4. Microservices Architecture

Introduction

Microservices architecture is an approach to building software as a **collection of small, independent services** that each run in their own process and communicate through lightweight mechanisms (usually HTTP APIs). Each service is built around a specific business capability, can be deployed independently, and may use different programming languages and databases.

Imagine a restaurant where instead of one kitchen handling everything, you have specialized stations: a sushi bar, a pasta station, a dessert kitchen, and a grill—each operating independently, with their own tools and staff, but coordinating to serve complete meals to customers.

This architectural style emerged in the early 2010s, popularized by companies like Netflix, Amazon, and Uber who faced challenges that monolithic architectures couldn't solve. However, **microservices solve specific problems at specific scales**—they're not a universal solution and come with significant complexity.

The name "microservices" is somewhat misleading. The "micro" doesn't mean the services are tiny (a few lines of code), but rather that each service has a **focused, well-defined purpose**. A better term might be "focused services" or "bounded-context services."

References

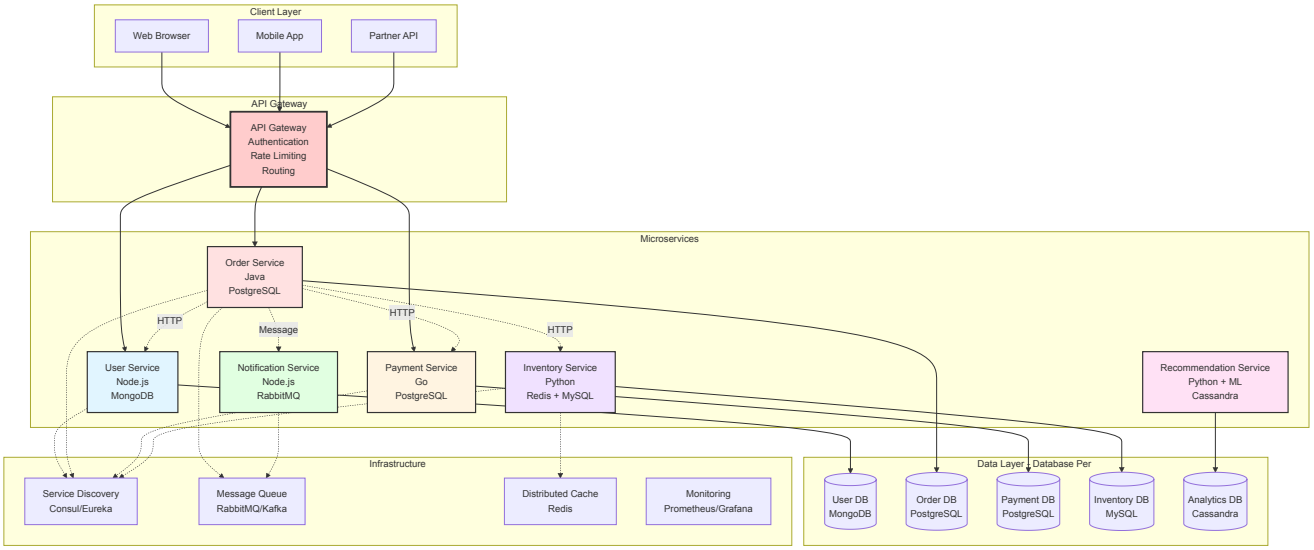
1. Newman, S. (2015). *Building Microservices: Designing Fine-Grained Systems.* O’Reilly Media.
2. Richardson, C. (2020). *Microservices Patterns: With Examples in Java* (2nd ed.). Manning Publications.
3. Fowler, M., & Lewis, J. (2014). *Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith.* O’Reilly Media.

