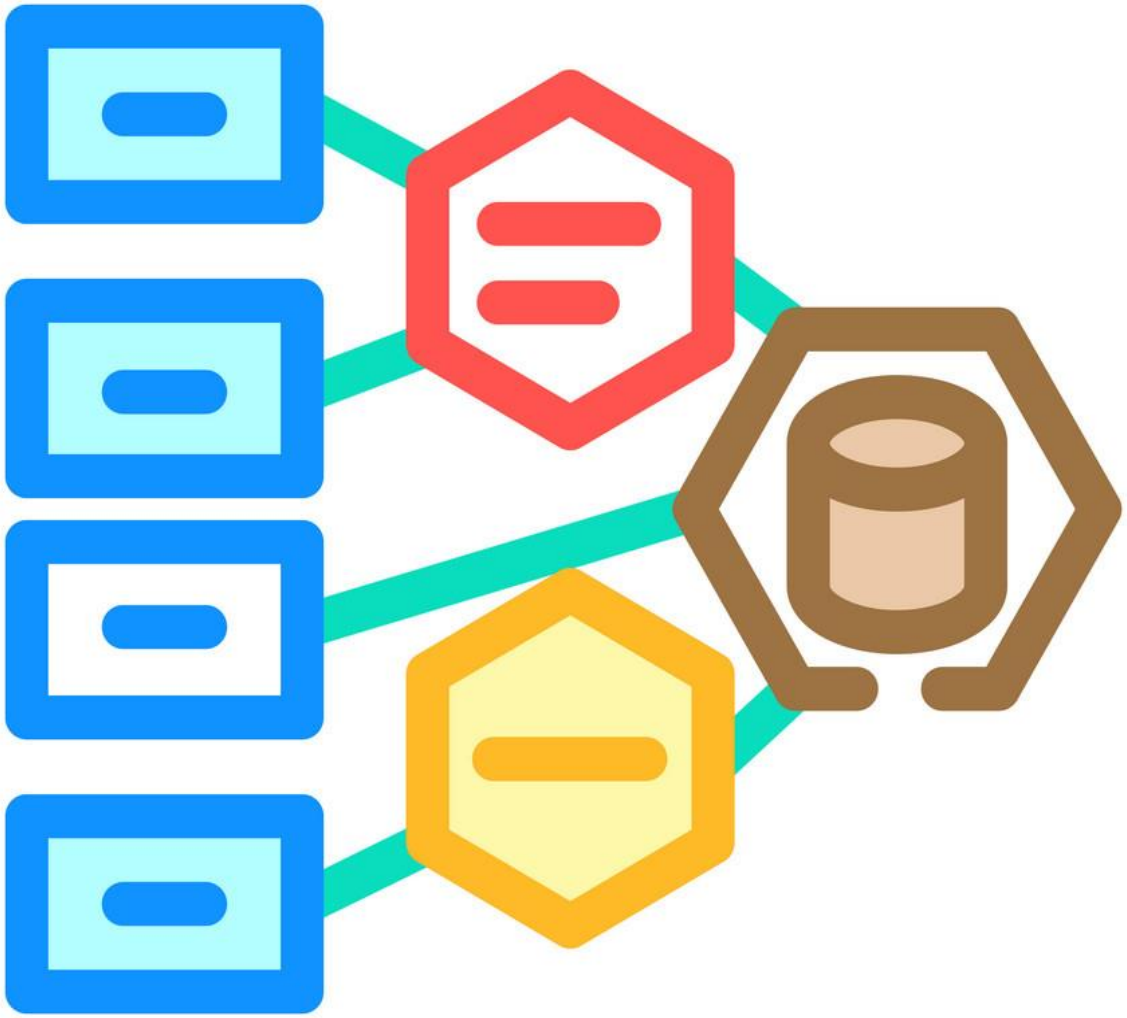
Software Design and architecture

Fall 2024

Dr. Natalia Chaudhry



Outline

- Architectural styles
  - ✓ Data flow architectures
  - ✓ Layered
  - ✓ Event-based
  - ✓ Data-centered
  - ✓ MVC
  - ✓ Multi-tier distributed
  - ✓ Service Oriented

- **Multi-Tier Distributed Architecture** is an architectural style that divides an application into multiple tiers, typically including presentation, application (or business logic), and data layers.
- Each of these tiers can run on separate servers or systems, interacting over a network.
- The architecture is especially common in enterprise applications where scalability, availability, and maintainability are priorities.

