Architecture of Large-Scale Systems

By Dr. Taghinezhad

Mail:
a0taghinezhad@gmail.com

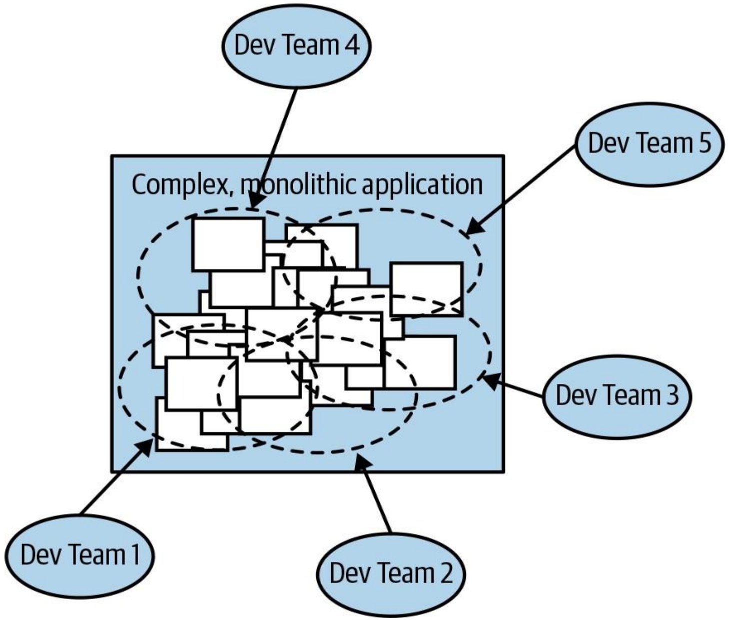
CHAPTER 3 Using Services

Introduction to Modern Software Architectures

- **Modern Software Needs**: Today’s applications demand **scalability** and **high availability**.
- **Challenge with Monoliths**: Traditional monolithic applications don’t support these demands.
  - Because, all business logic is bundled into a single unit, which limits flexibility and leads to conflicts during development (developers step on one another’s code).
- **Solution**: Move to **service-oriented architectures** or **microservices** that break down the application into smaller, independently deployable units.

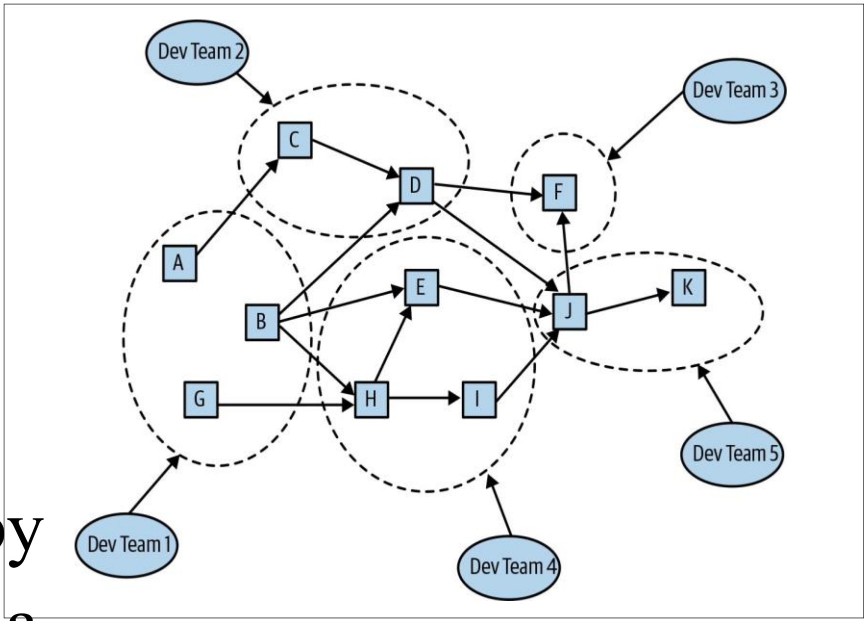
Comparison of Monolith vs. Service-Based Architectures

