**MATHEUS GUSTAVO COPPI DA SILVA**

# UMA ARQUITETURA BASEADA EM MICROSSERVIÇOS CENTRADA NA OBSERVABILIDADE

Balneário Camboriú
2025

**MATHEUS GUSTAVO COPPI DA SILVA**

# UMA ARQUITETURA BASEADA EM MICROSSERVIÇOS CENTRADA NA OBSERVABILIDADE

Trabalho de Conclusão de Curso apresentado ao curso de Sistemas de informação do Centro Universitário Avantis de Balneário Camboriú para a obtenção do título de Bacharel em Sistemas de informação.

Orientador: Prof. Luiz Fernando Arruda, Msc

Balneário Camboriú
2025

## RESUMO

A crescente complexidade dos sistemas e a demanda por escalabilidade impulsionaram a adoção da arquitetura de microsserviços em detrimento do modelo monolítico. Essa abordagem favorece a modularização, a escalabilidade granular e a autonomia das equipes de desenvolvimento, mas também impõe desafios operacionais e de observabilidade. Neste Trabalho de Conclusão de Curso, realiza-se uma comparação entre os protocolos de comunicação *REST* e *gRPC* em arquiteturas baseadas em microsserviços, utilizando métricas de observabilidade como latência (P95 e P99), *throughput* e uso de recursos. A metodologia envolve a implementação de dois serviços equivalentes, instrumentação com Prometheus, execução de testes de carga e análise estatística dos resultados. O objetivo é identificar o protocolo mais eficiente em diferentes cenários de uso. Os resultados esperados visam fornecer subsídios técnicos para orientar decisões arquiteturais em sistemas distribuídos, promovendo um equilíbrio entre desempenho e flexibilidade.



**Palavras-chave**: microsserviços. *REST*. *gRPC*.

## ABSTRACT

The increasing complexity of systems and the demand for scalability have driven the adoption of microservices architecture over the monolithic model. This approach promotes modularization, granular scalability, and development team autonomy, but also introduces operational and observability challenges. In this Undergraduate Thesis, a comparison is conducted between the communication protocols *REST* and *gRPC* in microservices-based architectures, using observability metrics such as latency (P95 and P99), throughput, and resource usage. The methodology involves implementing two equivalent services, instrumenting them with Prometheus, conducting load tests, and performing statistical analysis of the results. The objective is to identify the most efficient protocol in different usage scenarios. The expected results aim to provide technical insights to guide architectural decisions in distributed systems, promoting a balance between performance and flexibility.



**Keywords**: microservices. *REST*. *gRPC*.

## LISTA DE ILUSTRAÇÕES

Figura 1 – Estrutura típica de arquitetura monolítica. . . . . 15
Figura 2 – Arquitetura Orientada a Eventos, ilustrando a comunicação assíncrona entre microserviços através de um barramento de eventos. . . 17
Figura 3 – Arquitetura Serverless típica, mostrando os componentes principais: API Gateway, Funções como Serviço (FaaS), e serviços gerenciados. Fonte: Adaptado de (Shekhar, 2023). . . . . 18
Figura 4 – Funcionamento do padrão Circuit Breaker em sistemas distribuídos. 20
Figura 5 – Padrões fundamentais em arquitetura de microsserviços . . . . . 22
Figura 6 – Arquitetura REST: comunicação stateless entre cliente e servidor, utilizando métodos HTTP (GET, POST, PUT, DELETE) para manipulação de recursos identificados por URI. O diagrama destaca os princípios REST, exemplos de URIs e resposta JSON, evidenciando a interface uniforme e a ausência de estado. . . . . 26
Figura 7 – Padrões de streaming no gRPC: (A) Unário, (B) Servidor, (C) Cliente, (D) Bidirecional. . . . . 27

## LISTA DE TABELAS

Tabela 2 – Comparação de protocolos de comunicação . . . . . 21
Tabela 3 – Impacto da migração para microsserviços . . . . . 24
Tabela 4 – Operações REST e códigos HTTP . . . . . 25

## LISTA DE ABREVIATURAS E SIGLAS

| | |
|---|---|
| CPU | Central Processing Unit |
| GraphQL | Graph Query Language |
| gRPC | gRPC Remote Procedure Calls |
| HTTP | Hypertext Transfer Protocol |
| HTTP/2 | Hypertext Transfer Protocol version 2 |
| JSON | JavaScript Object Notation |
| REST | Representational State Transfer |
| URI | Uniform Resource Identifier |

**SUMÁRIO**


**1 INTRODUÇÃO . . . . . 9**
**1.1 OBJETIVOS . . . . . 11**
**1.1.1 Objetivo Geral . . . . . 11**
**1.1.2 Objetivos Específicos . . . . . 11**
**1.1.3 LIMITAÇÃO DO ESCOPO . . . . . 11**
**2 FUNDAMENTAÇÃO TEÓRICA . . . . . 12**
**2.1 PADRÕES ARQUITETURAIS . . . . . 14**
**2.1.1 Arquitetura Monolítica . . . . . 14**
**2.1.2 Event-Driven Architecture . . . . . 16**
**2.1.3 Serverless . . . . . 18**
**2.1.4 Microsserviços . . . . . 19**
2.1.4.1 Mecanismos de Comunicação . . . . . 21
2.1.4.2 Padrões Arquiteturais Essenciais . . . . . 22
2.1.4.3 Desafios Operacionais e Soluções . . . . . 23
2.1.4.4 Melhores Práticas de Implementação . . . . . 23
2.1.4.5 Casos de Estudo Relevantes . . . . . 23
2.1.4.6 Recomendações Estratégicas . . . . . 24
**2.1.5 Protocolos de Comunicação para Microsserviços . . . . . 24**
2.1.5.1 REST (Representational State Transfer) . . . . . 24
2.1.5.2 gRPC (gRPC Remote Procedure Calls) . . . . . 26
**REFERÊNCIAS . . . . . 29**

# 1 INTRODUÇÃO

Em poucos anos, a complexidade dos sistemas de software em ascenção e a necessidade de escalabilidade; flexibilidade e pulseira continua têm consequentemente impulsionado a propagação da arquitetura de microsserviços ao invés da tradicional arquitetura monolítica (NIAZI et al., 2020).

Por que a fragmentação de aplicações de serviços independentes, torna as aplicações mais flexíveis em termos de desenvolvimento, teste, implantação e expansão a se harmonizar mais rapidamente e de modo mais eficiente, tornando-se apto para times distribuídos com ciclos de entrega mais curtos (NIAZI et al., 2020).

A arquitetura de microsserviços apresenta diversas vantagens: a modularização em serviços independentes facilita a manutenção e evolução de componentes isolados (JAMSHIDI, 2016); a escalabilidade granular permite o dimensionamento seletivo de cada serviço conforme demanda, esta arquitetura se alinha perfeitamente com ambiente *cloud*, provendo escalabilidade, flexibilidade e resiliência (Gaurav Shekhar, 2023).

O ciclo de desenvolvimento é acelerado, pois equipes menores podem trabalhar em paralelo em serviços distintos sem necessariamente conhecer os outros serviços, ciclos de entrega mais rápidos pela habilidade de fazer *deploy updates* independente para cada *micro serviço* facilitando atualizações frequentes, além disso temos a questão da alta confiabilidade, falhas são isoladas, minimizando os impactos, por exemplo, um *crash* em um sistema de pagamentos não vai para outros componentes como a autenticação do usuário (Swapnil K Shevate, 2021).

Contudo, embora a adoção de microsserviços traga benefícios inegáveis, também impõe desafios significativos: A orquestração e o gerenciamento de múltiplos serviços demandam ferramentas como *Kubernetes*, elevando a sobrecarga operacional e a curva de aprendizado (JAMSHIDI, 2016); cada serviço necessita de uma infraestrutura adicional, como separação de banco de dados, ferramentas de monitoramento e *logging*. Além disso há uma dificuldade maior no *debbuging* da aplicação por conta dos *logs* serem distribuídos o que dificulta o *error tracing* e a resolução (Gaurav Shekhar, 2023).

