Sistemas Distribuídos

Aula 9 – Microsserviços

Prof. Ana Carolina Sokolonski

Bacharelado em Sistemas de Informação
Instituto Federal da Bahia – Campus Feira de Santana

2026

Agenda

Do Monólito aos Microsserviços

O que são Microsserviços?

Comunicação entre Serviços

Padrões de Resiliência

Implantação e Orquestração

Observabilidade

Vantagens, Desafios e Quando Usar

Referências

Do Monólito aos Microsserviços

A evolução das arquiteturas de software

Arquitetura Monolítica — O Ponto de Partida

O que é um Monólito?

Aplicação onde **todos os módulos** (interface, lógica de negócio, acesso a dados) são construídos e implantados como **uma única unidade**. Um único processo, um único binário, um único banco de dados.

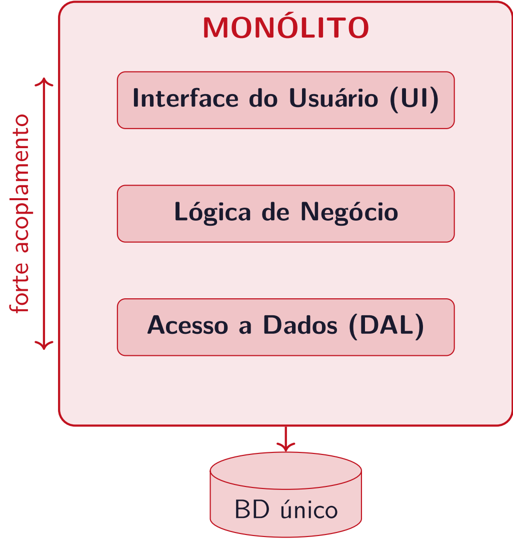
Quando funciona bem

- Times pequenos, projeto inicial
- Baixa complexidade de domínio
- Implantação simples e rápida
- Debugging e testes mais diretos

Problemas à medida que cresce

- **Acoplamento total**: mudar um módulo quebra outros
- **Escala ineficiente**: escala-se a aplicação inteira
- **Deploy arriscado**: qualquer mudança reimplanta tudo
- **Tecnologia única**: stack presa ao início do projeto

Arquitetura Monolítica — O Ponto de Partida



Monólito vs. Microsserviços — Comparação

Monólito

- Uma unidade de deploy
- Um único banco de dados
- Comunicação in-process (chamadas de função)
- Escala vertical (máquinas maiores)
- Time único gerencia tudo
- Testes de integração simples
- Falha afeta toda a aplicação

Microsserviços

- Múltiplas unidades de deploy independentes
- Cada serviço tem seu próprio banco
- Comunicação via rede (HTTP/gRPC/mensagens)
- Escala horizontal por serviço
- Times pequenos e autônomos por serviço
- Testes de integração distribuídos
- Falha isolada (circuit breaker)

O que são Microsserviços?

Definição, princípios e características

Definição e Princípios

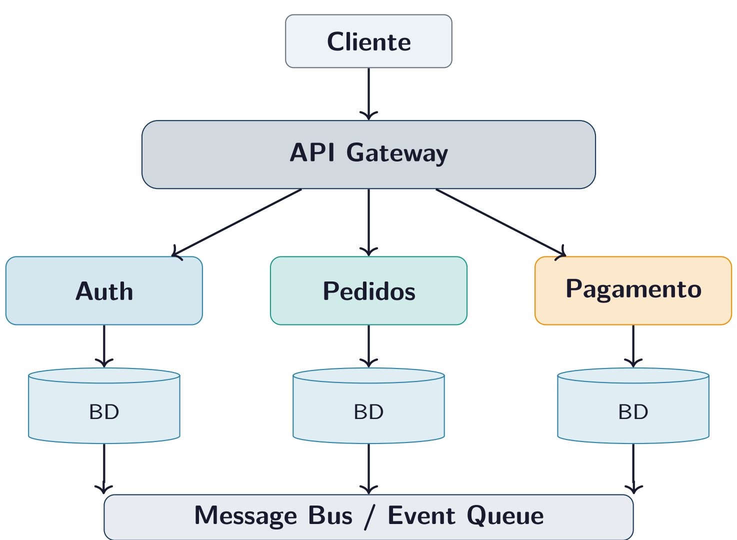
Definição (Martin Fowler, 2014)

Estilo arquitetural que estrutura uma aplicação como uma **coleção de serviços pequenos e independentes**, cada um executando em seu próprio processo e comunicando-se via mecanismos leves (geralmente HTTP/REST ou mensageria). Cada serviço é **implantado independentemente** e organizado em torno de **capacidades de negócio**.

