisam ser facilmente consumidas por uma variedade de clientes e plataformas. Sua base no protocolo HTTP e o uso de JSON facilitam a integração e reduzem a barreira de entrada para desenvolvedores. A curva de aprendizado mais suave e o vasto ecossistema de ferramentas consolidadas tornam o REST uma aposta segura para equipes que buscam produtividade e um padrão de mercado amplamente aceito.

5.1.1.2. Cenários Indicados para gRPC

O gRPC é a escolha indicada para sistemas distribuídos onde a performance é um fator não negociável. Em cenários de comunicação interna entre microserviços, que exigem baixa latência e alto throughput, o gRPC se sobressai. Sua eficiência na serialização de dados com Protobuf e o uso de HTTP/2 permitem uma comunicação muito mais rápida que a do REST. É particularmente poderoso em aplicações que dependem de streaming de dados em tempo real, como em plataformas de IoT, sistemas financeiros e jogos online, onde a resposta imediata é crucial.

5.1.1.3. Desafios na Escolha da Tecnologia

A principal consideração na escolha entre REST e gRPC é o balanço entre os ganhos de desempenho e a complexidade de sua implementação. Embora o gRPC ofereça uma performance superior, ele impõe uma curva de aprendizado mais acentuada devido à necessidade de trabalhar com contratos de serviço fortemente tipados (.proto) e um ecossistema mais específico. Para equipes com menor maturidade técnica ou em projetos que precisam de integração com sistemas legados, a adoção do gRPC pode representar um desafio operacional significativo.

6 CONCLUSÃO

A análise realizada neste trabalho evidenciou que a escolha entre REST e gRPC em arquiteturas de microserviços não deve ser pautada apenas pela popularidade ou pela adoção de mercado, mas, sobretudo, pelas necessidades específicas de cada projeto e pela maturidade técnica da equipe envolvida. REST continua sendo uma solução consolidada e amplamente utilizada, destacando-se por sua simplicidade,

robusto ecossistema de suporte e alta interoperabilidade, o que o torna especialmente adequado para APIs públicas e integrações com sistemas heterogêneos. Por outro lado, o gRPC se apresenta como uma alternativa moderna e de alto desempenho, sendo mais indicado para cenários que exigem baixa latência, grande volume de requisições e suporte a comunicação bidirecional, como sistemas financeiros, jogos online e aplicações em tempo real.

A partir da comparação entre os critérios de desempenho, complexidade de implementação e adequação aos casos de uso, constatou-se que não há uma abordagem universalmente superior, mas sim soluções que se complementam em diferentes contextos. Dessa forma, recomenda-se uma visão estratégica, na qual REST e gRPC podem ser utilizados de maneira combinada, explorando as vantagens de cada tecnologia conforme o perfil do sistema em desenvolvimento.

REFERÊNCIAS

AIN, M. Z. et al. Comparative performance analysis of grpc and rest api under various traffic conditions and data sizes using a quantitative approach. *Journal of Applied Informatics and Computing*, v. 9, n. 2, p. 450–457, 2025. Citado na página 16.

BALALAIE, A.; HEYDARNOORI, A.; JAMSHIDI, P. Microservices architecture enables devops: Migration to a cloud-native architecture. *Ieee Software*, IEEE, v. 33, n. 3, p. 42–52, 2016. Citado na página 5.

BALALAIE, A. et al. Microservices migration patterns. *Software: Practice and Experience*, Wiley Online Library, v. 48, n. 11, p. 2019–2042, 2018. Citado na página 7.

CARTHEN, C. et al. Speciserve. a grpc infrastructure concept. In: IEEE. *2024 IEEE/ACIS 22nd International Conference on Software Engineering Research, Management and Applications (SERA)*. [S.l.], 2024. p. 273–276. Citado na página 11.

CHEN, J. et al. Remote procedure call as a managed system service. In: *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. [S.l.: s.n.], 2023. p. 141–159. Citado na página 11.

CHEN, L. Continuous delivery: Huge benefits, but challenges too. *IEEE software*, IEEE, v. 32, n. 2, p. 50–54, 2015. Citado na página 4.

