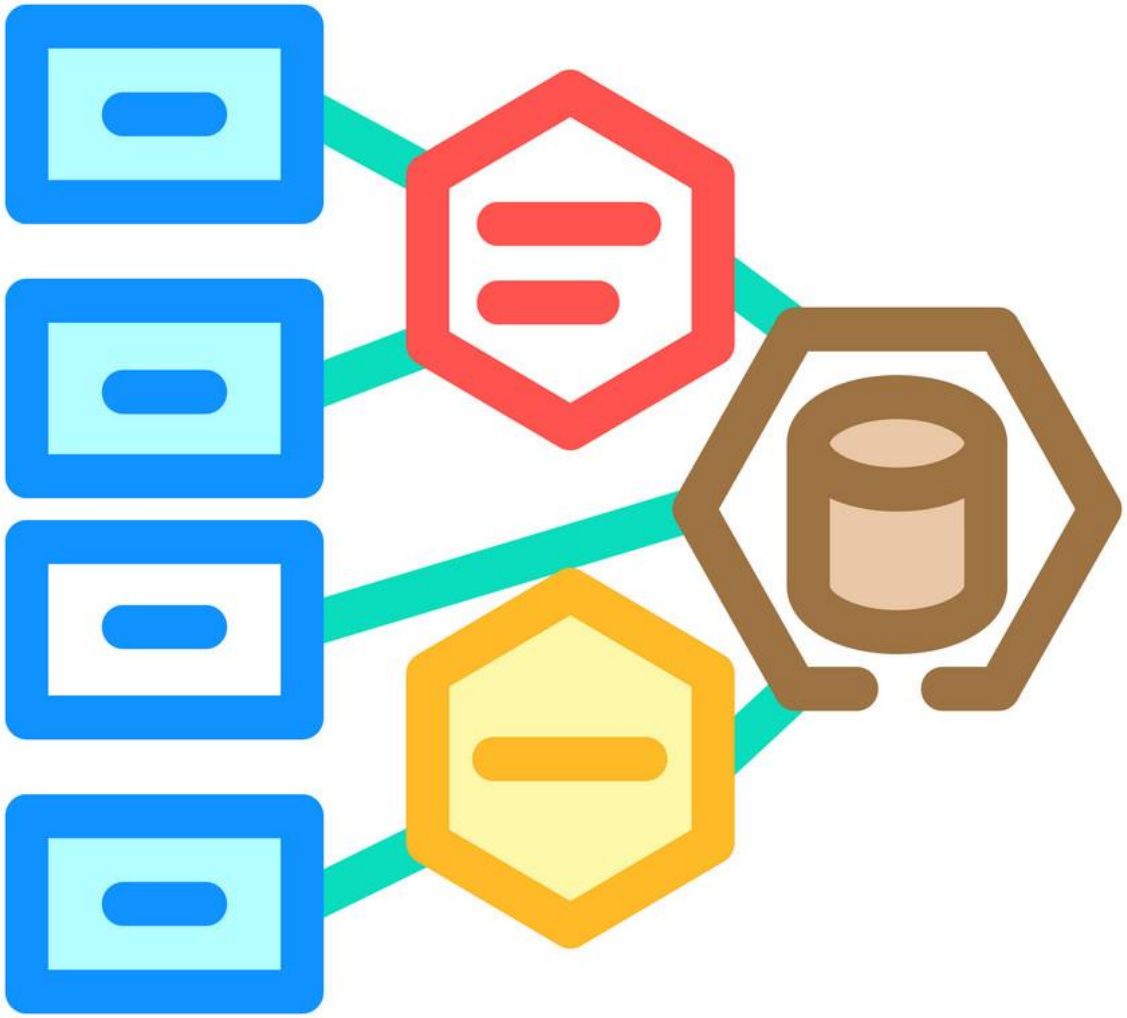
Software Design and architecture

Fall 2024

Dr. Natalia Chaudhry



Outline

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

- organizes the system into a hierarchy of layers, where each layer performs a specific role and interacts only with the adjacent layers.
- This design promotes separation of concerns, where different aspects of the system are encapsulated in different layers, enhancing maintainability and scalability.

Presentation Layer:

This layer handles the user interface and interaction, displaying data to users and receiving inputs.

User Login

The user enters their credentials (username and password) through a web or mobile interface.

- The input is validated (e.g., ensuring that the fields are not empty).
- If valid, the credentials are passed to the application layer for authentication.

Application Layer (Service Layer):

Application Layer (Service Layer): It defines the business logic and operations, coordinating tasks and responding to user requests.