Princípios fundamentais

1. **Responsabilidade única**: cada serviço faz uma coisa bem
2. **Banco de dados por serviço**: sem compartilhamento de dados
3. **Falha isolada**: um serviço caindo não derruba o sistema
4. **Deploy independente**: sem coordenar com outros times
5. **Descentralização**: tecnologias diferentes por serviço

Definição e Princípios



Características dos Microsserviços

▪ Organização por negócio

Serviços refletem **bounded contexts** do domínio (DDD). Ex: Pedidos, Estoque, Notificações, Pagamentos.

▪ Produtos, não projetos

O time dono do serviço o **mantém em produção**. "You build it, you run it" (Amazon). Responsabilidade total.

▪ Design para falha

Qualquer chamada de rede pode falhar. Serviços devem implementar **Circuit Breaker**, **Retry**, **Timeout** e **Fallback**.

▪ Pologlotismo

Cada serviço pode usar a linguagem e tecnologia mais adequada: Java, Python, Go, Node.js — sem restrição de stack.

▪ Endpoints inteligentes

Lógica nos serviços, não no barramento. Pipes simples: REST, gRPC, AMQP. Contrário do ESB corporativo.

▪ Evolução independente

Versionar APIs, manter compatibilidade retroativa. Deploy sem coordenação com outros serviços.

Comunicação entre Serviços

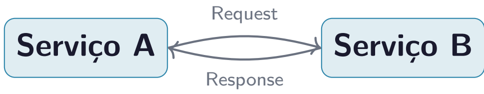
Síncrona, assíncrona e padrões de integração

▪ Comunicação Síncrona

O serviço chamador **aguarda a resposta** antes de continuar. Simples de implementar e debugar.

REST / HTTP: padrão web, stateless, fácil de consumir.
gRPC: binário, eficiente, tipado (Protocol Buffers), ideal para comunicação interna.

Problema: acoplamento temporal se B cai, A falha.



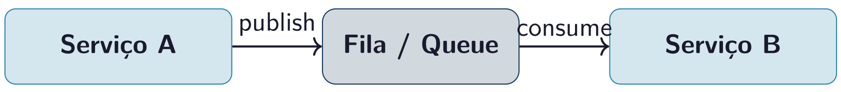
bloqueia até receber

▪ Comunicação Assíncrona

O serviço **publica uma mensagem** e não espera resposta. Desacopla temporalmente os serviços.

Message Brokers: RabbitMQ, Apache Kafka, AWS SQS.
Event-Driven: serviços reagem a eventos publicados.

Vantagem: resiliência — B pode estar offline e processar depois.
Desvantagem: consistência eventual, debugging mais difícil.



Desacoplamento temporal

Padrões de Integração — API Gateway e Service Mesh

API Gateway

Ponto único de entrada para todos os clientes. Responsável por:

- Roteamento de requisições
- Autenticação e autorização (JWT, OAuth2)
- Rate limiting e throttling
- Composição de respostas (BFF)
- SSL termination e load balancing

Ex: Kong, AWS API Gateway, NGINX, Traefik.

Service Mesh

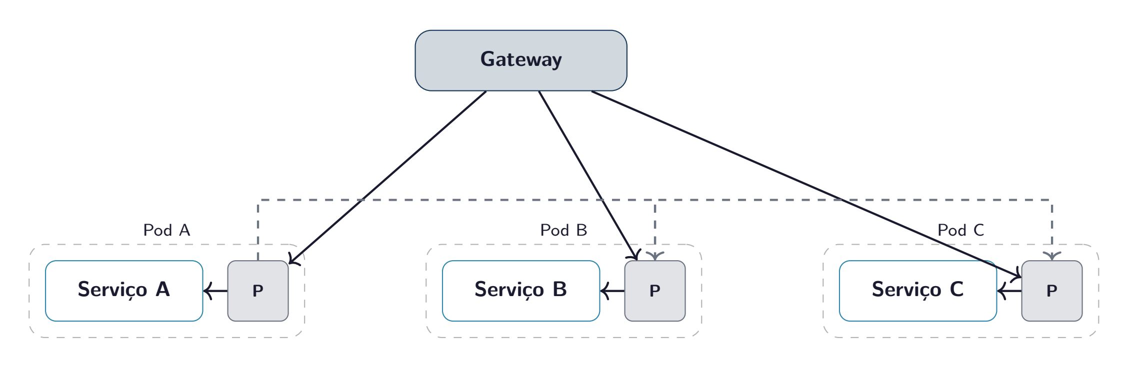
Infraestrutura de comunicação **transparente** entre microsserviços, implementada como **sidecar proxies** ao lado de cada serviço.