ÇIFTÇI, B.; ÇILOĞLUGIL, B. A review of comparative studies on performance evaluation of communication mechanisms for microservices. In: SPRINGER. *International Conference on Computational Science and Its Applications*. [S.l.], 2025. p. 98–113. Citado na página 15.

CRUZ, J. d. S. Modelagem de microsserviços e arquiteturas de apis. 002, 2021. Citado na página 12.

DETTI, A.; FUNARI, L.; PETRUCCI, L. μbench: An open-source factory of benchmark microservice applications. *IEEE Transactions on Parallel and Distributed Systems*, IEEE, v. 34, n. 3, p. 968–980, 2023. Citado na página 14.

DRAGONI, N. et al. Microservices: yesterday, today, and tomorrow. *Present and ulterior software engineering*, Springer, p. 195–216, 2017. Citado 2 vezes nas páginas 4 e 7.

FIELDING, R. T. *Architectural styles and the design of network-based software architectures*. [S.l.]: University of California, Irvine, 2000. Citado na página 10.

FOWLER, M. *Software Architecture*. 2013. <https://martinfowler.com/architecture/>. Acessado em: 13 ago. 2025. Citado na página 7.

FOWLER, M. *Microservices: a definition of this new architectural term*. 2015. <https://martinfowler.com/articles/microservices.html>. Accessed: 2025-05-29. Citado na página 4.

FRANCESCO, P. D.; MALAVOLTA, I.; LAGO, P. Research on architecting microservices: Trends, focus, and potential for industrial adoption. In: IEEE. *2017 IEEE International conference on software architecture (ICSA)*. [S.l.], 2017. p. 21–30. Citado na página 4.

GARIMILLA, M. Microservices architecture: Revolutionizing modern software development. 2024. Citado na página 9.

GRAMBOW, M. et al. Benchmarking microservice performance: a pattern-based approach. In: *Proceedings of the 35th Annual ACM Symposium on Applied Computing*. [S.l.: s.n.], 2020. p. 232–241. Citado na página 15.

HEDELIN, M. N. *Benchmarking and performance analysis of communication protocols: A comparative case study of gRPC, REST, and SOAP*. [S.l.]: KTH Royal Institute of Technology, 2024. Citado 4 vezes nas páginas 10, 11, 13 e 16.

KAMISETTY, A. et al. Microservices vs. monoliths: Comparative analysis for scalable software architecture design. *Engineering International*, v. 11, n. 2, p. 99–112, 2023. Citado na página 7.

KATAL, A. et al. Evolution from monolithic to microservices architecture: A new era in software architecture. In: *Advancements in Optimization and Nature-Inspired Computing for Solutions in Contemporary Engineering Challenges*. [S.l.]: Springer, 2025. p. 235–279. Citado na página 8.

KONDREDDY, V. K.; RASTOGI, D. D. *Microservices and Automation Excellence: Full-Stack Development for the Intelligent Enterprise 2025*. [S.l.]: YASHITA PRAKASHAN PRIVATE LIMITED, 2025. Citado na página 8.

LI, S. et al. Understanding and addressing quality attributes of microservices architecture: A systematic literature review. *Information and software technology*, Elsevier, v. 131, p. 106449, 2021. Citado na página 8.

LUNTOVSKYY, A.; SHUBYN, B. Highly-distributed systems based on micro-services and their construction paradigms. In: IEEE. *2020 IEEE 15th International Conference on Advanced Trends in Radioelectronics, Telecommunications and Computer Engineering (TCSET)*. [S.l.], 2020. p. 7–14. Citado na página 15.

MALTSEV, E.; AMIN, R. U. Impact of serialization format on inter-service latency. *Advances in Cyber-Physical Systems*, v. 9, p. 89–94, 2024. Citado na página 13.

MASSE, M. *REST API design rulebook: designing consistent RESTful web service interfaces*. [S.l.]: "O'Reilly Media, Inc.", 2011. Citado na página 11.