Explanation

Core Concept

Microservices architecture is defined by several key characteristics:

1. **Independent Deployability**: Each service can be deployed without deploying others
2. **Business Capability Focus**: Services organized around business functions, not technical layers
3. **Decentralized Data**: Each service manages its own database
4. **Technology Diversity**: Different services can use different tech stacks
5. **Communication via APIs**: Services interact through network calls (REST, gRPC, messaging)
6. **Failure Isolation**: One service failure doesn't bring down the entire system
7. **Independent Scaling**: Scale individual services based on their specific needs



Key Components

1. API Gateway

- Single entry point for all clients
- Handles authentication, rate limiting, request routing
- Can aggregate responses from multiple services
- Examples: Kong, AWS API Gateway, Nginx

2. Service Discovery

- Services register themselves when they start
- Other services query to find available instances
- Dynamic routing as services scale up/down

- Examples: Consul, Eureka, etcd

3. Individual Microservices

- Self-contained applications
- Own their data and business logic
- Expose well-defined APIs
- Can be in different languages/frameworks

4. Message Broker (for async communication)

- Services publish events when things happen
- Other services subscribe to events they care about
- Decouples services from each other
- Examples: RabbitMQ, Apache Kafka, AWS SQS

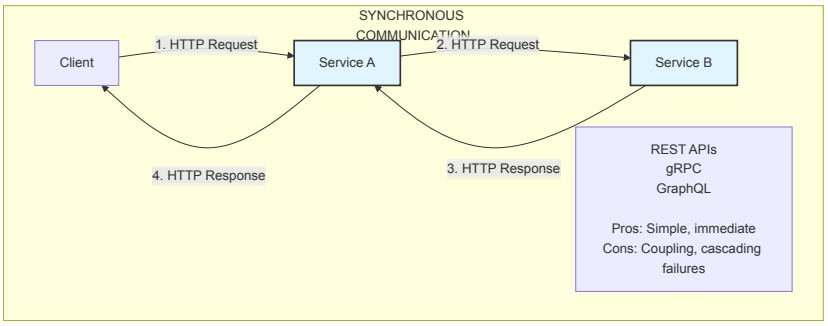
5. Distributed Tracing & Monitoring

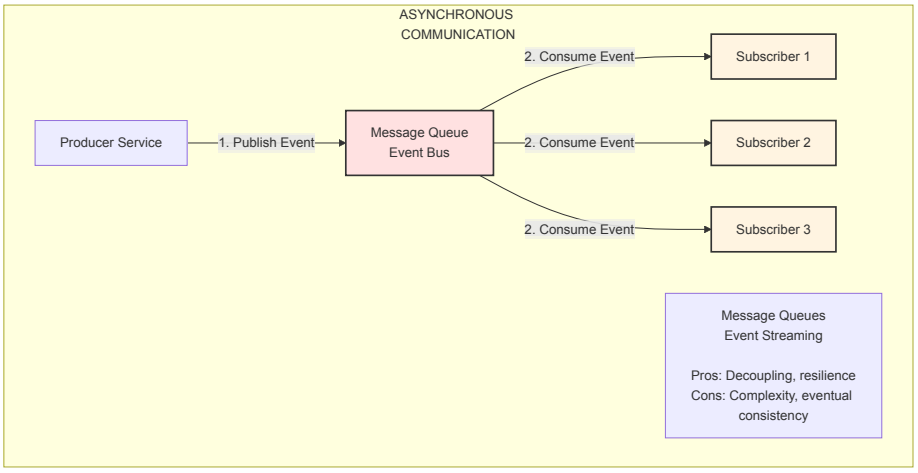
- Track requests across multiple services
- Identify performance bottlenecks
- Examples: Jaeger, Zipkin, Datadog

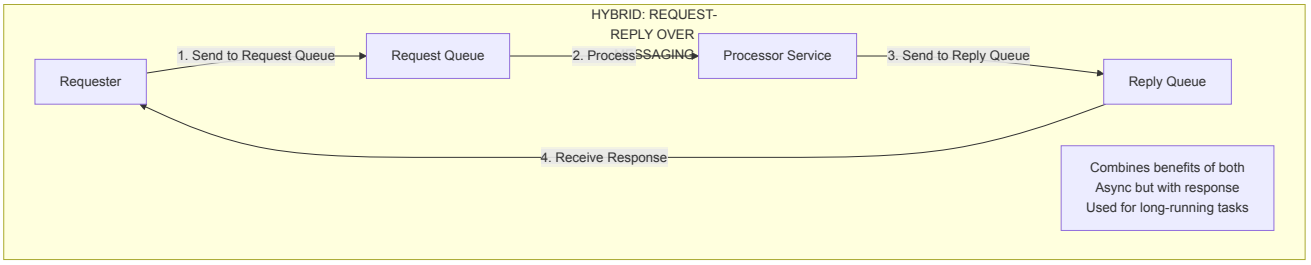
6. Configuration Management

- Centralized configuration for all services
- Environment-specific settings
- Examples: Consul, Spring Cloud Config