Pros of Multi-Tier Distributed Architecture

1.Scalability: Each tier can be scaled independently. For instance, if there's a high demand on the presentation layer, it can be scaled up or load-balanced separately from the application or data tiers.

2.Enhanced Security: Each tier can have its own security protocols, making it easier to isolate data access and enforce security policies. Sensitive data stays within the data tier, minimizing security risks.

3.Maintainability and Modularity: Each tier is modular, with a clear separation of concerns. This makes it easier to update or replace parts of the system without affecting other layers.

4. Availability and Redundancy: Distributed deployment across multiple servers allows for redundancy. If one server fails, other instances of the same tier can take over, reducing downtime.

5. Improved Performance: Load balancing across servers can improve response times, as processing tasks can be divided among multiple resources.

Cons of Multi-Tier Distributed Architecture

1. **Increased Complexity**: Managing a distributed system with multiple tiers adds complexity in terms of configuration, deployment, and maintenance.
2. **Higher Costs**: Multiple servers and the networking infrastructure to support them increase the cost of hardware, software, and personnel for managing the infrastructure.
3. **Latency Issues**: Network latency between tiers can slow down response times, especially if there are multiple back-and-forth communications between the presentation and data tiers.

4. Data Consistency Challenges: Since data is accessed across distributed servers, ensuring data consistency and managing concurrent data access can become challenging.

5. Testing and Debugging Complexity: Testing and debugging across distributed servers with multiple layers is more complex compared to single-server or monolithic architectures.

High-Volume E-commerce Platforms

- E-commerce platforms (like Amazon or Alibaba) require the ability to handle thousands of concurrent users, high transaction volumes, and diverse features, such as product catalog management, user sessions, payment processing, and real-time inventory.
- **Why Multi-Tier Works Best**: Multi-tier allows each tier (UI, business logic, and data) to scale independently. For example, during a sales event, only the UI layer might need extra scaling for handling traffic surges, while business logic and data layers may remain as they are. Other architectures, like monolithic, would struggle with scaling parts independently, making it difficult to manage spikes and resource allocation.

Enterprise Resource Planning (ERP) Systems

- ERP systems (like SAP or Oracle ERP) need to manage diverse functions, from HR and accounting to inventory and customer relationships, often across multiple departments and locations. Each module requires distinct processing and storage needs.
- **Why Multi-Tier Works Best**: Multi-tier allows separate modules to be maintained and scaled as needed without impacting the entire system. Each tier can be optimized according to its module's needs, enabling easier updates, versioning, and scaling per department or function. A monolithic architecture would be less flexible, as changes to one function would require system-wide updates, making maintenance challenging in ERP scenarios.

Telecommunications Billing Systems

- Telecom billing systems process vast amounts of data from multiple channels, requiring complex business rules for calculating charges, managing customer plans, and producing detailed billing statements.
- **Why Multi-Tier Works Best**: Multi-tier architecture can distribute data collection, billing logic, and customer presentation across different tiers, each optimized for specific tasks. This setup allows for easy scaling as data volumes grow and provides clear isolation for billing logic, improving maintainability and preventing disruptions. Microservices would add unnecessary complexity due to the need for orchestration, whereas multi-tier is simpler to manage and deploy in such scenarios.

Government Information Systems

- Government systems often involve multiple applications and services, from tax processing to citizen services, with varying access levels, privacy requirements, and workloads.
- **Why Multi-Tier Works Best**: Multi-tier architecture provides a clear division between public-facing services, internal processing, and data storage. It enables independent scaling of public-facing layers during high-demand periods, like tax filing season, while keeping backend layers secure and unaffected. Alternative architectures like microservices would require greater orchestration and overhead, which can be more challenging to manage for a centralized government system.