Além disso, estudos como o de Niswar et al. [6] avaliam o impacto da escolha de protocolos de comunicação (*REST*, *gRPC*, *GraphQL*) sobre o desempenho de microsserviços, destacando a importância de decisões arquitetônicas bem-informadas para garantir uma observabilidade eficaz. Complementarmente, o trabalho de SHA et al. [7] investiga como *logs* de clusters *Kubernetes* podem ser usados para automatizar a análise de falhas em testes, evidenciando o papel crítico que a observabilidade desempenha na manutenção da qualidade de sistemas distribuídos.

Os três pilares fundamentais da observabilidade, conforme destacado por (Chi-

namanagonda, 2022), fornecem visibilidade integral sobre o comportamento de sistemas distribuídos.

O **monitoramento** acompanha continuamente o desempenho e disponibilidade do sistema através de métricas como utilização de CPU, consumo de memória e tempos de resposta. Como observado no MZ Computing Journal, ferramentas modernas transformam essa prática em estratégia proativa, alertando equipes sobre anomalias antes que escalem e mantendo a estabilidade operacional (Chinamanagonda, 2022).

O ***logging*** registra eventos sistêmicos detalhados - incluindo erros, advertências e informações operacionais - criando um histórico auditável. Em arquiteturas de microsserviços, onde serviços independentes geram volumosos fluxos de dados, sistemas centralizados de *logging* tornam-se indispensáveis para agregar e correlacionar informações distribuídas, viabilizando diagnóstico ágil de falhas (Chinamanagonda, 2022).

Já o ***tracing*** mapeia o fluxo transacional através de múltiplos serviços, expondo dependências e padrões de comunicação. Em ambientes distribuídos, este pilar identifica gargalos de desempenho e pontos críticos de falha no ciclo das requisições, sendo particularmente vital em microsserviços onde transações transversais exigem visibilidade topológica (Chinamanagonda, 2022). Conforme evidenciado no estudo, essa tríade evoluiu de abordagem reativa para modelo estratégico contínuo, capacitando resolução acelerada de incidentes em sistemas complexos.

O rastreamento distribuído complementa os pilares da observabilidade ao reconstruir o fluxo completo das requisições através de múltiplos serviços. Esta técnica mapeia jornadas transacionais complexas, revelando dependências ocultas e gargalos de desempenho que métricas isoladas não conseguem detectar (Chinamanagonda, 2022). Através da criação de linhas do tempo detalhadas, o rastreamento permite identificar pontos de falha e latência em cadeias de microsserviços, mesmo em ambientes altamente distribuídos. A sinergia entre métricas contínuas, registros contextualizados e mapas transacionais proporciona uma visão holística do sistema, essencial para a operação eficiente de arquiteturas modernas em escala.

A observabilidade consolida-se assim como catalisadora essencial na maturidade operacional de microsserviços. Conforme evidenciado por (Chinamanagonda, 2022) e (Sha et al., 2023), sua implementação estratégica converte dados distribuídos em inteligência acionável, permitindo: (1) manutenção proativa da estabilidade sistêmica; (2) diagnóstico preciso de falhas transacionais; (3) otimização contínua de desempenho. Este trabalho investiga como tais capacidades podem ser amplificadas mediante integração de padrões arquiteturais inovadores e ferramentas especializadas, respondendo aos desafios de complexidade destacados por (Shekhar, 2023).

## 1.1 OBJETIVOS

Diante disso, o presente Trabalho de Conclusão de Curso tem como objetivo geral comparar o desempenho dos protocolos de comunicação *REST* e *gRPC* em arquiteturas de microsserviços, utilizando métricas de observabilidade para avaliar latência (P95 e P99), *throughput* e uso de recursos.

### 1.1.1 Objetivo Geral

Comparar o desempenho dos protocolos de comunicação *REST* e *gRPC* em arquiteturas de microsserviços, utilizando métricas de observabilidade para avaliar latência, *throughput* e uso de recursos.

### 1.1.2 Objetivos Específicos

- Implementação de dois microserviços equivalentes, cada um empregando um dos protocolos em análise;
- Instrumentação dos serviços para expor métricas em uma rota, coletadas pelo prometheus;
- Execução de testes de carga controlados;
- Coleta e análise de métricas de latência, throughput e recursos pelo prometheus;
- Comparação estatística dos resultados para identificar o protocolo de melhor desempenho em diferentes cenários de carga.

### 1.1.3 LIMITAÇÃO DO ESCOPO

Este trabalho limita-se à comparação do desempenho entre os protocolos de comunicação *REST* e *gRPC* em dois serviços equivalentes implementados em uma arquitetura de *microsserviços*. A análise será conduzida em ambiente de testes controlado, com foco em métricas de observabilidade como latência (P95 e P99), *throughput* e uso de recursos.

Não serão abordados aspectos relacionados à segurança, autenticação, autorização, persistência de dados ou integração com sistemas legados. Da mesma forma, não se pretende avaliar outros protocolos de comunicação, como *GraphQL* ou *SOAP*, nem realizar testes em ambientes de produção. O uso de ferramentas como o *Prometheus* será restrito à coleta e análise de métricas, não sendo objetivo do estudo compará-las com outras soluções de monitoramento.

# 2 FUNDAMENTAÇÃO TEÓRICA

A arquitetura de software é um dos pilares centrais no desenvolvimento de sistemas computacionais modernos. Ela define a estrutura organizacional de um sistema, estabelecendo diretrizes para a disposição dos seus componentes, suas interações e restrições, além de orientar decisões técnicas que impactam diretamente na escalabilidade, manutenção e evolução do software ao longo do tempo (Jamshidi et al., 2016).

De acordo com (Jamshidi et al., 2016), a arquitetura de software pode ser compreendida como o conjunto de estruturas necessárias para raciocinar sobre o sistema, que compreende elementos de software, as relações entre eles e as propriedades de ambos. Em outras palavras, a arquitetura de software não se limita apenas à divisão do sistema em módulos, mas também envolve a definição de padrões de comunicação, protocolos, mecanismos de segurança, estratégias de escalabilidade e diretrizes para integração entre componentes.

A definição de uma arquitetura adequada é fundamental para garantir que o sistema atenda aos requisitos funcionais e não funcionais, como desempenho, segurança, confiabilidade e facilidade de manutenção. Segundo (Niazi et al., 2020), a escolha da arquitetura influencia diretamente a capacidade do sistema de evoluir e se adaptar a novas demandas, sendo um fator determinante para o sucesso de projetos de software em ambientes dinâmicos e competitivos.

Historicamente, a arquitetura monolítica foi o modelo predominante no desenvolvimento de sistemas. Nesse paradigma, todas as funcionalidades do software são implementadas em um único bloco, compartilhando o mesmo ambiente de execução, banco de dados e ciclo de vida (Niazi et al., 2020). Essa abordagem, embora simples de implementar e gerenciar em projetos de menor escala, apresenta limitações significativas à medida que o sistema cresce, como dificuldades de manutenção, escalabilidade restrita e maior risco de falhas generalizadas.

Com o avanço das demandas por sistemas mais flexíveis, escaláveis e resilientes, especialmente em ambientes de computação em nuvem, emergiu a arquitetura de microsserviços como uma alternativa ao modelo monolítico (Jamshidi et al., 2016; Shekhar, 2023). Os microsserviços propõem a fragmentação do sistema em serviços independentes, cada um responsável por uma funcionalidade específica e comunicando-se por meio de interfaces bem definidas. Essa abordagem permite que equipes distintas desenvolvam, testem e implantem serviços de forma autônoma, promovendo ciclos de entrega mais curtos e maior adaptabilidade às mudanças.

Além disso, a arquitetura de microsserviços facilita a adoção de práticas modernas de *DevOps*, automação de *deploys* e escalabilidade granular, alinhando-se às necessidades de negócios que exigem respostas rápidas e contínuas às mudanças

do mercado (Shekhar, 2023). No entanto, essa evolução também traz novos desafios, como a necessidade de ferramentas avançadas para orquestração, monitoramento e observabilidade, além de uma curva de aprendizado mais acentuada para as equipes de desenvolvimento (Jamshidi et al., 2016).