Fornece automaticamente:

- Descoberta de serviços
- Circuit breaker e retry
- Observabilidade (traces, métricas)
- mTLS entre serviços

Ex: Istio, Linkerd, Consul Connect.

Padrões de Integração — API Gateway e Service Mesh



P = Sidecar Proxy (Service Mesh)

Padrões de Resiliência

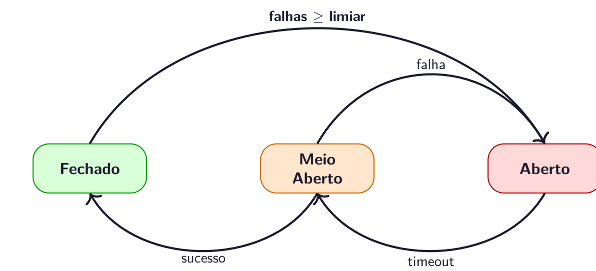
Circuit Breaker, Saga e mais

▪ Circuit Breaker

Inspirado no disjuntor elétrico. **Três estados**:

- **Fechado**: chamadas passam normalmente
- **Aberto**: chamadas bloqueadas (serviço falhou), retorna fallback
- **Meio-aberto**: testa se serviço se recuperou

Evita que falhas em cascata derrubem todo o sistema.
Ex: Netflix Hystrix, Resilience4j.



Padrão Saga

Gerencia **transações distribuídas** sem 2PC. Divide uma transação longa em etapas locais com **transações compensatórias** em caso de falha.

Coreografia: cada serviço reage a eventos (event-driven).
Orquestração: um orquestrador central coordena os passos.

Ex.: pedido → reserva de estoque → cobrança → entrega. Se cobrança falha: cancela reserva.

Outros padrões importantes

- **Retry**: tentar novamente com backoff exponencial
- **Timeout**: não esperar indefinidamente
- **Bulkhead**: isolar recursos por serviço
- **Health Check**: monitorar saúde continuamente

Implantação e Orquestração

Docker, Kubernetes e CI/CD

Contêineres e Orquestração

Por que contêineres?

Cada microsserviço roda em seu próprio **contêiner Docker**: ambiente isolado, portável e reprodutível.

Vantagens: mesma imagem em dev, teste e produção; inicialização rápida; menor overhead do que VMs.

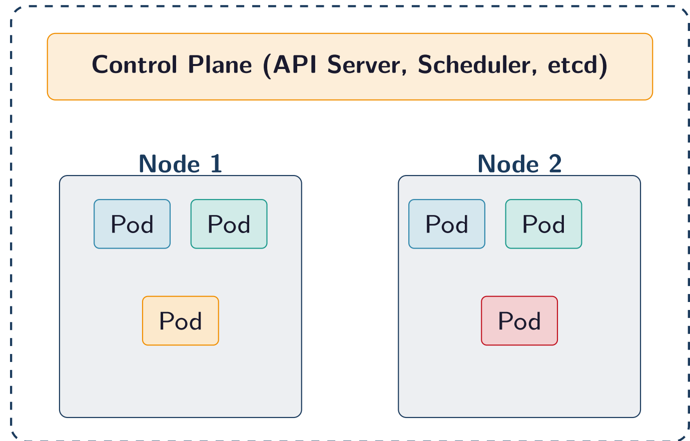
CI/CD

Cada serviço tem seu próprio **pipeline independente**: commit → build → test → deploy automatizado.