Supply Chain Management (SCM) Systems

- SCM systems coordinate activities across various organizations, including suppliers, manufacturers, distributors, and retailers. They handle a large amount of data and must facilitate real-time inventory tracking, logistics coordination, and demand forecasting.
- **Why Multi-Tier Works Best**: Multi-tier architecture supports scalability and modularity for different functions across the supply chain (e.g., order management, inventory tracking, and shipping coordination) while maintaining synchronized data access. Other architectures would struggle to manage complex cross-organization coordination and real-time updates efficiently.

Layered vs multi tier Architecture

Scope of Separation

- **Layered Architecture**: Layers are logical divisions within an application (such as presentation, business logic, and data access). Each layer has a dedicated responsibility, but all layers may exist within a single application or server, operating sequentially. Layers don't necessarily run on separate physical servers; instead, they are organizational within the application code.
- **Multi-Tier Architecture**: Tiers refer to separate physical locations where different parts of the application reside. Multi-tier implies that each tier (e.g., presentation, application, data) can be deployed on different servers or systems, allowing each tier to scale independently.

Deployment Strategy

- **Layered Architecture**: Primarily a **logical** structure within an application; it doesn't require multiple servers. A layered application could run entirely on one machine, with each layer stacked on top of another logically.
- **Multi-Tier Architecture**: By design, this is a **distributed** setup. The presentation tier, application tier, and data tier are often deployed on different servers, making it possible to balance load independently across tiers.

Purpose and Flexibility

- **Layered Architecture**: Focuses on **separation of concerns** within the application codebase, ensuring modularity and maintainability. It is useful in applications where functionality needs to be well-organized but doesn't necessarily need to be distributed across multiple systems.
- **Multi-Tier Architecture**: Provides **scalability and fault tolerance** by physically separating components. This allows each tier to be scaled independently, which is beneficial for high-traffic applications where different tiers (e.g., user interface vs. data storage) may have distinct performance needs.

Examples

- **Layered Architecture Example**: In a desktop application, a three-layer setup might consist of a UI layer, a business logic layer, and a data access layer. All layers exist on the same machine but maintain separate responsibilities.
- **Multi-Tier Architecture Example**: In a web application, the presentation layer might run on a web server, the business logic layer on an application server, and the database on a separate database server. Each server (tier) can scale independently.

Monolithic architecture

- This structure integrates the user interface, business logic, and data access layer within a single codebase.

Pros of Monolithic Architecture

Simplicity:
Monolithic systems are often easier to build initially since all components are in one place, making the system straightforward to understand and develop.

Easy to Test:
Testing can be simpler because there's one codebase. Unit tests, integration tests, and end-to-end tests can be run against a single deployable artifact.

Unified Deployment:
All components are deployed together, reducing complexity in deployment and configuration compared to microservices. A single package can be deployed, which is simpler to manage.

Performance:
Communication within the application is typically faster in a monolithic design, as inter-process calls happen within a single runtime environment rather than over a network.
Easier Debugging and Logging:
Monitoring and logging across a single codebase are easier to set up and maintain, making debugging faster and more straightforward.

Cons of Monolithic Architecture

Scalability Limitations:
Scaling is limited because the whole application is deployed as a single unit. Scaling often involves replicating the entire application instead of specific bottleneck components.

Tight Coupling:
Components are tightly coupled, meaning changes in one part of the system can impact the entire application, leading to high-risk deployments.

Limited Flexibility in Technology Choices:
It’s challenging to adopt new technologies for a single component without refactoring the entire application.

Deployment and Downtime Challenges:
Updates require redeploying the whole application, which can lead to downtime or necessitate sophisticated deployment strategies to ensure availability.

Complexity with Growth:
As the application grows, it can become unwieldy, making it harder to manage, understand, and maintain over time.