A definição de padrões arquiteturais é essencial para garantir a organização, escalabilidade e manutenção de sistemas de software. Diversos padrões foram consolidados na literatura internacional, cada um com características, vantagens e limitações específicas (Jamshidi et al., 2016).

Um dos padrões mais tradicionais é a arquitetura monolítica, na qual todas as funcionalidades do sistema são implementadas em um único bloco de código, compartilhando o mesmo ambiente de execução e banco de dados. Embora seja simples de desenvolver e implantar, esse modelo apresenta dificuldades de escalabilidade e manutenção à medida que o sistema cresce. Como afirmam (Niazi et al., 2020):

"A arquitetura monolítica é fácil de desenvolver e fazer *deploy*, mas, à medida que a aplicação cresce, torna-se difícil de gerenciar, escalar e manter. Qualquer pequena mudança requer que toda a aplicação seja reconstruída e reimplantada, o que aumenta o risco e o esforço."

Essa limitação faz com que, em projetos de maior porte, a arquitetura monolítica se torne menos atrativa, especialmente quando há necessidade de frequentes atualizações e escalabilidade do sistema.

A arquitetura em camadas é amplamente utilizada em sistemas corporativos. Nesse modelo, o sistema é dividido em diferentes camadas, como apresentação, lógica de negócio e acesso a dados, cada uma com responsabilidades bem definidas. Essa separação facilita a manutenção, a reutilização de componentes e a evolução do sistema, além de permitir que equipes distintas trabalhem de forma mais organizada (Jamshidi et al., 2016).

A *Service-Oriented Architecture* (*SOA*) propõe a construção de sistemas a partir de serviços independentes, que se comunicam por meio de protocolos padronizados, como *REST* ou *SOAP*. O *SOA* foi fundamental para a integração de sistemas heterogêneos e para a criação de soluções mais flexíveis, permitindo que diferentes aplicações compartilhem funcionalidades de forma segura e controlada. No entanto, a complexidade de orquestração e a dependência de barramentos de serviço podem tornar a implementação desse padrão mais desafiadora (Jamshidi et al., 2016).

Nos últimos anos, a arquitetura de microsserviços ganhou destaque, principalmente devido à sua capacidade de promover a independência entre equipes, a escalabilidade granular e a facilidade de adoção de práticas modernas de *DevOps*. Cada microsserviço é responsável por uma funcionalidade específica e pode ser desenvolvido, testado, implantado e escalado de forma autônoma. Essa flexibilidade, no entanto, exige um investimento maior em ferramentas de monitoramento, orquestração

e observabilidade, além de demandar uma curva de aprendizado mais acentuada para as equipes de desenvolvimento (Jamshidi et al., 2016; Shekhar, 2023; Niazi et al., 2020).

## 2.1 PADRÕES ARQUITETURAIS

### 2.1.1 Arquitetura Monolítica

A arquitetura monolítica caracteriza-se pela integração de todos os componentes funcionais do sistema - interface de usuário, lógica de negócio e acesso a dados - em uma única unidade de *deploy* e execução (Niazi et al., 2020). Nesse modelo, o acoplamento entre módulos é extremamente elevado, uma vez que compartilham o mesmo espaço de memória, base de dados e ciclo de vida. Conforme destacado por (Farhan et al., 2023), essa estrutura promove rapidez no desenvolvimento inicial devido à simplicidade de configuração e depuração, sendo ideal para protótipos ou sistemas com escopo limitado.

Entretanto, à medida que o sistema expande sua base de código e adquire maior complexidade, surgem desafios significativos que impactam diretamente a manutenção e a evolução da aplicação. Um dos principais entraves é a escalabilidade vertical obrigatória: a única estratégia viável consiste na replicação da aplicação inteira, mesmo quando apenas componentes específicos demandam mais recursos (Niazi et al., 2020). Essa limitação resulta em desperdício de recursos computacionais e aumento de custos operacionais, dificultando o crescimento sustentável do sistema.

Outro ponto crítico refere-se à rigidez tecnológica. A adoção de novas tecnologias ou versões de bibliotecas exige a atualização do monolito como um todo, o que gera *lock-in* tecnológico e limita a capacidade de inovação (Jamshidi et al., 2016). Esse cenário dificulta a experimentação e a incorporação de soluções modernas, tornando o sistema menos adaptável às mudanças do mercado e às demandas dos usuários.

Além disso, há o comprometimento da confiabilidade. Falhas em módulos secundários podem paralisar toda a aplicação, uma vez que todos os componentes compartilham o mesmo ambiente de execução. Testes de carga realizados por (Farhan et al., 2023) demonstram que a indisponibilidade de um único módulo pode resultar em *downtime* total, evidenciando a fragilidade desse modelo em relação à resiliência.

Por fim, destaca-se a complexidade evolutiva. Modificações em qualquer parte do sistema exigem reteste e reimplantação completos, ampliando os riscos e o *lead time* para a entrega de novas funcionalidades. Esse processo torna-se cada vez mais oneroso à medida que o sistema cresce, dificultando a manutenção e a evolução contínua da aplicação.

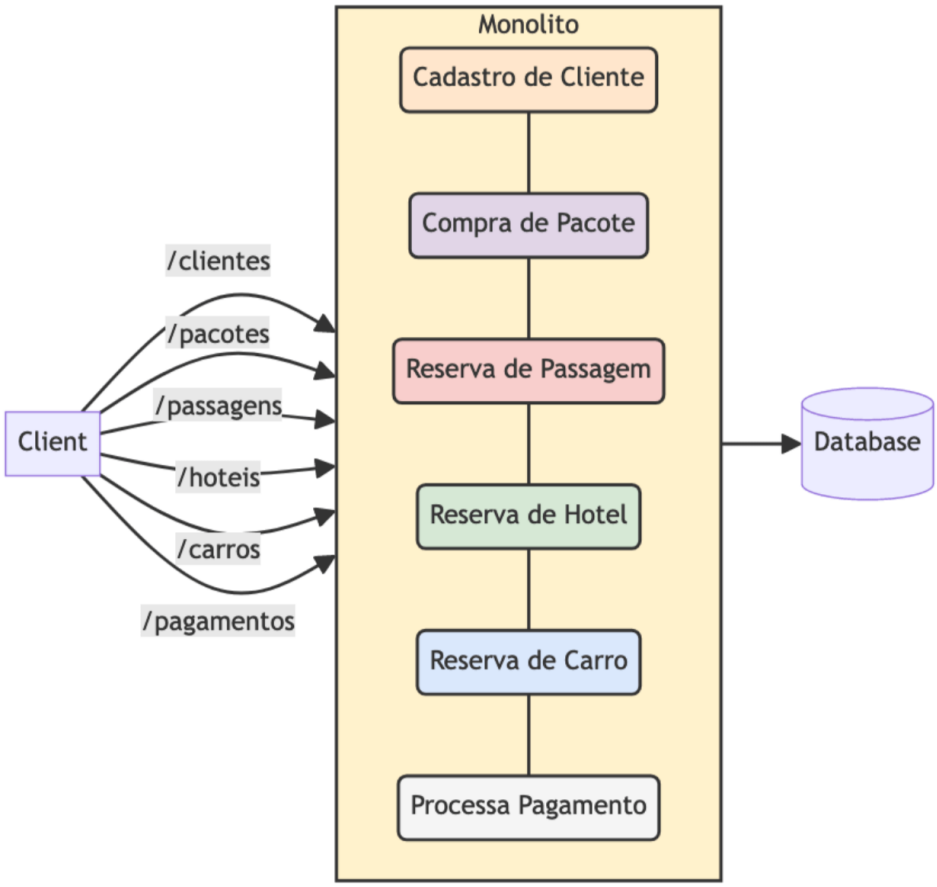


Figura 1 – Estrutura típica de arquitetura monolítica.