Communication Patterns







Synchronous Communication

Traditional **request-response pattern** where everything happens in real-time:

- **Client** makes an HTTP request to **Service A**
- **Service A** needs data from **Service B**, so it makes another HTTP request
- **Service A** waits (blocks) until **Service B** responds
- Only then can **Service A** send its response back to the **Client**

Key characteristic: Each step waits for the previous one to complete. If Service B is slow or down, everything backs up—this is the "cascading failure" problem mentioned. It's simple to implement and you get immediate results, but services are tightly coupled.

Asynchronous Communication

The **event-driven/publish-subscribe pattern**:

- **Producer Service** publishes an event/message to a **Message Queue** or Event Bus
- The producer doesn't wait—it immediately continues with other work
- Multiple **Subscribers** (1, 2, 3) independently consume the message when they're ready
- Each subscriber processes the message at their own pace

Key characteristic: Fire-and-forget. Services don't know about each other directly—they just publish/consume messages. This creates loose coupling and resilience (if one subscriber is down, others still work), but you lose immediate responses and deal with eventual consistency.

Hybrid: Request-Reply over Messaging

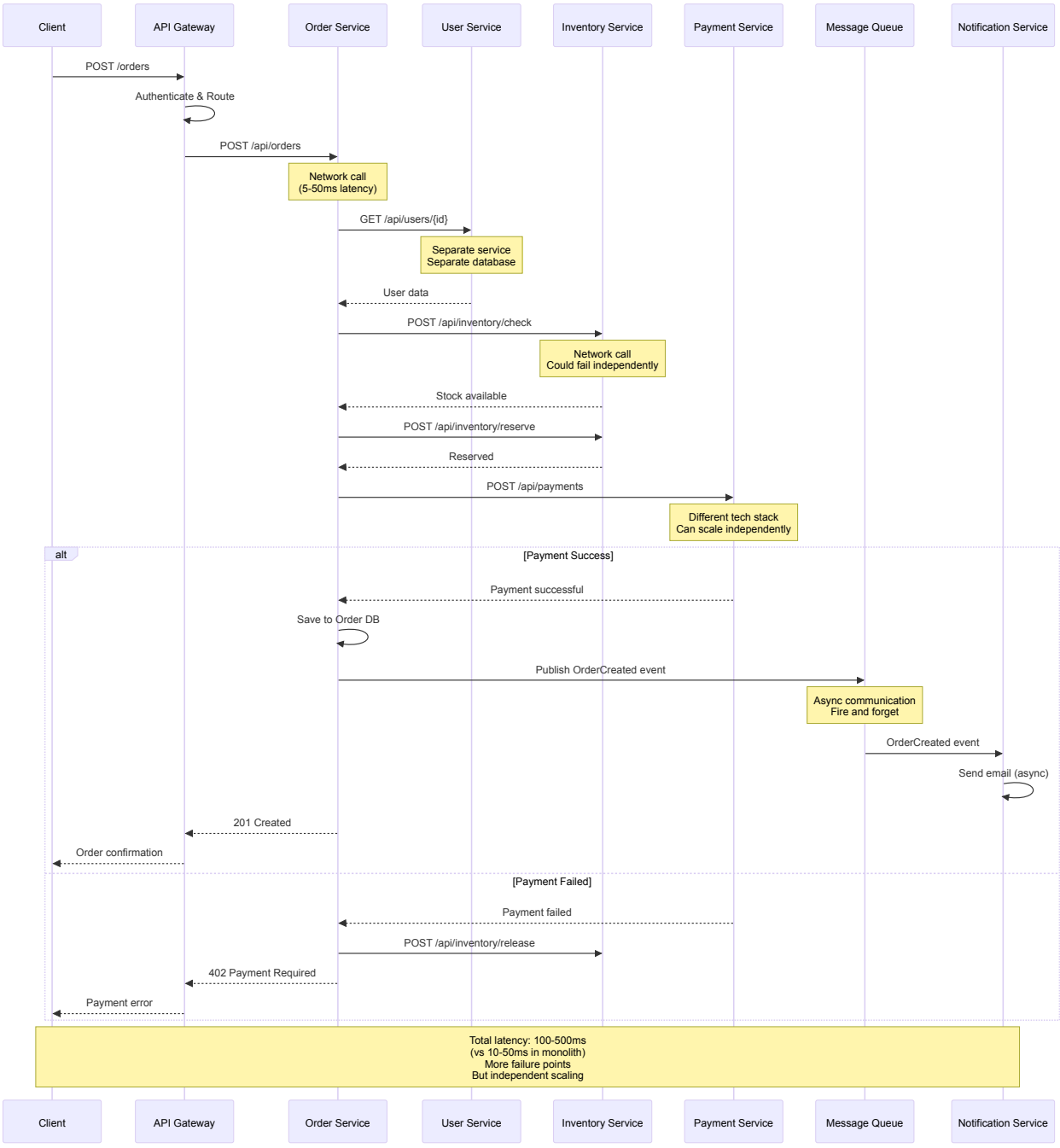
Both previous approaches for scenarios where you need a response but want messaging benefits:

- **Requester** sends a message to a **Request Queue**
- **Processor Service** picks it up when ready and does potentially long-running work
- Results go to a **Reply Queue**
- **Requester** retrieves the response from the reply queue

Key characteristic: Asynchronous execution with a response mechanism. Useful for long-running tasks (like video processing, report generation) where you don't want to block, but still need to know when it's done and get results back.

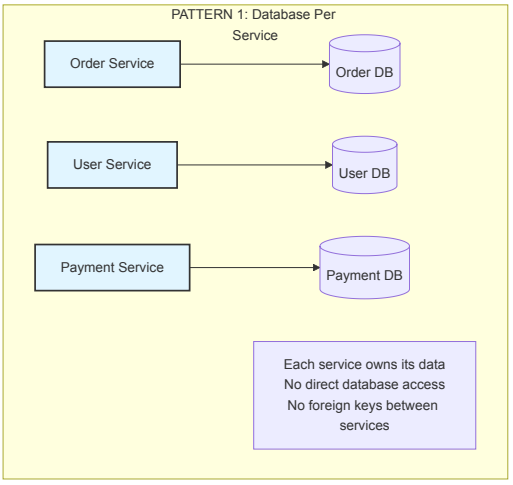
Data Flow Example

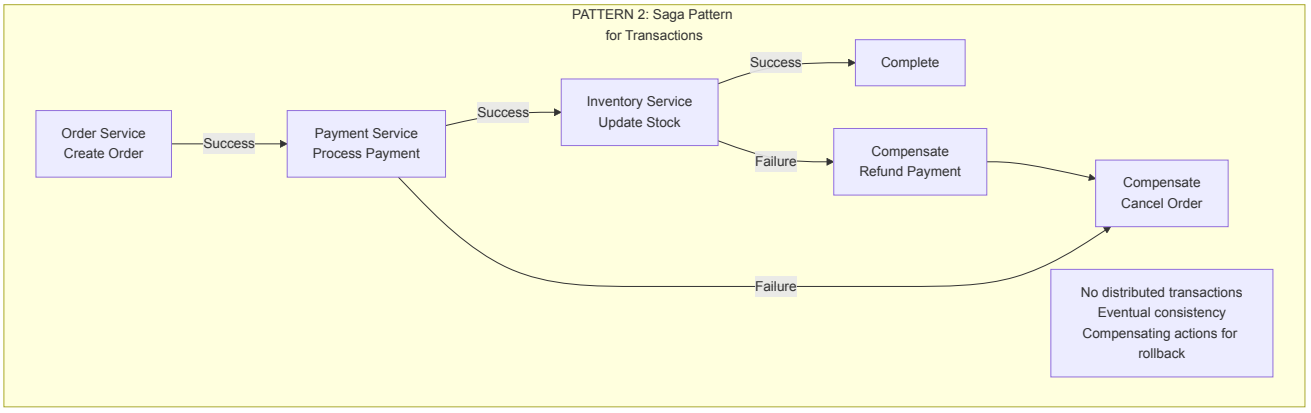
Let's see what happens when a user places an order in a microservices system:

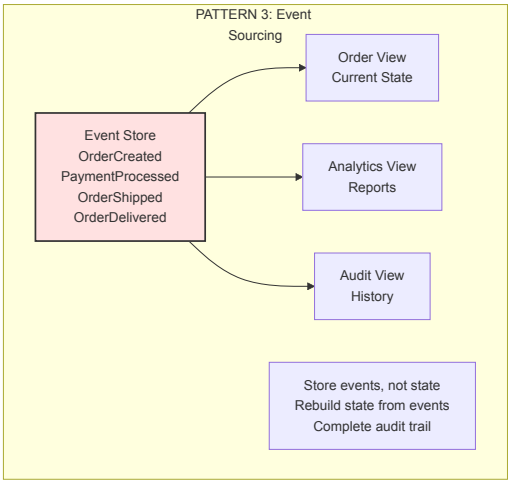


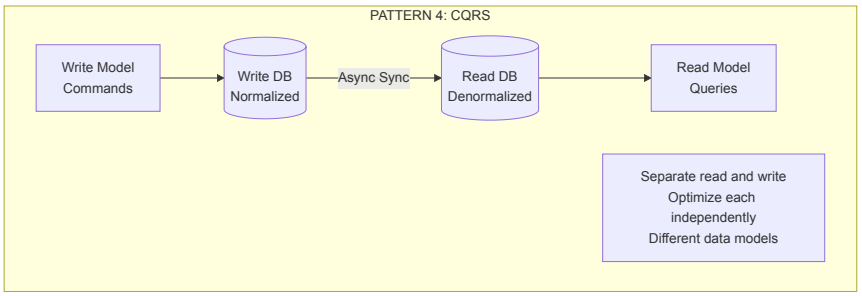
Data Management Patterns

One of the biggest challenges in microservices is data management. Each service should own its data, but business operations often need to span multiple services.









Database Per Service

The fundamental principle of microservices data architecture:

- Each microservice (Order, User, Payment) has its **own dedicated database**
- Services cannot directly query another service's database
- No shared tables or foreign keys across service boundaries

Why? This enforces **data ownership** and **loose coupling**. If the Order Service needs user information, it must call the User Service's API, not query the User DB directly. This allows each service to evolve independently—you can change the Order DB schema without breaking the User Service.

Trade-off: You lose the convenience of database joins and referential integrity across services.

Saga Pattern for Transactions

This solves the problem: "How do I handle a transaction that spans multiple services?"

The diagram shows an order flow:

- **Happy path** (top): Order → Payment → Inventory → Complete
- **Failure paths** (bottom): If payment fails, cancel the order. If inventory update fails, refund payment AND cancel order.

Key characteristic: Instead of a single database transaction, you have a **sequence of local transactions**. If any step fails, **compensating transactions** undo previous steps. It's like a choreographed dance where each service knows what to do if something goes wrong.

Why? Distributed transactions (two-phase commit) don't scale well in microservices. Sagas provide eventual consistency with explicit rollback logic.

Event Sourcing

Instead of storing the **current state** (e.g., "Order status: Delivered"), you store **every event** that happened:

- OrderCreated → PaymentProcessed → OrderShipped → OrderDelivered

The **Event Store** is the source of truth. Different views (Order View, Analytics, Audit) read these events and build their own representations:

- **Order View**: Replays events to show current order status
- **Analytics View**: Aggregates events for reports
- **Audit View**: Shows complete history

Why? You get a complete audit trail, can rebuild any state at any point in time, and can create new views from historical data. Think of it like a bank account—you don't just store the balance, you store every transaction.

Trade-off: More complex to implement and requires replaying events to get current state.

CQRS

Command Query Responsibility Segregation. This pattern

separates reads from writes:

- **Write Model**: Handles commands (create, update, delete) using a normalized database optimized for consistency
- **Read Model**: Handles queries using a denormalized database optimized for fast reads

Data flows from Write DB → Read DB asynchronously

Why? Reads and writes have different requirements. Writes need consistency and validation. Reads need speed and often query data differently than it's stored. For example:

- Write DB: Normalized tables with proper relationships
- Read DB: Denormalized view with everything pre-joined for a dashboard

Real-world example: An e-commerce site might write orders to a transactional DB, then sync to Elasticsearch for fast product searches.