- **Monolithic Applications (Figure 3-1):**
  - All functionality in a single component
  - Scalability bottlenecks, especially when multiple teams work on the same codebase
    - Lack of Visibility of Code when developers work on the same module
  - Impact on Code Quality: merging code frequently with multiple developers leads to spaghetti code
    - Intertwined code, making changes complex and risky



Comparison of Monolith vs. Service-Based Architectures

- **Service-Based Applications (Figure 3-2):**
  - Functionality split into independent services
  - Clear ownership of services by distinct teams, each team has a clear non-overlapping set of responsibilities
  - Reduced conflicts, better scalability, and independent deployments



Key Benefits of Service-Based Architectures

- **1. Scaling Decisions**:
  - Each service can be scaled independently.
- **2. Team Assignment and Focus**:
  - Teams can focus on specific scaling and availability requirements of their systems in small without affecting others.
- **3. Complexity Localization**:
  - Each service is a "black box" for other teams, reducing the cognitive load. Other developers know the capabilities of the service not the details of it.
- **4. Easier Testing**:
  - Service-based systems allow for more isolated, targeted testing.
  - Service-oriented architectures can increase system complexity if service boundaries are poorly designed, potentially reducing scalability and availability.

Service Ownership and API Contracts (Figure 3-3)

- If two teams develop two services, left service teams know about their services completely, but what should they know about the right service?
  - As a start, the team needs to know the following:
    - The capabilities provided by the service
    - How to call those capabilities (the API syntax)
    - The meanings and results of calling those capabilities (the API semantics)
- What doesn’t it need to know about the Right Service?

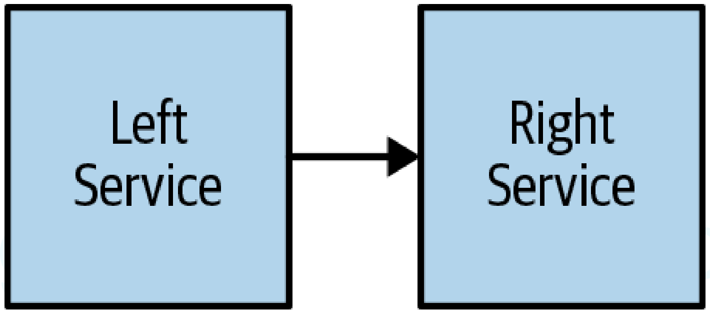


Figure.3.3. we see two services owned by two distinct teams. The Left Service is consuming the capabilities exposed by the Right Service.

Service Ownership and API Contracts (Figure 3-3)

- What doesn't it need to know about the Right Service?
  1. **Service Transparency**
     1. One service (e.g., **Left Service**) does not need to know whether the other service (e.g., **Right Service**) is composed of **a single service** or **multiple subservices**.
  2. **Dependency Abstraction**
     1. The Left Service does not need to be aware of the **dependencies** used by the Right Service to perform its tasks.
  3. **Technology Agnosticism**
     1. The implementation details, such as the **programming language** of the Right Service, are irrelevant to the Left Service.
     2. It also does not need to know the **hardware or system infrastructure** required to run the Right Service.
  4. **Service Operations**
     1. The Left Service does not need to know **who operates** the Right Service but must know how to **contact the owner** to report issues or request support.

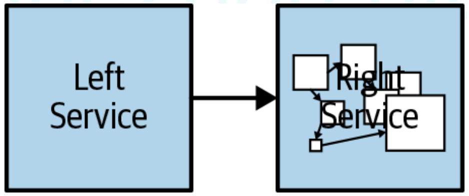


Figure 3-4. Left Service does not what's inside the Right Service,is it complex or simple

Splitting into Services

- **Definition of a Service**:
  - Provides specific capabilities (e.g., billing, account creation, notification)
  - A **standalone component** (critical for independent functionality)
  - Owns its **code base** and **data store**
  - **Exposes an API** for interaction with other services