Como ilustrado na Figura 1, a arquitetura monolítica caracteriza-se pela centralização de todos os módulos e funcionalidades do sistema em uma única aplicação, onde as camadas como apresentação, lógica de negócio e acesso a dados encontram-se fortemente acopladas e compartilham o mesmo espaço de execução. O fluxo de requisições percorre camadas internamente acopladas, sem isolamento de processos. Essa centralização, embora simplifique o desenvolvimento inicial, apresenta limitações significativas em cenários de alta demanda e necessidade de escalabilidade.

Segundo Farhan et al. (Farhan et al., 2023), sistemas monolíticos apresentaram, em testes empíricos, 40% maior tempo de resposta sob carga crescente, 68% mais *downtime* durante atualizações e limite máximo de escalonamento em cinco instâncias, devido ao compartilhamento de recursos e à ausência de isolamento entre processos. Portanto, embora a arquitetura monolítica seja adequada para aplicações de menor porte ou com requisitos estáveis, suas restrições de escalabilidade, resiliência e flexibilidade tornam-se evidentes em ambientes dinâmicos e de grande escala, justificando a busca por alternativas arquiteturais mais modernas.

Portanto, embora a arquitetura monolítica seja adequada para aplicações de menor porte ou com requisitos estáveis, suas restrições de escalabilidade, resiliência e flexibilidade tornam-se evidentes em ambientes dinâmicos e de grande escala,

justificando a busca por alternativas arquiteturais mais modernas.

O fluxo de requisições percorre camadas internamente acopladas, sem isolamento de processos. Essa centralização explica os resultados empíricos de (Farhan et al., 2023), onde sistemas monolíticos apresentaram: O fluxo de requisições em sistemas monolíticos percorre camadas fortemente acopladas, sem qualquer tipo de isolamento entre processos. Essa centralização estrutural contribui para os resultados observados por (Farhan et al., 2023), que identificaram um aumento de 40% no tempo de resposta sob carga crescente, além de 68% mais *downtime* durante atualizações. Outro dado relevante é o limite máximo de escalonamento, que se restringe a cinco instâncias antes que ocorra degradação crítica do desempenho, evidenciando a dificuldade de adaptação a cenários de alta demanda.

Apesar dessas limitações, a arquitetura monolítica ainda mantém relevância em contextos específicos, como aplicações com tráfego previsível, equipes de desenvolvimento co-localizadas ou situações em que a simplicidade operacional é mais importante do que requisitos de elasticidade (Shekhar, 2023). No entanto, para sistemas empresariais modernos que exigem entrega contínua, resiliência e escalabilidade dinâmica, a viabilidade do modelo monolítico torna-se progressivamente reduzida, justificando a busca por alternativas arquiteturais mais flexíveis.

### 2.1.2 Event-Driven Architecture

A *Event-Driven Architecture* (EDA) é um paradigma arquitetural que se destaca pela capacidade de lidar com sistemas distribuídos, dinâmicos e de alta escalabilidade, sendo especialmente recomendada para aplicações que exigem resposta em tempo real e processamento assíncrono de grandes volumes de dados (Jamshidi et al., 2016). Nesse modelo, os componentes do sistema frequentemente implementados como microserviços comunicam-se por meio de eventos, que representam mudanças de estado ou ocorrências relevantes no domínio da aplicação. Cada evento é transmitido por um produtor e pode ser consumido por um ou mais consumidores, promovendo o desacoplamento entre os módulos e facilitando a evolução independente de cada serviço (Shekhar, 2023).

A adoção da EDA é comum em cenários como plataformas de *e-commerce*, sistemas financeiros, aplicações de *streaming* e ambientes de *Internet das Coisas* (IoT), onde a escalabilidade horizontal e a resiliência são requisitos críticos (Jamshidi et al., 2016; Shekhar, 2023). Nesses contextos, a arquitetura orientada a eventos permite que múltiplos serviços processem informações de forma paralela e reativa, reduzindo gargalos e aumentando a capacidade de resposta do sistema como um todo.

Segundo Jamshidi et al. (Jamshidi et al., 2016), a EDA contribui para a redução do acoplamento entre componentes, pois os serviços não dependem diretamente

uns dos outros, mas sim de eventos compartilhados por meio de um barramento ou intermediário, como um *message broker*. Essa característica facilita a manutenção, a escalabilidade e a resiliência, uma vez que falhas em um serviço não necessariamente impactam os demais, desde que o barramento de eventos permaneça operacional.

No entanto, a adoção da EDA traz desafios, como a complexidade na gestão do fluxo de eventos, a necessidade de garantir a ordem e a consistência das mensagens, além do aumento da dificuldade de depuração e testes em ambientes altamente distribuídos (Sha et al., 2023). Ainda assim, os benefícios em termos de escalabilidade, flexibilidade e resiliência tornam a arquitetura orientada a eventos uma escolha estratégica para sistemas modernos e de grande porte.

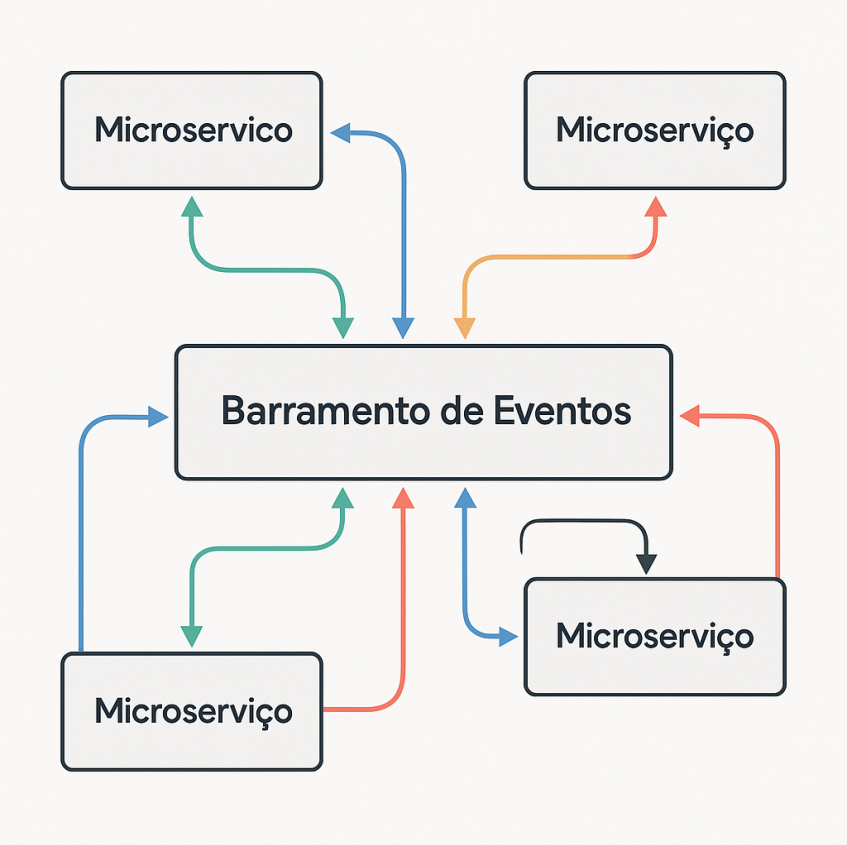


Figura 2 – Arquitetura Orientada a Eventos, ilustrando a comunicação assíncrona entre microserviços através de um barramento de eventos.

A Figura 2 apresenta um diagrama conceitual da Arquitetura Orientada a Eventos (*Event-Driven Architecture*). Nesta arquitetura, microserviços independentes comunicam-se de forma assíncrona através de um barramento de eventos central popularmente conhecido como *message broker*. Os microserviços podem tanto produzir eventos, enviando-os para o barramento, quanto consumir eventos, recebendo-os do barramento. Essa abordagem promove o desacoplamento entre os serviços, facilitando a escalabilidade, a resiliência e a manutenção do sistema.

A Figura 2 apresenta um diagrama conceitual da Arquitetura Orientada a Eventos (*Event-Driven Architecture*). Nesta arquitetura, microserviços independentes comunicam-se de forma assíncrona através de um barramento de eventos central popularmente

conhecido como *message broker*. Os microserviços podem tanto produzir eventos, enviando-os para o barramento, quanto consumir eventos, recebendo-os do barramento. Essa abordagem promove o desacoplamento entre os serviços, facilitando a escalabilidade, a resiliência e a manutenção do sistema.