Use Cases for Monolithic Architecture

Startups and Small Projects:

Small applications with limited complexity or for startups aiming to launch a product quickly can benefit from the simplicity of monolithic architecture.

Well-Defined, Stable Applications:

Applications with stable requirements, fewer updates, and clear functionality can thrive as monoliths due to low deployment needs.

Internal Tools and Enterprise Apps:
Applications used internally within companies, like inventory management systems or simple ERP tools, often work well as monoliths because of the lower scalability requirements and higher need for ease of management.
Applications with Tight Performance Requirements:
Applications needing high-performance with minimal inter-service latency, such as certain financial systems, can benefit from the single runtime environment provided by a monolithic structure

Client-Server Architecture

Use Cases:

1. **Web Applications**: Traditional websites and web-based applications use client-server architecture where the browser (client) communicates with a web server to fetch resources.
2. **Email Services**: Email protocols (like IMAP, SMTP) rely on client-server architecture where the email client communicates with a server to send and receive messages.
3. **Database Systems**: Database management systems (DBMS) use client-server architecture for querying and managing data, where the database server handles data requests from multiple client applications.
4. **File Sharing**: Services like FTP (File Transfer Protocol) enable clients to upload or download files from a centralized server.
5. **Gaming**: Online multiplayer games use client-server architecture, where clients (players) connect to a central server to synchronize game state.

Pros:

1. **Centralized Control**: All data and resources are stored and managed on the server, which can help simplify management, maintenance, and security.
2. **Improved Security**: Security measures can be implemented at the server level, providing a single point for authentication, data encryption, and access control.
3. **Easy Maintenance and Updates**: Updates and maintenance can be applied on the server, which reflects immediately for all clients, making it easier to manage without requiring client-side updates.
4. **Resource Sharing**: Clients can access shared resources on the server, which reduces redundancy and can lead to cost savings on storage or processing power.

Cons:

1. **Scalability Limitations**: Scaling can be difficult and costly as the server becomes a bottleneck with increased load, leading to performance issues.
2. **Single Point of Failure**: If the server goes down, clients lose access to resources and services, which can impact service availability.
3. **Network Dependency**: Client-server applications depend heavily on network connectivity. Network outages or latency can disrupt service and degrade user experience.
4. **Cost**: Maintaining and securing a server, along with the infrastructure needed to support multiple clients, can be costly, especially for high-traffic applications.

Microservices architecture

style of designing software systems by breaking down a large application into smaller, independent parts called *microservices*.

Each of these parts handles one specific function, and they communicate with each other to form a complete application.

Imagine a food delivery app. Instead of being one big app, it’s split into several smaller parts:

- **One microservice** might handle user registration and login.
- **Another microservice** could be in charge of restaurant listings.
- **Another one** might manage orders and payments.
- **Another** microservice could track the delivery driver’s location.

Each microservice can be developed, deployed, and updated independently. This means if the developers want to improve or fix something in the "order tracking" part, they don’t have to touch the "login" or "restaurant listings" parts

Pros of Microservices

1. **Scalability**: You can scale parts of the application independently. If the payment system is getting high traffic, you can increase its capacity without affecting other parts.
2. **Flexibility**: Different teams can work on different microservices using the best tools or programming languages for the job.
3. **Reliability**: If one microservice fails (e.g., restaurant listings), it doesn't necessarily crash the entire system.
4. **Faster Updates**: Developers can work on updates or fixes for a specific part without having to update the whole application.

Cons of Microservices

1. **Complexity**: Managing multiple services can become complicated, especially as the number grows. You need extra tools to manage the connections between these parts.
2. **Data Handling**: Different microservices may need to access shared data, which can complicate data storage and synchronization.
3. **Testing Challenges**: Since there are multiple independent parts, testing the entire application requires checking each part and how they interact, making testing more complex.
4. **Network Latency**: Communication between microservices, especially over the internet, can be slower and requires careful management.