Splitting into Services

- the **Left Service** must be able to **depend** on a **contract** that the **Right** Service provides.
  - This **contract describes** everything the **Left** Service **needs** to use the Right Service.
    - The contract contains two parts:
      - 1) The capabilities of the service (the **API**)
        - What the service does
        - How to **call** it and what each call means
      - 2) The responsiveness of the service
        - How **often** can the **API** be **used**?
        - **When can** it be used?
        - How **fast** will the API **respond**?
        - Is the **API dependable**?
- The responsiveness part of the contract is a service-level agreement (SLA)

Service Characteristics

- **Service Characteristics**:
  - Maintains its **own code base**
  - **Manages its own data** (stored separately)
  - Provides **capabilities** to other services via APIs
  - Consumes **other services' APIs**
  - Single team **ownership**

What Should Be a Service?

How do you decide when a piece of an application or system should be separated into its service?

1. **No Single Correct Answer**
    1. Determining when to split a part of an application into its own service depends on the **specific needs and goals** of the organization.
    2. Different companies adopt different approaches:
        1. **Microservices**: Hundreds or thousands of small, independent services.
        2. **Larger Services**: Only a few services, each handling a broader set of functionalities.
2. **Industry Trends Toward Microservices**
    1. The trend favors **smaller microservices** with more modular functionality.
3. **Technological Enablers**
    1. Tools like **Docker** and **Kubernetes** make managing many microservices feasible.
    2. These technologies provide **infrastructure** for deploying, scaling, and managing large numbers of small, independent services efficiently.

Guidelines for Dividing Services (Figure 3-6)

When determining **service boundaries**, factors like **company organization**, **culture**, and the type of application play key roles.

These **guidelines (not strict rules)** help to think through how to divide an application into services. These boundaries may evolve as the industry progresses.

Guidelines for Defining Service Boundaries (in order of priority):

1. **Specific Business Requirements**
    - **Specific Business Requirements: Regulatory needs** (e.g., credit card processing)
    - **Security concerns** (e.g., firewalls, validation)
    - **Access restriction** (limit access to sensitive data)
    1. Are there specific **business needs** (e.g., accounting, security, or regulatory requirements) that dictate where a service boundary must be?
        1. **Example**: In a financial app, regulatory rules might require separating **Payment** and **Accounting** services to ensure compliance and audits.

Guidelines for Defining Service Boundaries (in order of priority):

2. Distinct and Separable Team Ownership

- Each service is owned by a **single team** (3-8 developers)
- Teams handle development, testing, deployment, and performance
- **Loosens dependencies** between teams

1. Is the team responsible for the service’s functionality **independent** (e.g., located in another city, on a different floor, or managed separately)?
    1. **Example**: A team in a different office might manage a **Customer Support Service**, making it logical to create a boundary around that service.

Naturally Separable Data (Figure 3-7)

3. **Naturally Separable Data**
   1. Can the data managed by the service be **separated** naturally from other data? Will putting that data in a separate store **burden the system**?
      1. **Example**: An **Inventory Service** in an e-commerce system may have data that's independent of the **Customer Data**, making it ideal for separation.

- **Data Ownership**:
  - Services must manage their own data
  - Data should be accessed **only via APIs** (no direct access)
  - **Correct way**: Access through service APIs (Figure 3-6)
  - **Incorrect way**: Direct data access between services (Figure 3-7)