### 2.1.3 Serverless

A arquitetura *serverless* tem se destacado com o avanço das plataformas de computação em nuvem, oferecendo uma abordagem inovadora para o desenvolvimento e a execução de aplicações. Nesse modelo, os desenvolvedores podem se concentrar exclusivamente na lógica de negócio, delegando a gestão da infraestrutura ao provedor de nuvem. Essa abstração permite maior agilidade no desenvolvimento e escalabilidade automática, embora apresente desafios relacionados à portabilidade e ao controle sobre o ambiente de execução (Shekhar, 2023).

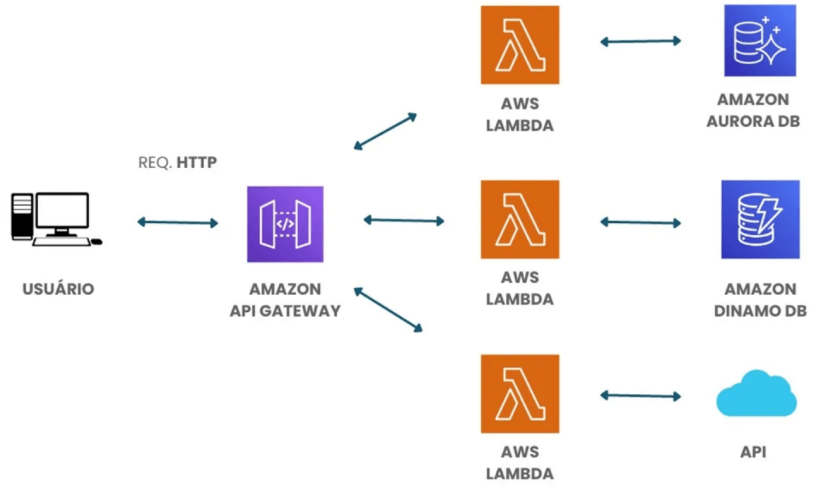


Figura 3 – Arquitetura Serverless típica, mostrando os componentes principais: API Gateway, Funções como Serviço (FaaS), e serviços gerenciados. Fonte: Adaptado de (Shekhar, 2023).

A Figura 3 ilustra um exemplo de arquitetura serverless, destacando seus componentes principais. O *API Gateway* atua como o ponto de entrada único para todas as requisições, roteando o tráfego para as funções de backend. As *Funções como Serviço (FaaS)*, como AWS Lambda ou Azure Functions, permitem a execução de código sob demanda, sem a necessidade de provisionar ou gerenciar servidores. Além disso, a arquitetura serverless utiliza *Serviços Gerenciados*, como bancos de dados e filas de mensagens, que são providos e mantidos pelo provedor de nuvem, simplificando ainda mais a operação e a manutenção da aplicação.

A adoção do padrão serverless elimina a necessidade de gerenciamento de servidores, permitindo que os desenvolvedores foquem exclusivamente na implementação

da lógica de negócios (Shekhar, 2023). A escolha do padrão arquitetural mais adequado depende de fatores como requisitos do sistema, contexto de negócio, equipe envolvida e recursos disponíveis. Compreender as vantagens e limitações de cada abordagem é essencial para alinhar a arquitetura às necessidades presentes e futuras do projeto (Jamshidi et al., 2016; Niazi et al., 2020).

### 2.1.4 Microsserviços

Como abordado anteriormente, a arquitetura de microsserviços decompõe sistemas em serviços autônomos. Sua implementação prática demanda atenção a três dimensões críticas: comunicação, observabilidade e orquestração. (Jamshidi et al., 2016; Niazi et al., 2020).

Entre os principais benefícios dos microsserviços está a escalabilidade granular, que permite dimensionar apenas os serviços que realmente demandam mais recursos, otimizando custos e desempenho (Shekhar, 2023). Além disso, a independência entre equipes é favorecida, pois times distintos podem desenvolver, testar e implantar serviços de forma autônoma, acelerando o ciclo de entrega e facilitando a adoção de práticas *DevOps* (Niazi et al., 2020; Farhan et al., 2023).

Outro ponto relevante é a resiliência: falhas em um serviço tendem a ser isoladas, reduzindo o impacto sobre o sistema como um todo (Farhan et al., 2023). Isso é especialmente importante em ambientes de missão crítica, onde a disponibilidade contínua é fundamental. Para garantir essa resiliência, padrões como o *Circuit Breaker* são frequentemente empregados.

A observabilidade em microsserviços demanda o uso de ferramentas especializadas para coleta e análise de métricas, logs e rastreamento distribuído (Madupati, 2023; Sha et al., 2023). Essa visibilidade é indispensável para identificar gargalos, diagnosticar falhas e garantir a saúde do sistema em produção. Soluções como Prometheus e Jaeger são amplamente utilizadas para monitorar o desempenho e rastrear o fluxo de requisições entre serviços, facilitando a manutenção e a evolução contínua da arquitetura.

Outro ponto relevante é a resiliência: falhas em um serviço tendem a ser isoladas, reduzindo o impacto sobre o sistema como um todo (Farhan et al., 2023). Isso é especialmente importante em ambientes de missão crítica, onde a disponibilidade contínua é fundamental. Para garantir essa resiliência, padrões como o *Circuit Breaker* são frequentemente empregados.

Por fim, a orquestração de microsserviços envolve o gerenciamento automatizado de implantação, escalonamento e atualização dos serviços, frequentemente utilizando plataformas como Kubernetes (Shekhar, 2023). Essa abordagem permite maior resiliência operacional, reduz o tempo de indisponibilidade e simplifica a administração de ambientes complexos. A automação dos processos de entrega contínua e

recuperação de falhas é um dos pilares para o sucesso de arquiteturas baseadas em microsserviços.

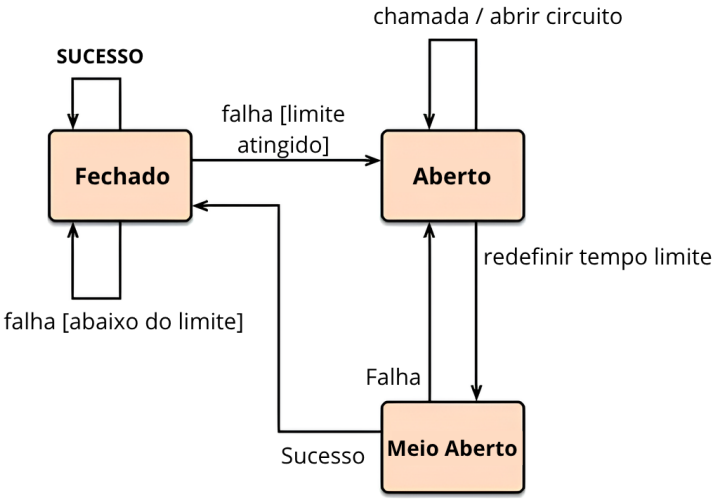


Figura 4 – Funcionamento do padrão Circuit Breaker em sistemas distribuídos.

O *Circuit Breaker* é um padrão arquitetural amplamente utilizado em sistemas distribuídos para aumentar a resiliência e evitar falhas em cascata (Nair et al., 2019; Fowler, 2012). Ele atua como um "interruptor"que monitora as chamadas a um serviço externo. Seu funcionamento é dividido em três estados principais: fechado, aberto e meio-aberto.

No estado fechado, todas as requisições são encaminhadas normalmente ao serviço. Se o número de falhas ultrapassar um limite configurado, o circuito entra no estado aberto, bloqueando novas tentativas e retornando imediatamente uma resposta de erro ou acionando um mecanismo de *fallback* uma solução alternativa, como dados em cache ou uma resposta padrão, para manter a aplicação funcionando de forma degradada. Após um tempo de espera, o circuito passa para o estado meio-aberto, permitindo algumas requisições de teste. Se essas requisições forem bem-sucedidas, o circuito retorna ao estado fechado; caso contrário, volta ao estado aberto, protegendo o sistema de sobrecarga e falhas sucessivas (Nair et al., 2019; Fowler, 2012).