Real-life Applications

- **Netflix**: Netflix uses microservices to manage different aspects like user recommendations, video streaming, and account management, all as separate services.
- **Amazon**: Amazon’s e-commerce platform has microservices for product listing, payment, customer reviews, and shipping tracking.
- **Uber**: Uber’s app has various microservices for location tracking, driver management, pricing, and notifications.

Outline

- Architectural styles
  - ✓ Data flow architectures
  - ✓ Layered
  - ✓ Event-based
  - ✓ Data-centered
  - ✓ MVC
  - ✓ Multi-tier distributed
  - ✓ Service Oriented

- Service-Oriented Architecture (SOA) is a design pattern that enables services to communicate and share data across different applications in a network.
- Each service in SOA is an independent, self-contained unit designed to perform a specific function, making it easier to integrate and reuse services within different applications.

Key Concepts in SOA

- **Services**: These are self-contained units that perform specific tasks, like checking a customer's credit score, processing a payment, or managing a user's account details. Each service is independent and designed to fulfill a particular function.
- **Loose Coupling**: In SOA, services are loosely coupled, meaning they don't rely on each other's inner workings. This separation makes it easier to update, replace, or change services without disrupting the overall system.
- **Interoperability**: SOA supports communication between services across different platforms, languages, and systems. For instance, one service could be built in Java, another in Python, and a third in .NET, yet they can still communicate because SOA uses standardized communication protocols.

•**Standardized Communication Protocols**: Services in SOA commonly communicate using protocols like HTTP, SOAP (Simple Object Access Protocol), or REST (Representational State Transfer). This standardization enables services to be reused and integrated across a wide range of applications.

•**Service Reusability**: Services in SOA can be reused across multiple applications or systems, making it easier to develop and maintain new software. For example, a billing service could be used by multiple departments in an organization, avoiding duplicate code and processes.

SOA vs. Microservices

SOA and microservices share some principles, like modularity and service independence, but they differ in scale and scope:

- **SOA**: Typically integrates larger, coarser-grained services that might handle multiple business processes.
- **Microservices**: Often breaks down each function into smaller, fine-grained services, typically with a single purpose, making it more granular and suitable for modern, cloud-native applications.

Pros of Service-Oriented Architecture

1. **Reusability**: Services can be reused in different applications, reducing development time and cost.
2. **Scalability**: SOA allows easy scaling of services independently, which is essential for handling varying workloads.
3. **Interoperability**: SOA promotes compatibility across various platforms and programming languages, enhancing communication across systems.
4. **Maintenance and Flexibility**: Individual services can be modified or replaced without disrupting the entire system, making updates and maintenance easier.
5. **Improved Business Agility**: New services can be rapidly created and deployed, making it easier for businesses to adapt to changes in the market or customer needs.

Cons of Service-Oriented Architecture

1. **Complexity**: SOA can be complex to implement, especially in large systems, due to the high number of services and interdependencies.
2. **Overhead and Latency**: The communication between services (usually through network calls) can introduce latency and performance overhead.
3. **Security Risks**: Exposing services over a network introduces security challenges, as each service endpoint becomes a potential entry point for malicious attacks.
4. **High Initial Investment**: Implementing SOA, especially in a legacy system, may require a significant initial investment in terms of time and resources.

Real-Life Applications of Service-Oriented Architecture

1. **E-commerce Platforms**: Amazon uses SOA to manage its vast network of microservices, enabling different parts of the website (search, product listing, reviews) to work independently and scale according to demand.
2. **Banking Systems**: Many banks implement SOA to offer flexible services for different banking operations, such as online transactions, loan processing, and account management, which can be accessed by mobile apps, ATMs, and online platforms.
3. **Healthcare Systems**: Hospitals and healthcare providers use SOA to enable seamless data sharing across different departments, such as patient records, diagnostics, and billing systems, improving patient care.
4. **Telecommunications**: Telecom companies use SOA to handle customer service functions, billing, service activation, and other services, making it easier to offer unified services to customers across different channels.

That’s it