NADAREISHVILI, I. et al. *Microservice architecture: aligning principles, practices, and culture*. [S.l.]: "O'Reilly Media, Inc.", 2016. Citado na página 5.

NEWMAN, S. *Building microservices: designing fine-grained systems*. [S.l.]: "O'Reilly Media, Inc.", 2021. Citado 2 vezes nas páginas 4 e 10.

PAPAPANAGIOTOU, I.; CHELLA, V. Ndbench: Benchmarking microservices at scale. *arXiv preprint arXiv:1807.10792*, 2018. Citado na página 15.

POPOVIC, M. Performance engineering of a microservice-based system. 2024. Citado na página 15.

RAHMAN, F. Cloud-native microservices for next-gen computing applications and scalable architectures. 2025. Citado na página 16.

RICHARDSON, L.; AMUNDSEN, M.; RUBY, S. *RESTful web APIs: services for a changing world*. [S.l.]: "O'Reilly Media, Inc.", 2013. Citado na página 10.

RODRÍGUEZ, G. et al. Agile infrastructure for cloud-based environments: A review. In: SPRINGER. *Information and Software Technologies: 27th International Conference, ICIST 2021, Kaunas, Lithuania, October 14–16, 2021, Proceedings 27*. [S.l.], 2021. p. 3–15. Citado na página 9.

SAID, M. A. et al. A systematic framework to enhance reusability in microservice architecture. *International Journal of Computing*, v. 17, n. 1, p. 1–18, 2025. Citado na página 9.

SULEIMAN, N.; MURTAZA, Y. Scaling microservices for enterprise applications: comprehensive strategies for achieving high availability, performance optimization, resilience, and seamless integration in large-scale distributed systems and complex cloud environments. *Applied Research in Artificial Intelligence and Cloud Computing*, v. 7, n. 6, p. 46–82, 2024. Citado na página 9.

TAIBI, D.; LENARDUZZI, V. On the definition of microservice bad smells. *IEEE software*, IEEE, v. 35, n. 3, p. 56–62, 2018. Citado na página 7.

TAIBI, D.; LENARDUZZI, V.; PAHL, C. Architectural patterns for microservices: a systematic mapping study. In: SCITEPRESS. *CLOSER 2018: Proceedings of the 8th International Conference on Cloud Computing and Services Science; Funchal, Madeira, Portugal, 19-21 March 2018*. [S.l.], 2018. Citado 2 vezes nas páginas 4 e 8.

TAM, D. S. H. et al. Pert-gnn: Latency prediction for microservice-based cloud-native applications via graph neural networks. In: *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. [S.l.: s.n.], 2023. p. 2155–2165. Citado na página 15.

THÖNES, J. Microservices. *IEEE software*, IEEE, v. 32, n. 1, p. 116–116, 2015. Citado na página 4.

TUSA, F. et al. Microservices and serverless functions—lifecycle, performance, and resource utilisation of edge based real-time iot analytics. *Future Generation Computer Systems*, Elsevier, v. 155, p. 204–218, 2024. Citado na página 14.

USMAN, M. et al. A survey on observability of distributed edge & container-based microservices. *IEEE Access*, IEEE, v. 10, p. 86904–86919, 2022. Citado na página 15.

WEERASINGHE, L.; PERERA, I. Evaluation of decomposition to microservices with optimized inter-service communication strategy. 2024. Citado na página 12.

WEERASINGHE, S.; PERERA, I. Optimized strategy for inter-service communication in microservices. *International Journal of Advanced Computer Science and Applications*, Science and Information (SAI) Organization Limited, v. 14, n. 2, 2023. Citado na página 14.

WILDE, E.; PAUTASSO, C. *REST: from research to practice*. [S.l.]: Springer Science & Business Media, 2011. Citado 2 vezes nas páginas 10 e 11.

ZANIEWSKI, P.; ROSZCZYK, R. law. Comparative review of selected internet communication protocols. 2024. Citado na página 11.

ZENG, G. et al. Sectest: An integrated testing platform for qos in satellite edge clouds. *IEEE Transactions on Services Computing*, IEEE, 2025. Citado na página 16.