Esse padrão é fundamental para garantir a robustez de sistemas baseados em microsserviços, pois impede que falhas em um serviço se propaguem e comprometam toda a aplicação. Estudos recentes destacam a importância do *Circuit Breaker* em arquiteturas resilientes, especialmente em ambientes de alta disponibilidade e missão crítica (Nair et al., 2019).

No entanto, a adoção de microsserviços também traz desafios consideráveis. A complexidade operacional aumenta, exigindo ferramentas avançadas para orquestração (como *Kubernetes*), monitoramento e observabilidade (Jamshidi et al., 2016;

Shekhar, 2023). Cada serviço pode demandar sua própria infraestrutura, banco de dados e mecanismos de autenticação, o que eleva a sobrecarga de gerenciamento (Niazi et al., 2020).

A observabilidade torna-se um aspecto central, pois a identificação e resolução de falhas em sistemas distribuídos requerem coleta e análise de métricas, *logs* e *traces* distribuídos (Madupati, 2023; Sha et al., 2023). Ferramentas como *Prometheus* e *Jaeger* são amplamente utilizadas para monitorar o desempenho e rastrear requisições entre serviços, facilitando o diagnóstico de problemas e a manutenção da qualidade do sistema (Ahmed et al., 2022).

Além disso, a escolha dos protocolos de comunicação entre microsserviços (*REST*, *gRPC*, *GraphQL*) impacta diretamente o desempenho, a flexibilidade e a observabilidade do sistema (Niswar et al., 2023). Estudos recentes destacam que decisões arquiteturais bem fundamentadas são essenciais para garantir a eficiência e a robustez de sistemas baseados em microsserviços (Niswar et al., 2023; Sha et al., 2023).

A arquitetura de microsserviços representa uma evolução paradigmática no desenvolvimento de software, decompondo sistemas complexos em serviços autônomos especializados. Conforme (Jamshidi et al., 2016), esta abordagem oferece:

- Autonomia tecnológica: Cada serviço pode utilizar linguagens e tecnologias distintas (Java, Python, Node.js)
- Resiliência aprimorada: Isolamento de falhas através de padrões como *Circuit Breaker*
- Entrega contínua: *Deploys* independentes permitindo múltiplas implantações diárias

#### 2.1.4.1 Mecanismos de Comunicação

A comunicação entre serviços em arquiteturas de microsserviços pode ser realizada por diferentes mecanismos, cada um com características, vantagens e limitações específicas. A escolha do protocolo de comunicação é um fator determinante para o desempenho, a escalabilidade e a complexidade do sistema, influenciando diretamente a experiência do usuário e a eficiência operacional (Niswar et al., 2023; Maso, 2024).

Tabela 2 – Comparação de protocolos de comunicação

| Protocolo | Latência | Complexidade | Casos de Uso |
|---|---|---|---|
| REST/HTTP | Alta | Baixa | Sistemas com baixa frequência |
| gRPC | Baixa | Média | Microserviços acoplados |
| Eventos (Kafka) | Assíncrona | Alta | Sistemas escaláveis |

Conforme demonstrado na Tabela 2, a seleção do protocolo de comunicação deve considerar o perfil de uso e os requisitos de cada aplicação. O protocolo

REST/HTTP é amplamente adotado devido à sua simplicidade e compatibilidade com diferentes plataformas, sendo indicado para cenários em que a frequência de requisições não é elevada e a interoperabilidade é prioridade (Fielding, 2000; Amazon Web Services, 2023). Por outro lado, o gRPC destaca-se pela baixa latência e eficiência na serialização de dados, tornando-se uma escolha adequada para integrações entre microsserviços que exigem alto desempenho e comunicação síncrona (Niswar et al., 2023; IBM, 2025).

Já os mecanismos baseados em eventos, como o uso de filas e *brokers* (exemplo: Kafka), possibilitam comunicação assíncrona e desacoplamento entre os serviços, favorecendo a escalabilidade horizontal e a resiliência do sistema (Jamshidi et al., 2016; Shekhar, 2023). Essa abordagem é especialmente recomendada para sistemas que processam grandes volumes de dados ou que demandam alta disponibilidade, pois permite que os serviços operem de forma independente, reduzindo o risco de falhas em cascata.

Além dos aspectos técnicos, a complexidade de implementação e manutenção também deve ser considerada. Protocolos como REST tendem a ser mais simples de configurar e depurar, enquanto soluções baseadas em eventos exigem infraestrutura adicional e monitoramento especializado para garantir a integridade e a ordem das mensagens (Maso, 2024; Madupati, 2023). Dessa forma, a decisão sobre o mecanismo de comunicação deve ser pautada por uma análise criteriosa das necessidades do projeto, visando sempre o equilíbrio entre desempenho, escalabilidade e simplicidade operacional.

#### 2.1.4.2 Padrões Arquiteturais Essenciais

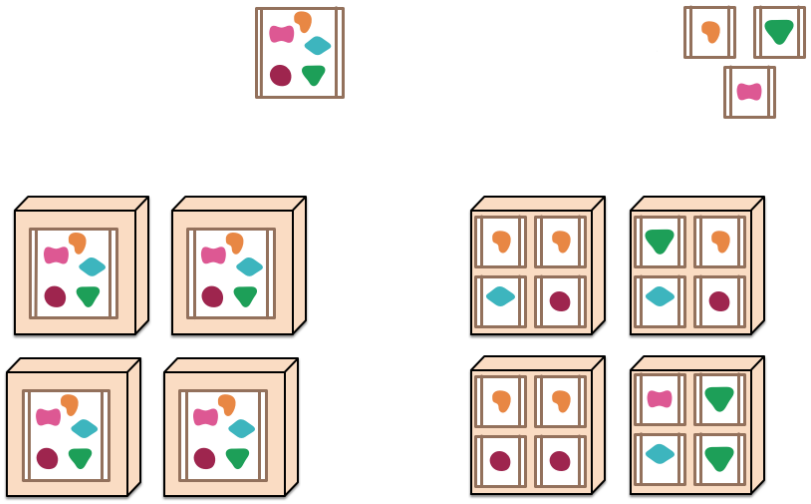

Figura 5 – Padrões fundamentais em arquitetura de microsserviços

A Figura 5 ilustra as diferenças fundamentais entre a arquitetura monolítica e a arquitetura de microsserviços. À esquerda, observa-se que uma aplicação monolítica

agrupa todas as funcionalidades em um único processo. Para escalar esse modelo, é necessário replicar toda a aplicação em múltiplos servidores, o que pode gerar desperdício de recursos e limitações de flexibilidade.

À direita, a arquitetura de microsserviços distribui cada funcionalidade em processos independentes, permitindo que cada serviço seja desenvolvido, implantado e escalado separadamente. Isso proporciona maior eficiência no uso de recursos, facilita a manutenção e possibilita o crescimento gradual do sistema conforme a demanda por funcionalidades específicas.

Essa abordagem modular e distribuída é um dos pilares das arquiteturas modernas voltadas à escalabilidade, resiliência e agilidade no desenvolvimento.

#### 2.1.4.3 Desafios Operacionais e Soluções

A implementação prática enfrenta obstáculos significativos:

- Observabilidade: Requer integração de métricas, *logs* e *traces* com ferramentas como:
  - *Prometheus* para monitoramento
  - *Jaeger* para rastreamento distribuído
  - *EFK Stack* (Elasticsearch, Fluentd, Kibana) para análise
- Orquestração: Plataformas como *Kubernetes* gerenciam:
  - Escalonamento automático
  - Balanceamento de carga
  - Recuperação de falhas

#### 2.1.4.4 Melhores Práticas de Implementação

Experiências de empresas líderes revelam estratégias eficazes:

- Amazon: Utiliza a prática do "time da pizza de dois"equipes pequenas e independentes, responsáveis por serviços autônomos do início ao fim.
- Netflix: Introduziu a cultura do *Chaos Engineering*, simulando falhas em produção para garantir a resiliência dos serviços.
- Spotify: Estruturou suas equipes em *squads* responsáveis por domínios funcionais, promovendo independência e entrega contínua.
- Uber: Implementou uma hierarquia de serviços baseada em criticidade, diferenciando serviços centrais de auxiliares para otimizar a operação.