Kubernetes (K8s)

Plataforma de **orquestração de contêineres** que gerencia:

- **Scheduling**: onde cada contêiner executa
- **Auto-scaling**: aumenta/reduz réplicas por demanda
- **Self-healing**: reinicia contêineres com falha
- **Rolling update**: atualiza sem downtime
- **Service discovery**: endereços dinâmicos
- **Load balancing**: distribui tráfego entre pods


Cluster Kubernetes
Control Plane (API Server, Scheduler, etcd)
Node 1
Pod
Pod
Pod
Node 2
Pod
Pod
Pod

Observabilidade

Logs, Métricas e Rastreamento Distribuído

Os Três Pilares da Observabilidade

Logs

Registros de eventos discretos. Em microsserviços, devem ser **estruturados** (JSON) e **centralizados**.

Desafio: correlacionar logs de 50 serviços para uma única requisição do usuário.

Ex: ELK Stack (Elasticsearch, Logstash, Kibana), Loki.

Métricas

Dados numéricos ao longo do tempo: latência, taxa de erros, uso de CPU, requisições/segundo.

Permitem alertas e dashboards em tempo real.

Ex: Prometheus + Grafana, Datadog, New Relic.

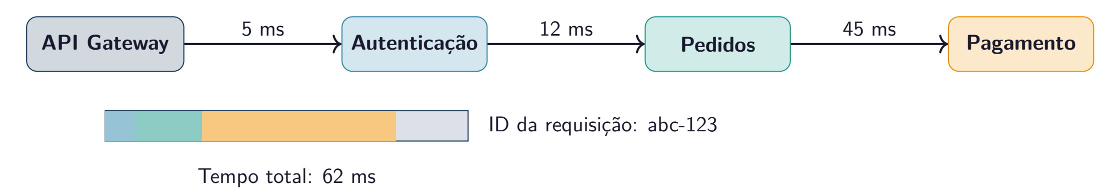
Rastreamento Distribuído

Acompanha uma requisição através de **todos os serviços** que ela percorre, calculando latência em cada hop.

Cada req. recebe um **trace ID** único propagado nos headers.

Ex: Jaeger, Zipkin, AWS X-Ray.

Os Três Pilares da Observabilidade



Vantagens, Desafios e Quando Usar

Microsserviços não são bala de prata

Vantagens e Desafios

Vantagens

- **Escala independente**: só o serviço sobrecarregado cresce
- **Deploy contínuo**: mudanças frequentes e seguras
- **Resiliência**: falha isolada, não cascata
- **Times autônomos**: squads pequenas e focadas
- **Tecnologia heterogênea**: linguagem certa para cada problema
- **Reutilização**: serviços consumidos por múltiplos clientes

Desafios (não são gratuitos!)

- **Complexidade operacional**: dezenas de serviços para gerenciar
- **Latência de rede**: chamadas entre serviços são mais lentas
- **Consistência de dados**: sem transações ACID distribuídas simples
- **Testes de integração**: ambiente complexo de replicar
- **Debugging**: rastrear erros entre serviços é difícil
- **Overhead**: infraestrutura muito mais cara e complexa

Quando Usar e Quando NÃO Usar

Quando usar

Times grandes, domínio complexo, necessidade de escala independente, múltiplos canais de entrega (web, mobile, parceiros).

Quando NÃO usar

Times pequenos, produto em estágio inicial, baixa complexidade de domínio. Comece com monólito bem estruturado.

Exemplos Reais — Quem usa microsserviços?

▪ Netflix

Pioneiro na adoção. Mais de **700 microsserviços**. Criou o Circuit Breaker (Hystrix), o Chaos Monkey e Zuul (gateway). Processa bilhões de requisições diárias.

▪ Amazon

Originou o modelo *"you build it, you run it"*. Cada serviço AWS é autônomo. O e-commerce usa centenas de microsserviços para checkout, recomendações, gestão de estoque, entre outros.

▪ Brasil: Nubank

Construído em Clojure e com microsserviços desde o início. Mais de 300 serviços, Kafka para mensageria. Referência nacional em arquitetura distribuída.

Microsserviços $\neq$ microserviços pequenos

O tamanho certo é o que uma equipe pequena pode manter

Referências I

- FOWLER, M.; LEWIS, J. Microservices. *martinfowler.com*, 2014. Disponível em: https://martinfowler.com/articles/microservices.html
- NEWMAN, S. *Building Microservices*. 2. ed. O'Reilly, 2021.
- RICHARDSON, C. *Microservices Patterns*. Manning Publications, 2018.
- TANENBAUM, A. S.; VAN STEEN, M. *Distributed Systems: Principles and Paradigms*. 3. ed. Pearson, 2017.
- BURNS, B. *Designing Distributed Systems*. O'Reilly, 2018.