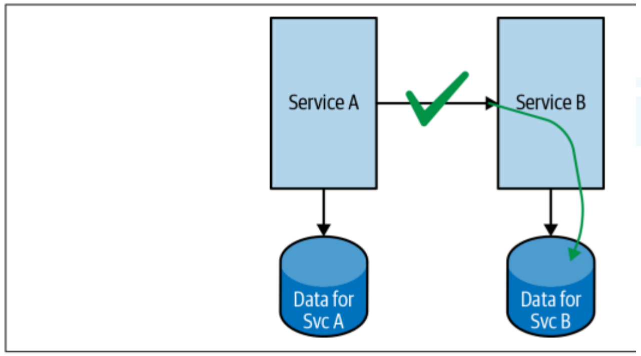


Figure 3-6. Correct way to share data

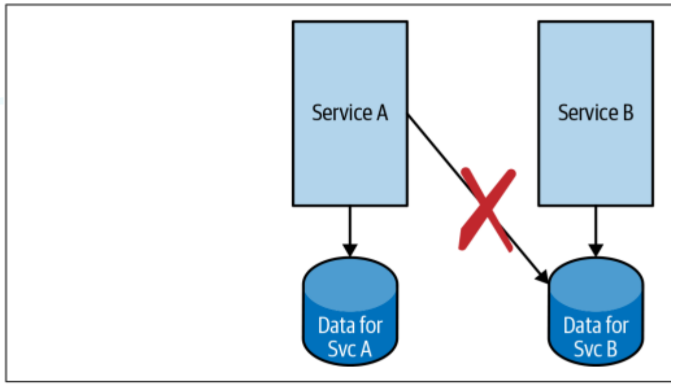


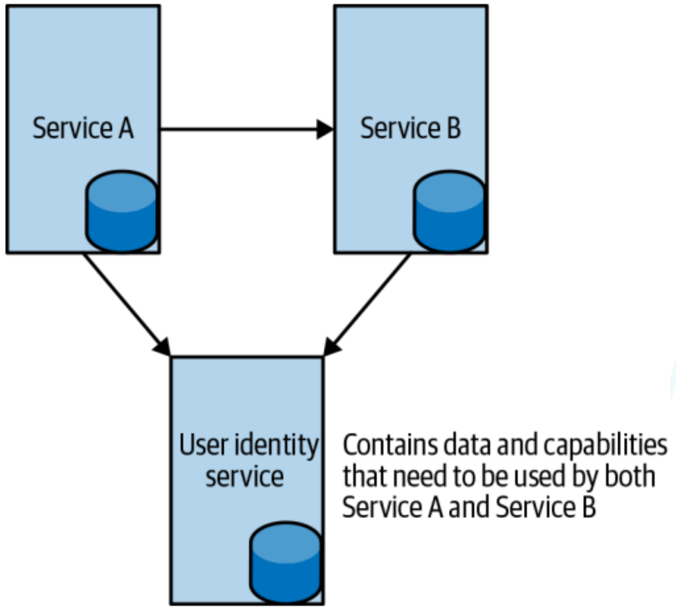
Figure 3-7. Incorrect way to share data

Guidelines for Defining Service Boundaries (in order of priority):

4. Shared Capabilities and Data

- Services like **user identity** offer shared capabilities
- Managed by a centralized service, accessed by others

1. Does the service provide **shared capabilities** used by multiple other services, and does it need **shared data** to operate?
    1. **Example**: An **Authentication Service** might be shared across many parts of an application (like login, payment, and account management).


Service A
Service B
User identity service
Contains data and capabilities that need to be used by both Service A and Service B

Service-Based Architecture Considerations

- **Balance between simplicity and complexity**:
  - Service size affects **operational management** and **scalability**
  - Consider business logic, data management, and team ownership
  - **Optimize** based on application needs and company culture

Key Challenges of Microservices

1. **Increased Overall Complexity:**
   - As you introduce more services, managing the interactions and dependencies between them becomes more challenging. This can lead to difficulties in understanding the entire application architecture.
2. **More Failure Opportunities:**
   - Each service is an independent component. If one service fails, it can affect multiple other services that depend on it, leading to cascading failures.
3. **Harder to Change Services:**
   - With more consumers for each service, changes to a service can have unintended consequences on other services, making it harder to evolve the system.

4. **Increased Dependencies:**
   - More services often mean more inter-service dependencies, which can complicate deployment and testing processes.

Finding the Right Balance

- **Balance of Services**:
  - Too few services = Monolith-like issues
  - Too many services = Increased complexity
  - Trade-offs between **service size** and **application complexity**

Finding the Right Balance:
Example: Online Food Delivery Application

Scenario: Imagine you're developing an online food delivery application. You need to consider how to break down the application into services.