#### 2.1.4.5 Casos de Estudo Relevantes

Resultados empíricos demonstram impactos mensuráveis:

Tabela 3 – Impacto da migração para microsserviços

| Métrica | Antes | Depois |
|---|---|---|
| Tempo de entrega | 30 dias | 2 dias |
| Disponibilidade | 92% | 99.95% |
| Custo de infraestrutura | $100k/mês | $65k/mês |

Conforme dados na Tabela 3, organizações reportam melhorias significativas após transição bem-sucedida (Farhan et al., 2023; Shekhar, 2023).

#### 2.1.4.6 Recomendações Estratégicas

A adoção deve considerar:

- Pré-requisitos: Maturidade em DevOps e cultura de automação
- Critérios: Complexidade sistêmica > 50k LOC > 15 desenvolvedores
- Alternativas: Arquiteturas híbridas para transição gradual
- Anti-padrões: Decomposição excessiva ("nanoserviços")

Em síntese, microsserviços oferecem vantagens transformacionais para sistemas complexos, mas exigem investimento proporcional em automação e governança (Niazi et al., 2020; Madupati, 2023).

### 2.1.5 Protocolos de Comunicação para Microsserviços

A seleção de protocolos de comunicação é um fator crítico em arquiteturas de microsserviços, influenciando diretamente o desempenho, acoplamento e capacidade de evolução do sistema. Dois padrões predominantes neste contexto são REST e gRPC, cada um com características técnicas distintas que os tornam adequados a cenários específicos (Niswar et al., 2023).

#### 2.1.5.1 REST (Representational State Transfer)

O REST, formalizado por Fielding (Fielding, 2000) em sua tese seminal, é um estilo arquitetural baseado em princípios de recursos identificáveis por URI e operações padronizadas via verbos HTTP (GET, POST, PUT, DELETE). Sua adoção generalizada deve-se a:

- Simplicidade: Utiliza formatos legíveis como JSON, facilitando a depuração e a integração entre sistemas heterogêneos. No entanto, a serialização textual pode consumir de 30% a 50% mais banda em comparação com protocolos binários (Niswar et al., 2023).
- *Stateless*: Cada requisição carrega todo o contexto necessário, eliminando a necessidade de manter estado no servidor e favorecendo a escalabilidade horizontal (Fielding, 2000).

- Interface Uniforme: Princípios como a identificação única de recursos via URI e o uso de HATEOAS (*Hypermedia as the Engine of Application State*) garantem consistência e desacoplamento. Com HATEOAS, as respostas da API incluem hipermídia (links) que orientam o cliente sobre as próximas ações possíveis, promovendo navegação dinâmica e reduzindo o acoplamento entre cliente e servidor (Maso, 2024).
- Compatibilidade: Possui suporte nativo em navegadores e ampla adoção em APIs públicas, facilitando a integração com diferentes plataformas (Maso, 2024).

Limitações em cenários complexos:

- Overhead de comunicação: Serialização textual aumenta latência e consumo de CPU, especialmente em payloads grandes (Niswar et al., 2023).
- Modelo síncrono: Suporta apenas comunicação unária (request/response), inviabilizando fluxos assíncronos (Niswar et al., 2023).
- Falta de contratos rígidos: Validação de esquemas depende de implementações adicionais (Maso, 2024).

Operações Básicas e Códigos HTTP: A Tabela 4 sintetiza o mapeamento CRUD em REST, conforme padrões industriais (Fielding, 2000):

Tabela 4 – Operações REST e códigos HTTP

| Método HTTP | Ação | Código Sucesso |
|---|---|---|
| GET | Ler recurso | 200 (OK) |
| POST | Criar recurso | 201 (Created) |
| PUT/PATCH | Atualizar recurso | 200 (OK) ou 204 (No Content) |
| DELETE | Excluir recurso | 204 (No Content) |

Fonte: Adaptado de (Niswar et al., 2023) e (Fielding, 2000).

A adoção do REST em arquiteturas de microsserviços proporciona diversos benefícios, especialmente pela padronização da comunicação e pela facilidade de integração entre componentes heterogêneos. O uso de URIs para identificação de recursos, aliado à interface uniforme baseada em métodos HTTP, simplifica o desenvolvimento e a manutenção dos serviços, permitindo que diferentes equipes trabalhem de forma independente e escalável (Fielding, 2000; Maso, 2024).

Além disso, a independência de estado (*stateless*) facilita o balanceamento de carga e a replicação dos serviços, pois qualquer instância do servidor pode processar requisições sem depender de informações armazenadas em sessões. Essa característica é fundamental para ambientes distribuídos e de alta disponibilidade, comuns em sistemas modernos baseados em microsserviços.

Por outro lado, é importante considerar que, apesar de sua simplicidade e ampla adoção, o REST pode apresentar limitações em cenários que exigem comunicação

em tempo real, operações transacionais complexas ou validação rígida de contratos. Nesses casos, pode ser necessário complementar a arquitetura com outros padrões ou protocolos, como gRPC ou eventos assíncronos, para atender a requisitos específicos de desempenho e flexibilidade (Niswar et al., 2023).

Dessa forma, a arquitetura REST permanece como uma das principais escolhas para a construção de APIs robustas, escaláveis e de fácil integração, sendo amplamente utilizada tanto em aplicações web quanto em ambientes de microsserviços.

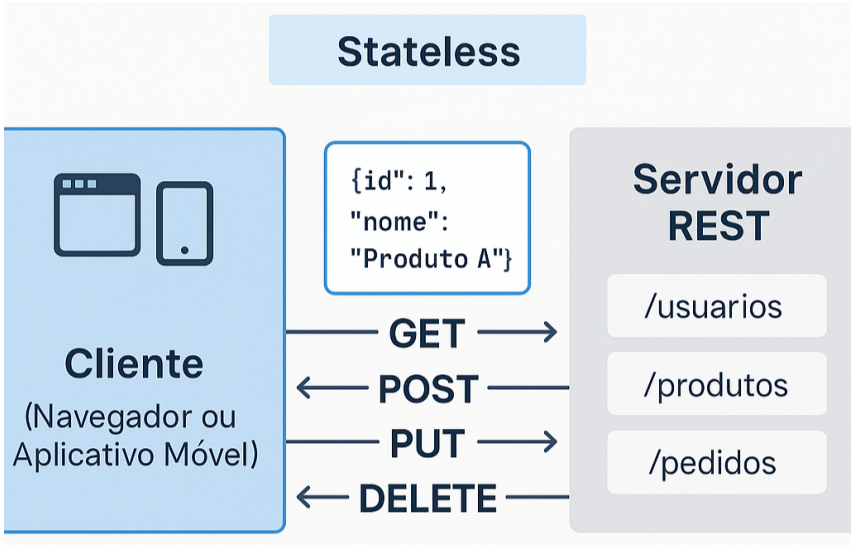


Figura 6 – Arquitetura REST: comunicação stateless entre cliente e servidor, utilizando métodos HTTP (GET, POST, PUT, DELETE) para manipulação de recursos identificados por URI. O diagrama destaca os princípios REST, exemplos de URIs e resposta JSON, evidenciando a interface uniforme e a ausência de estado.

A Figura 6 ilustra a arquitetura REST, evidenciando a troca de mensagens entre cliente e servidor por meio de métodos HTTP padronizados. Os recursos são identificados por URIs específicas, e as respostas são geralmente estruturadas em formato JSON. O princípio stateless garante que cada requisição seja independente, promovendo escalabilidade e simplicidade na integração entre sistemas.

#### 2.1.5.2 gRPC (gRPC Remote Procedure Calls)

O gRPC é um *framework* aberto de alta performance para chamadas de procedimento remoto (RPC), desenvolvido pelo Google e lançado em 2015, que utiliza HTTP/2 e *Protocol Buffers* para comunicação binária eficiente (Niswar et al., 2023; Maso, 2024). Sua adoção em arquiteturas de microsserviços deve-se a diferenciais como desempenho superior, suporte a streaming bidirecional, contratos fortemente tipados e geração automática de código.