How they work together:

These patterns often combine. You might use Database Per Service + Saga for distributed transactions + Event Sourcing to track everything + CQRS to optimize reads. They're building blocks that solve different pieces of the microservices data puzzle.

Advantages

Independent Deployability

- Deploy one service without touching others
- Reduce deployment risk (only one service at risk)
- Faster deployment cycles (no need to test entire system)
- Different teams can deploy on their own schedule
- Rollback individual services without affecting others
- Enable continuous deployment for individual teams

Independent Scalability

- Scale CPU-intensive services independently from memory-intensive ones
- Scale only what needs scaling (e.g., payment processing during sales)
- Optimize resource usage (don't over-provision everything)
- Different scaling strategies per service (horizontal, vertical, auto-scaling)
- Cost optimization (pay only for what each service needs)

Technology Flexibility

- Choose the right tool for each job
- Use Python for ML, Go for performance, Node.js for real-time
- Upgrade frameworks without affecting other services
- Experiment with new technologies in isolated services
- Different databases for different needs (SQL, NoSQL, graph, time-series)
- Easier to adopt new technologies incrementally

Team Autonomy

- Teams own services end-to-end
- No coordination needed for most changes
- Faster decision-making within teams
- Clear ownership and responsibility
- Different teams can work at different paces
- Reduced communication overhead

Fault Isolation

- One service failure doesn't crash entire system
- Graceful degradation (system continues with reduced functionality)
- Easier to identify which service is causing problems
- Contain issues (memory leak in one service doesn't affect others)

- Implement bulkheads to prevent cascading failures

Better for Large Organizations

- Support 50+ developers working simultaneously
- Multiple teams with clear boundaries
- Reduce merge conflicts (separate repositories)
- Enable parallel development
- Clear service contracts reduce miscommunication

Easier to Understand Individually

- Each service is smaller and focused
- New developers can understand one service
- Easier to reason about isolated functionality
- Reduced cognitive load per service

Incremental Rewrites

- Replace one service at a time
- No big bang rewrites
- Gradually migrate legacy systems
- Lower risk refactoring

Optimized for Cloud Native

- Natural fit for containers (Docker)
- Easy to orchestrate with Kubernetes
- Works well with serverless
- Leverage cloud auto-scaling
- Multi-region deployment easier

Disadvantages

Operational Complexity

- Must deploy, monitor, and manage dozens of services
- Need service discovery, API gateways, load balancers
- Distributed tracing required to debug issues
- Log aggregation from multiple sources
- Health checking and alerting for each service
- Requires DevOps expertise and tooling
- 3-5x more infrastructure to manage

Network Latency and Reliability

- Every service call is a network call (5-50ms vs < 1ms in-memory)
- Network can fail (must handle timeouts, retries)
- Serialization/deserialization overhead
- Requests may traverse 5-10 services (compounding latency)
- Need for circuit breakers and retry logic
- More failure points in the system

Distributed Transactions Are Hard

- No ACID transactions across services
- Must implement Saga pattern or event sourcing
- Eventual consistency instead of immediate consistency
- Compensating transactions for rollbacks
- Data can be temporarily inconsistent
- Complex error handling and recovery

Data Management Challenges

- Each service owns its data (good for autonomy, bad for queries)
- Joining data across services requires API calls
- Reporting and analytics become complex
- Data duplication across services
- Keeping data synchronized is difficult
- No referential integrity across services

Testing Complexity

- End-to-end tests require multiple services running
- Integration testing is challenging

- Test environments must replicate production
- Mocking service dependencies
- Contract testing between services
- Slower test execution
- More things that can fail during tests

Debugging Nightmares

- Request flows through multiple services
- Stack traces span multiple codebases
- Hard to reproduce bugs locally
- Must correlate logs from multiple services
- Finding root cause requires distributed tracing
- Time zone differences in logs

Increased Development Time (initially)

- More boilerplate code (API clients, error handling)
- Must implement cross-cutting concerns multiple times
- Service-to-service communication code
- Authentication/authorization in each service
- Configuration management across services
- More code to write and maintain

Higher Infrastructure Costs

- More servers/containers to run
- Need for API gateways, service mesh, monitoring tools
- Database instance per service
- Message queues and caches
- Development and staging environments multiply
- Network traffic costs