Too Few Services (Monolith-like Issues):

- Service Structure: You decide to create just three services:
  1. **User Service** (handles user profiles, authentication)
  2. **Order Service** (**manages orders** and payment processing)
  3. **Restaurant Service** (manages **restaurant listings** and menus)

Issues:

1. • **Tightly Coupled**: **All three services are tightly coupled.** For instance, **if you need to update the payment processing logic** in the Order Service, you may inadvertently affect the **User Service's authentication flow.**
2. • **Deployment Challenges**: **Deploying updates requires** taking down the **entire application**, leading to downtime.
3. • **Scaling** Difficulties: **If the Order Service experiences high traffic during peak hours, you can't scale it independently without scaling the entire application.**

Finding the Right Balance:
Example: Online Food Delivery Application (Cont.)

Issues: • **Scaling** Difficulties: inappropriate (Resource Allocation When Scaled 3x)

1. **Example: Friday Night Traffic Surge: Deep Dive Analysis**

- **1. Traffic Pattern Analysis**
- **Hourly Order Volume (Friday): -------→**

2 PM	100 orders/hour (Baseline)
4 PM	150 orders/hour (+50%)
6 PM	500 orders/hour (+400%)
8 PM	800 orders/hour (+700%)
10 PM	300 orders/hour (+200%)

- 6 PM - Normal Traffic:
  - User Service: 20% CPU usage
  - Order Service CPU: 90% utilization,
  - Restaurant Service CPU: 30% utilization

- Current Solution: Scale everything 3x - Cost: 3x infrastructure for all services - Result: Massive waste as 2 services run at 60-70% idle

Finding the Right Balance:
Example: Online Food Delivery Application (Cont.)

- **Finding the Right Balance in Scaling**
- **Case Study: Online Food Delivery Application**
- **Problem:** Inefficient scaling strategy during peak traffic.
- **Scenario:** *Friday night surge — detailed traffic analysis*

Time	Orders/hour	Increase vs Baseline
2 PM	100	—
4 PM	150	+50%
6 PM	500	+400%
8 PM	800	+700%
10 PM	300	+200%

Service Load at 6 PM (Normal Operation):
•**User Service:** 20 % CPU usage
•**Order Service:** 90 % CPU utilization (bottleneck)
•**Restaurant Service:** 30 % CPU usage

Finding the Right Balance: Example: Online Food Delivery Application (Cont.)

Current Practice: blanket scaling (scale-all $\times 3$)

•**Action:** All microservices are scaled uniformly.

•**Cost:** Infrastructure expenses triple.

•**Outcome:**

- Order Processing Service benefits
- User & Restaurant Services end up **60–70% idle**, causing substantial waste

Why teams use blanket scaling

- **Simplicity and speed**
- **Limited observability / metrics gaps**
- **Architectural constraints**
- **Autoscaler/Tooling limitations or misconfiguration**

Finding the Right Balance:
Example: Online Food Delivery Application

Too Many Services (Increased Complexity):

• Service Structure: You decide to break down the application into several smaller services:

1. User Service

2. Order Service

3. Restaurant Service

4. Payment Service

5. Notification Service

6. Review Service

7. Delivery Tracking Service

• Issues:

• Increased **Complexity:** With so many services, understanding how they interact becomes **difficult**.

• Deployment **Overhead**: Each service needs to be deployed independently,

• Monitoring and **Debugging**: **Identifying issues across multiple services** can be complex and time-consuming

Finding the Right Balance: Micro vs Macro Services

- **Define clear service boundaries:** Group functionality by business capability; each service should own a cohesive domain.
- **Apply Domain-Driven Design:** Use bounded contexts to make services meaningful, decoupled, and aligned with domain logic.
- **Align with organization:** Map services to team ownership so teams can iterate independently and take responsibility.
- **Right-size services:** Avoid trivial single-endpoint services and monoliths ; choose a scope that keeps complexity low and value high.
- **Enforce strong interfaces:** Stable, well-documented APIs (contracts, versioning) reduce coupling and make evolution safe.
  - *Contract:* Order Service exposes POST /v1/orders and GET /v1/orders/{id} with a documented JSON schema; changes require version bump.

- **Rule of thumb:** prefer smaller services when you need independent deploys and clear ownership; prefer larger (macro) services when cross-service coordination or latency dominates.

End Chapter