Lesson 4 -> Monolith vs Microservices

Monolith

Definition:
A monolith is a single, unified application where all functionality lives in one place.

Characteristics

✓ All features packaged together
✓ Single codebase
✓ Single deployment unit
✓ Simple to develop initially
✓ Limited scaling — you scale *everything* at once

Everyday Example (Retail context)

A retail app where:

- Product catalog
- Pricing
- Recommendations
- Cart checkout
- User accounts
  … all run from a single server

This is a **monolith**.

Microservices

Definition:
Microservices break an application into **small, independent services**.

Each service:

- Does **one thing well**
- Has its **own data store**
- Can be deployed independently
- Scales independently

Example in Retail

Instead of one app, you have:

- Product Service

- Pricing Service
- Recommendation Service
- Cart Service
- Billing Service
- User Authentication Service

Each service talks to others via APIs.

Monolith vs Microservices — Core Differences

Feature	Monolith	Microservices
Codebase	Single	Multiple
Deployment	One big deployment	Independent services
Scaling	Entire app	Only needed parts
Flexibility	Low	High
Team ownership	Hard	Easy
Fault isolation	Poor	Good
Dev speed	Good initially	Better long-term

Why this matters in ML System Design

Imagine you want to build:
Real-time recommendations at scale

With a monolith:

- Your model inference lives inside the web server
- If recommender load spikes → entire app slows
- You can’t scale recommender separately

In microservices:

- Recommendation becomes a *separate inference service*
- You scale only the recommender
- You can swap models independently

This is **critical for real-time ML at scale**.

Two Main Migration Strategies

Strategy 1: Big Bang Approach ❌ (Not Recommended)

Strategies for Migrating from Monoliths to Microservices

What it involves:

- Build the entire microservices architecture upfront
- Create all microservices at once
- Redirect a small percentage of users (e.g., 10%) to the new architecture
- Test and validate before full migration

Problems with this approach:

- **Massive engineering challenge**: Requires writing all microservices simultaneously
- **Database complexity**: Must ensure all databases are correctly configured
- **User routing complexity**: Need to implement user redirection mechanisms
- **High risk**: Significant effort with potential for failure
- **Resource intensive**: Requires substantial upfront investment

Verdict: This approach is generally **not recommended** due to its complexity and risk.

Strategy 2: Incremental Migration ✅ (Recommended)

Core Principle:

Instead of building everything at once, **extract new features as separate services** as they are developed.

How it works:

1. When adding a new feature to the monolith, **separate it into a standalone microservice**
2. Convert internal function calls to network calls (HTTP, gRPC, etc.)
3. Give the new service its own database (can be same type or different)
4. Maintain separate code repositories for each service

Example: Analytics Service Migration

- **Before**: Analytics module integrated within monolith
- **After**:
- Separate analytics service
- Network-based communication (HTTP/gRPC)
- Own database (MySQL, DynamoDB, Cassandra, etc.)
- Independent code repository
- Isolated deployment

Benefits:

- ✅ Lower risk: Gradual, controlled migration
- ✅ Less upfront investment
- ✅ Easier to test and validate
- ✅ Opens path for future services
- ✅ Better separation of concerns

Example Progression:

1. Start with analytics service (new feature)
2. Later, extract email service (used by payments, profiles, analytics)
3. Continue extracting services as needed

Required Infrastructure Changes

When adopting microservices, you need to implement five critical infrastructure components:

1. Service Contracts (Clients) 📋

Problem:

- Services need to communicate via network calls
- Each service must define its contract (API specification)
- Other services need to know how to call it

Solution: Client Libraries

- Each service provides a **client library**
- Other services use this library instead of making raw network calls
- Library handles:
- Contract enforcement
- Request construction
- Network communication
- Response parsing

Example:

```
Analytics Service → Provides AnalyticsClient library

Session Service → Uses AnalyticsClient.getReport(id)
```

Important Considerations:

- ⚠️ Libraries must be kept up-to-date
- ⚠️ Breaking changes can cause failures
- ⚠️ Version management is critical

Tools/Approaches:

- Service-specific client libraries
- API versioning
- Contract testing

---

2. Request Routing 🔀

Requirement:

- Need mechanism to route requests from any service to another
- Handle service discovery
- Load balancing

Solution:

- **Load balancers** for service routing
- Route requests based on service capabilities
- Handle service discovery
- Distribute load across service instances