Versioning and Compatibility

- Must maintain backward compatibility
- API versioning strategy needed
- Breaking changes affect multiple teams
- Coordinating deployments for breaking changes
- Supporting multiple API versions simultaneously

Requires Mature Development Practices

- Need CI/CD pipelines for each service

- Automated testing essential
- Infrastructure as code
- Monitoring and observability mature practices
- On-call rotation and incident management
- Distributed systems knowledge

Organizational Challenges

- Team dependencies don't disappear
- Service ownership can lead to silos
- Cross-cutting changes require coordination
- Need clear API contracts and governance
- Documentation becomes critical
- Requires cultural change in organization

Security Complexity

- More attack surface (every service is an endpoint)
- Authentication/authorization between services
- Network security policies
- Secret management across services
- API security and rate limiting
- Certificate management

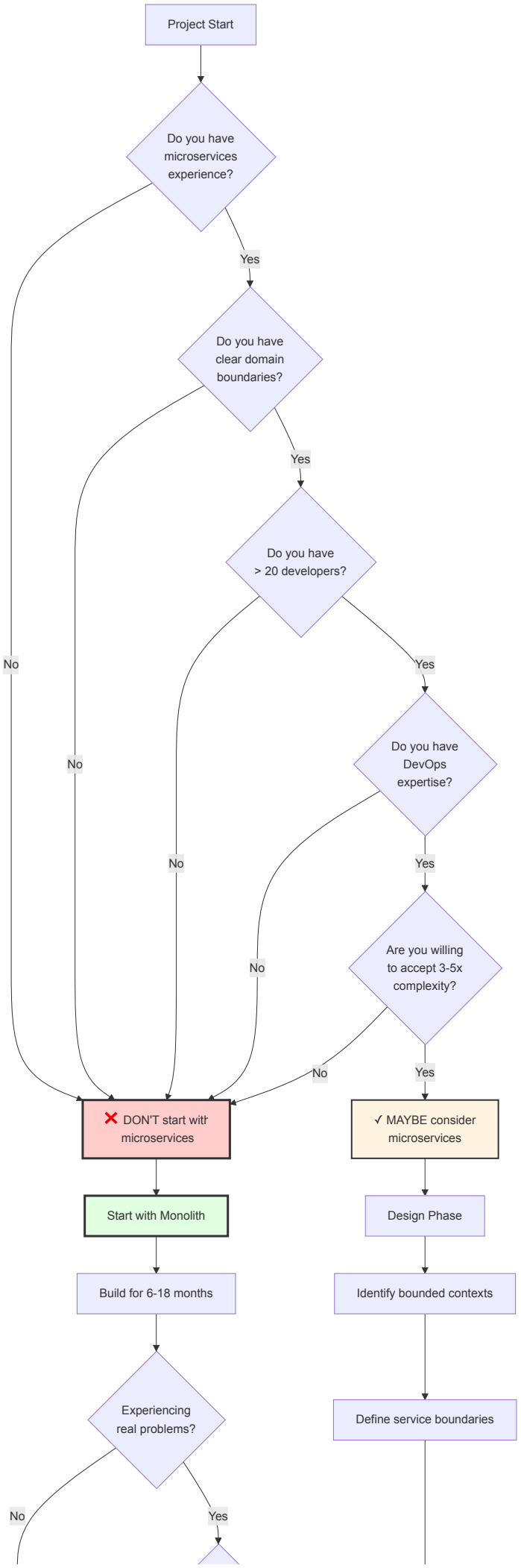
When to use Microservices

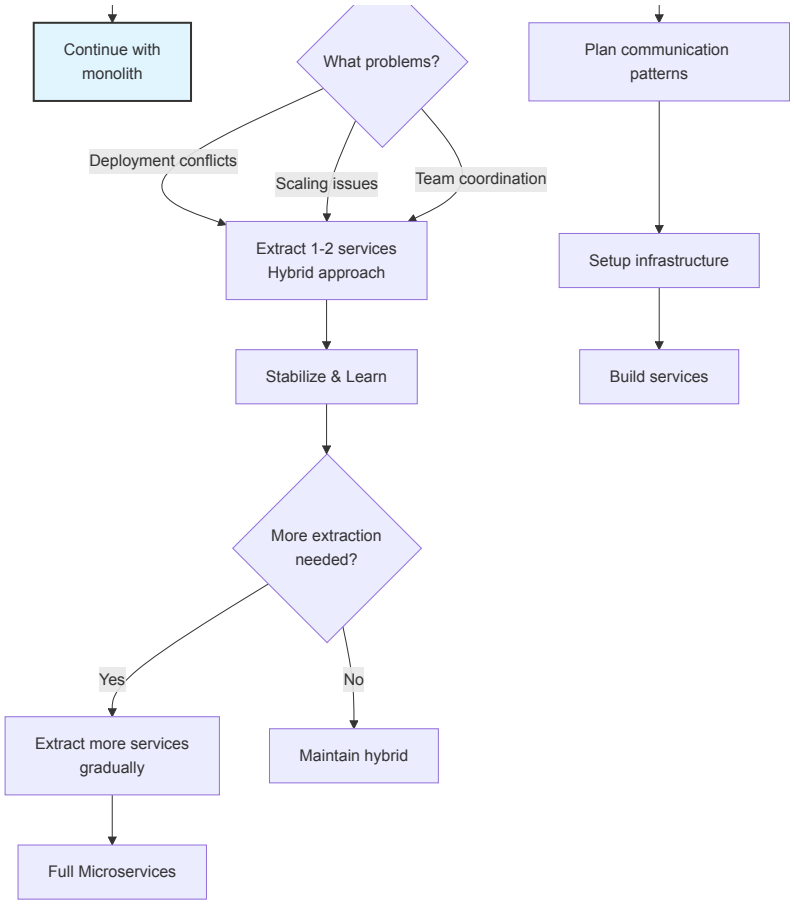
(Almost Never)

1. Planification

Microservices should be the **result of evolution**, not the starting point. The ideal time to adopt microservices is when you have:

1. **Evidence** that your monolith is causing specific, measurable problems
2. **Expertise** in distributed systems
3. **Resources** (people, money, time) to handle the complexity
4. **Clear domain boundaries** that are stable and well-understood


Project Start
Do you have microservices experience?
Yes
No
Do you have clear domain boundaries?
Yes
No
Do you have > 20 developers?
Yes
No
Do you have DevOps expertise?
Yes
No
Are you willing to accept 3-5x complexity?
Yes
No
❌ DON'T start with microservices
✓ MAYBE consider microservices
Start with Monolith
Design Phase
