Microservices: Definition, Benefits, and Trade-Offs

Servicios y Aplicaciones Distribuidas

Context

- Monolith strengths met growth limits.
- Need for team autonomy and independent releases.
- Microservices emerged as an organizational & technical response.

What Is a Microservice?

- Small, independently deployable service.
- Owns a narrowly scoped capability / bounded context.
- Loosely coupled, highly cohesive; communicates over the network.

Monolith vs Microservices

- Monolith: one deployable; fewer moving parts.
- Microservices: many deployables; higher autonomy.
- Trade-off: simplicity vs independent evolution.

Principle: Bounded Context

- Use domain-driven design (DDD) to define boundaries.
- Each service owns a coherent domain capability.
- Interfaces between contexts are explicit and stable.

Independent Deployability

- Each service ships on its own cadence.
- Avoid shared databases and shared libraries that force lockstep.
- Contracts and backward compatibility are essential.

Organizational Alignment

- Team structure mirrors system structure.
- Service ownership maps to long-lived teams.
- Cognitive load must match team capacity.

Microservices vs SOA

- Smaller units, stricter autonomy than traditional SOA.
- Operational culture: DevOps, CI/CD, automation.
- Lightweight protocols vs heavy ESB-centric designs.

Communication Styles Overview

- Synchronous: REST, gRPC; simple request/response.
- Asynchronous: events, queues, streams; decoupled timing.
- Hybrid patterns blend both.

Synchronous Comms: REST

- Human-friendly, widely supported, cacheable over HTTP.
- Great for CRUD and resource-centric APIs.
- Beware multi-hop latency and cascading failures.

Synchronous Comms: gRPC

- Contract-first with protobuf; strong typing and speed.
- Bi-directional streaming, efficient binary transport.
- Couples clients to schemas—manage versioning carefully.

Asynchrony: Messaging & Events

- Queues (task distribution) and topics (pub/sub).
- Decouples producers and consumers; smooths traffic spikes.
- Requires idempotency and explicit ordering choices.

Data Ownership: per Service?

- Each service owns its schema and storage.
- Avoid shared databases that create hidden coupling.
- Integrate via APIs/events, not cross-service SQL.

Consistency & CQRS Basics

- Accept eventual consistency between services.
- CQRS separates writes (commands) and reads (queries).
- Projections/read models serve low-latency queries.

Sagas & Distributed Transactions

- Long-lived workflows coordinated via messages.
- Compensating actions instead of global 2-phase commit.
- Orchestration vs choreography styles.

API Gateways

- Single entry point for external clients.
- Cross-cutting concerns: authN/Z, rate limiting, TLS.
- Request shaping: routing, aggregation, protocol translation.

Service Discovery

- Dynamic environments require discovery.
- Registries or DNS + health checks.
- Config and secrets management are foundational.

Observability in Microservices

- Three pillars: logs, metrics, traces.
- Correlate requests across services (trace IDs).
- SLOs and error budgets guide priorities.

Testing Strategies

- Contract tests to protect APIs between teams.
- Pyramid: unit ≫ component/integration ≫ end-to-end.
- Test data and environments must be automatable.

CI/CD per Service

- Pipeline per service with clear promotion stages.
- Canary and blue-green deployments reduce risk.
- Automate rollbacks; keep artifacts immutable.

Security Fundamentals

- Zero-trust: authenticate and authorize every request.
- mTLS for service-to-service encryption.
- Secret management and least privilege.

Service Mesh (Brief Overview)

- Offload retries, timeouts, and mTLS.
- Uniform telemetry and policy enforcement.
- Beware added complexity—adopt when ready.

Containers & Orchestration

- Containers package runtime; orchestration manages fleet.
- Kubernetes: scheduling, scaling, self-healing.
- Declarative manifests and controllers.

Platform

- Platform team provides paved roads (golden paths).
- Templates, scaffolding, and guardrails reduce toil.
- Self-service + sensible defaults speed teams.

Costs & Trade-Offs

- Operational overhead: more services, more things to run.
- Latency and partial failures are everyday realities.
- People costs: skills, on-call, coordination.

Common Anti-Patterns

- Nanoservices: splitting too far, chatty networks.
- Shared database across 'services' (hidden coupling).
- Logic in the gateway/ESB recreating a central bottleneck.

When NOT to Use Microservices

- Small team, simple domain, low scale.
- Unclear boundaries or volatile requirements.
- Lack of platform/observability maturity.

Migration Strategies: Strangler

- Wrap the monolith; route new capabilities to services.
- Gradually replace parts behind stable interfaces.
- Continuously measure progress and outcomes.

Migration: Domain Extraction

- Identify seams via DDD context maps and change cadence.
- Extract independent or painful domains first.
- Establish contract and data ownership before cutover.

Case Study: E-Commerce Split

- Orders, Payments, Catalog, Users, Notifications.
- Independent scaling and release cadences.
- Clear contracts and resilience patterns.

Case Study: Lessons Learned

- Invest early in observability and platform tooling.
- Keep boundaries aligned to business outcomes.
- Resist premature decomposition and tech sprawl.

Discussion Prompts

- Which domains in your system change most frequently?
- Where are teams blocked by centralized releases?
- What platform gaps would slow a microservices migration?

Key Takeaways

- Microservices trade simplicity for autonomy and speed.
- Bounded contexts and independent deployability are non-negotiable.
- Success requires platform maturity and discipline.

Closing & Next Session

- Preview: Communication deep-dive (sync vs async).
- Hands-on lab setup reminder.
- Q&A and reading list.