Components:

- Service registry
- Load balancer
- API Gateway (optional)

---

3. Simplified Deployments 🚀

Initial Approach (Not Scalable):

- Manual deployment: Copy code from GitHub to cloud server
- SSH into server and run service
- Works for few services, but doesn't scale

Better Approach:

- **CI/CD Pipelines**: Automated deployments

- Tools: Jenkins, GitLab CI, GitHub Actions
- Deploy on every merge to master (after tests pass)
- **Containerization**: Use Docker/Kubernetes
- Consistent environments
- Easier scaling
- Better DevOps experience

Benefits:

- Faster deployments
- Reduced human error
- Consistent environments
- Better scalability

---

4. Service Communication Patterns 📡

Different communication patterns for different needs:

a) Request-Response (Synchronous)

- **Use case**: Immediate response required
- **Example**: Profile service → Payment service
- **Protocol**: HTTP, gRPC
- **When**: Need immediate confirmation

b) Message Queue (Asynchronous)

- **Use case**: Non-critical, eventual consistency OK
- **Example**: Session service → Analytics service
- **Protocol**: Kafka, RabbitMQ, SQS
- **When**: Event can be processed later (e.g., analytics, logging)

c) Batch Processing

- **Use case**: Process multiple requests together
- **Example**: Bulk data processing
- **When**: Efficiency matters more than latency

Key Point: Choose the right pattern based on:

- Latency requirements
- Criticality of the operation
- Consistency needs

5. Centralized Logging 📊

Problem:

- Requests flow through multiple services
- Example flow: User → Session → Analytics → Profile → Payment
- Traditional approach: Check logs in each service separately
- **This doesn't work** - too complex and time-consuming

Solution: Centralized Logging

- Aggregate all logs from all services into a single repository
- Use correlation IDs (request ID, user ID) to track requests across services
- Enable easy debugging and root cause analysis

Technology Stack:

- **Log Aggregation**: Kafka (message queue)
- **Storage & Search**: Elastic Stack (Elasticsearch, Logstash, Kibana)
- **Alternative**: Splunk, Datadog, CloudWatch

Benefits:

- ✅ Track entire request flow
- ✅ Debug issues faster
- ✅ Root cause analysis
- ✅ Performance monitoring
- ✅ Query logs by user ID, request ID, etc.

Example:

```
User ID: 123, Request ID: 256

→ Pull all related logs from all services in one query
```

Migration Checklist

When moving to microservices, ensure you have:

□ **Service Contracts**: Well-defined APIs with client libraries

- □ **Routing**: Load balancer/service discovery mechanism
- □ **Deployment**: CI/CD pipeline with containerization
- □ **Communication**: Appropriate patterns (sync/async/batch)
- □ **Logging**: Centralized logging infrastructure

Key Principles to Remember

1. Data Encapsulation

- Each microservice must own its data
- Single source of truth per service
- Other services request data via network calls
- Caching is OK, but service remains authoritative source

2. Business Responsibility Consolidation

- Don't break services just because it's easier
- Keep related business functionality together
- Ask: "Is this used by other services?" "Are requirements coupled?"
- If tightly coupled, keep as component/library, not separate service

Example - What NOT to do:

- Profile service needs data from Google, Facebook, LinkedIn
- Creating separate "External Auth Aggregator" service
- **Problem**: Only used by profile service, requirements change together
- **Solution**: Keep as component within profile service

3. Infrastructure Cost Consideration

- Initial infrastructure cost is **high**
- Includes: logging, deployment infrastructure, documentation, monitoring
- **Not worth it for small teams** - focus on features instead
- Move to microservices when **team scales**, not when user base scales

Team Size Guidelines

Startups

- **Rule**: ≤ 2 services per person
- Flexible based on user scale

- Focus on rapid feature development

Medium Organizations

- **Rule**: ≤ 1 service per person (ideally 2 people per service)
- Avoids single point of failure
- Ensures knowledge sharing

Large Organizations

- **Rule**: 2-4 people per service
- **2 people**: Core development, no single point of failure
- **3-4 people**:
- 1 person: Proof of concept
- 2 people: Current features
- 1 person: Bug fixes and support
- **> 4 people**: Service too big, consider breaking down

Key Insight: If service needs > 4 people, business requirements are too large → break it down

---