Build for 6-18 months
Identify bounded contexts
Experiencing real problems?
Define service boundaries
No
Yes



When it might make sense (rare cases)

- Greenfield project at a large company (100+ developers planned)
- Team already has extensive microservices experience
- Clear, stable domain model from previous version
- Budget for significant infrastructure investment
- Regulatory requirements for physical separation
- Proven product with known boundaries

When it definitely doesn't make sense

- Startup or new product (domain unknown)
- Small team (< 15 developers)
- Limited DevOps experience
- Tight deadline or MVP
- Unclear requirements
- **99% of projects fall into this category**

2. Design Phase

Critical design activities:

Domain Decomposition

```
Step 1: Identify Bounded Contexts
- What are the core business capabilities?
- Where are the natural boundaries?
- What data belongs together?
```

Service Boundary Definition

```
For each service, define:
- What data it owns (database tables)
- What operations it performs (API endpoints)
- What events it publishes
- What events it subscribes to
- Dependencies on other services
- SLA requirements (latency, availability)
```

Communication Patterns

```
Decide for each interaction:
- Synchronous (REST, gRPC) or Asynchronous (events)
- Request-response or fire-and-forget
- Real-time or eventual consistency
- Timeout and retry strategies
```

Infrastructure Planning

```
Must decide on:
- Container orchestration (Kubernetes, ECS)
- Service mesh (Istio, Linkerd)
- API Gateway (Kong, AWS API Gateway)
- Service discovery (Consul, Eureka)
- Message broker (Kafka, RabbitMQ)
- Monitoring (Prometheus, Datadog)
- Distributed tracing (Jaeger, Zipkin)
- Log aggregation (ELK, Splunk)
- CI/CD tooling (Jenkins, GitLab, GitHub Actions)
```

Following phases

Out of the scope of this course.