- Desempenho superior: A serialização binária via *Protocol Buffers* reduz o tamanho dos *payloads* e a latência em relação a APIs REST tradicionais, conforme mensurações em ambientes controlados (Niswar et al., 2023).

- Streaming bidirecional: Suporte nativo a fluxos contínuos de dados (cliente-servidor, servidor-cliente e bidirecional), habilitando comunicação assíncrona para aplicações em tempo real (Maso, 2024).
- Contratos fortemente tipados: Definição explícita de serviços e estruturas de mensagens em arquivos `.proto`, garantindo compatibilidade e evolução controlada de APIs (Shekhar, 2023).
- Geração de código automática: Compilação de *stubs* clientes e servidores em múltiplas linguagens de programação a partir de arquivos `.proto`, promovendo consistência e redução de erros (Shekhar, 2023).

Apesar das vantagens, alguns desafios são observados:

- Acoplamento forte: Alterações nos contratos `.proto` exigem atualização sincronizada de clientes e servidores, aumentando a complexidade de evolução em sistemas distribuídos (Shekhar, 2023).
- Curva de aprendizagem: A definição de contratos e manipulação de fluxos de *streaming* pode ser complexa, especialmente para desenvolvedores familiarizados com paradigmas REST (Maso, 2024).

A Figura 7 apresenta os quatro padrões de comunicação suportados pelo gRPC, cada um adequado a diferentes cenários de integração entre serviços distribuídos.

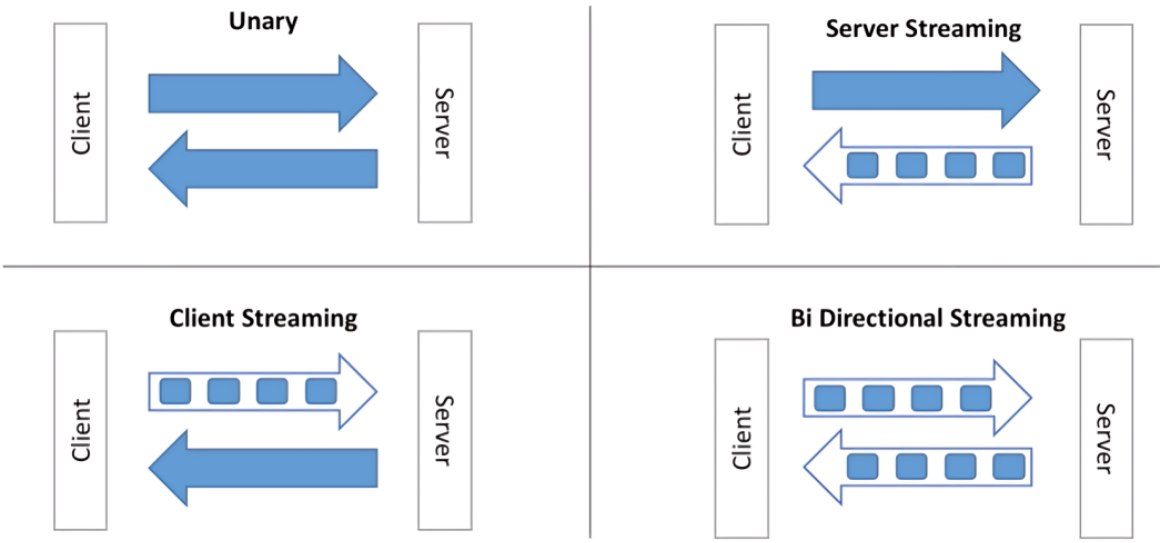


Figura 7 – Padrões de streaming no gRPC: (A) Unário, (B) Servidor, (C) Cliente, (D) Bidirecional.

No padrão *unary* (A), o cliente realiza uma chamada simples, enviando uma única requisição ao servidor e recebendo uma única resposta. Esse modelo é o mais próximo das operações tradicionais de APIs REST, sendo indicado para transações pontuais e de baixa complexidade, como consultas ou comandos diretos.

O padrão *server streaming* (B) ocorre quando o cliente envia uma requisição e o servidor responde com um fluxo contínuo de mensagens. Esse padrão é útil em

situações em que o servidor precisa retornar uma grande quantidade de dados ou fornecer atualizações progressivas, como em relatórios extensos ou notificações em tempo real.

No padrão *client streaming* (C), o cliente envia um fluxo de mensagens ao servidor, que processa todas as informações recebidas e retorna uma única resposta ao final. Esse modelo é apropriado para cenários em que múltiplos dados precisam ser enviados de forma agrupada, como uploads de arquivos ou envio de lotes de registros.

Por fim, o padrão *bidirectional streaming* (D) permite que tanto o cliente quanto o servidor enviem múltiplas mensagens de forma independente e assíncrona, compartilhando um canal de comunicação contínuo. Esse padrão é especialmente poderoso para aplicações que exigem troca dinâmica de dados em tempo real, como chats, jogos online ou monitoramento de sensores, proporcionando máxima flexibilidade e desempenho na comunicação entre microsserviços (Niswar et al., 2023; Shekhar, 2023).

Vantagens Técnicas Comprovadas: Estudos empíricos demonstram ganhos operacionais mensuráveis (Niswar et al., 2023):

- Redução de 35-45% no consumo de CPU durante serialização;
- Até 7x maior throughput em comunicações inter-serviços.

# REFERÊNCIAS

AHMED, M. et al. Observability in Kubernetes Cluster: Automatic Anomalies Detection using Prometheus. **Procedia Computer Science**, 2022.

AMAZON WEB SERVICES. **Diferenças entre gRPC e REST**. [S.l.*:* s.n.], 2023. Acesso em: 31 maio 2025. Disponível em: https://aws.amazon.com/pt/compare/the-difference-between-grpc-and-rest/.

CHINAMANAGONDA, Sandeep. Observability in Microservices Architectures -Advanced observability tools for microservices environments. **MZ Research Journal**, 2022.

FARHAN, M. et al. **Performance Comparison between Monolith and Microservice Using Docker and Kubernetes**. [S.l.*:* s.n.], 2023.

FIELDING, Roy Thomas. **Architectural Styles and the Design of Network-based Software Architectures**. 2000. Tese (Doutorado) – University of California, Irvine, EUA. Tese de Doutorado.

FOWLER, Martin. **Circuit Breaker**. [S.l.*:* s.n.], 2012. https://martinfowler.com/bliki/CircuitBreaker.html.

IBM. **gRPC vs. REST**. [S.l.*:* s.n.], 2025. Acesso em: 31 maio 2025. Disponível em: https://www.ibm.com/think/topics/grpc-vs-rest.

JAMSHIDI, P. et al. A Systematic Mapping Study in Microservice Architecture. **Institute of Electrical and Electronics Engineers**, 2016.

MADUPATI, Bhanuprakash. **Observability in Microservices Architectures: Leveraging Logging, Metrics, and Distributed Tracing in Large-Scale Systems**. [S.l.*:* s.n.], 2023.

MASO, Nicolas Nascimento. Comparativo entre arquiteturas de APIs: REST, GraphQL e gRPC*.* In*:* REPOSITÓRIO UFSC. [S.l.*:* s.n.], 2024. Trabalho de Conclusão de Curso.

NAIR, S. et al. A survey on circuit breaker pattern in microservices architecture. **International Journal of Recent Technology and Engineering**, 2019.

NIAZI, M. et al. The Comparison of Microservice and Monolithic Architecture*.* In*:* 2020 International Conference on Computing, Electronic and Electrical Engineering (ICE Cube). [S.l.*:* s.n.], 2020.

NISWAR, M. et al. Performance evaluation of microservices communication with REST, GraphQL, and gRPC. **Journal of Systems and Software**, 2023.

SHA, K. et al. Automating Microservices Test Failure Analysis using Kubernetes Cluster Logs. **IEEE Access**, 2023.

SHEKHAR, Gaurav. **Microservices Design Patterns for Cloud Architecture**. [S.l.]: IEEE Chicago, 2023.