User Authentication

- Receives the username and password from the presentation layer.
- It sends a request to the domain layer to check whether the credentials match.
- If the domain layer confirms the user is valid, the application layer creates a session for the user and sends a success message back to the presentation layer.
- If invalid, it returns an error message to the presentation layer.

Domain Layer (Business Logic Layer):

Domain Layer (Business Logic Layer): Contains core business rules, ensuring that operations comply with domain-specific logic.

Validate User Credentials

- Receives the request from the application layer to verify the user's credentials.
- It contains the business logic for authentication, such as password hashing or two-factor authentication.
- Calls the data layer to retrieve the stored user data (e.g., hashed password).
- Compares the hashed password from the data layer with the hashed input password.
- If they match, it informs the application layer that the credentials are valid.

Data Layer (Persistence Layer):

Responsible for managing data, including databases, files, or external storage systems.

Retrieve User Information

- Receives a request from the domain layer to get the user's details.
- Queries the database to retrieve the stored username, hashed password, and any other relevant data (e.g., roles, permissions).
- Sends the retrieved data back to the domain layer for further processing.

Application Layer: Focuses on user interactions and coordinating processes. It is responsible for ensuring that the application works as intended by controlling the flow between layers.

Domain Layer: Focuses on the business logic, rules, and domain models. It is the heart of the system where core business decisions and logic reside.

Layered architecture does not have a strict limit of only four layers (Presentation, Application, Domain, Data).

While the common four-layer model is a typical pattern, systems can have more layers with different names and purposes depending on the complexity, requirements, and design choices of the system.

Additional Layers that Can Exist:

Integration Layer (Communication Layer):

- Manages interactions with external systems, APIs, or services.
- Provides adapters or gateways to integrate third-party services or legacy systems.

Example: An adapter that connects the e-commerce system to an external shipping API for real-time tracking and updates.

Security Layer:

- Encapsulates all security-related concerns, such as authentication, authorization, and encryption.
- Ensures that sensitive data and communication are handled securely.

Example: A module that handles user authentication (login) and verifies user roles to restrict access to certain parts of the system.

Caching Layer:

- Handles temporary data storage to improve performance by reducing load on the database.
- Often sits between the data layer and domain layer or application layer.

Example: A Redis cache that stores frequently requested product details to reduce load on the database and speed up response times.

Logging & Monitoring Layer:

- Manages logging, error handling, and performance monitoring of the system.
- Useful in auditing and diagnosing issues.

Example: A centralized logging system like ELK (Elasticsearch, Logstash, Kibana) that collects error logs from all microservices and monitors performance.

Workflow/Orchestration Layer:

- Found in systems that rely heavily on processes or workflows.
- Manages the execution of business processes and workflows across different services or modules.

Example: A business process manager (BPM) tool that coordinates a loan approval process across different departments in a bank, including credit checks, risk assessment, and final approval.

Communication generally flows from the top (UI) down to the data layer. However, the dependency between layers remains unidirectional, where each layer depends only on the layer directly below it.

Use Cases:

- **Web Applications:** Separation of presentation (frontend), business logic, and data handling layers is common in web development (e.g., MVC architecture).

- **Enterprise Systems:** In large systems, where separation of services and complexity is needed, layered architecture fits well.

- **Microservices Architecture:** Although microservices architecture emphasizes the decomposition of an application into independently deployable services, **each service internally** might still adhere to a layered architectural style.

- **Banking Systems:** Where modularity is key, and separation between layers allows handling user interfaces, business rules, and data access distinctly.

Pros:

1. **Separation of Concerns:** Each layer is focused on a specific responsibility, leading to clear boundaries between the system's components.
2. **Scalability:** The system is more modular, allowing independent scaling and development of each layer.
3. **Ease of Maintenance:** Layers can be modified, replaced, or updated independently without affecting the entire system.
4. **Testability:** Layers can be tested individually, simplifying unit testing and bug detection.
5. **Reusability:** Components of layers can be reused in other applications, especially common services or business rules.

Cons:

1.Performance Overhead: Due to multiple layers, data passes through various stages, leading to potential latency and performance issues.

2.Complexity in Small Systems: For small-scale applications, this architectural style may introduce unnecessary complexity.

3.Tight Coupling Between Layers: If not designed properly, layers can become tightly coupled, especially if upper layers start to depend on details of lower layers, leading to maintenance challenges.

4.Difficulty in Modifications Across Layers: Changes in one layer can impact others, especially in tightly coupled implementations.

Real-Time Smart Traffic Management System (STMS)

RECALL

Data Flow Architecture in a Smart Transportation System

Pros:

- **Real-Time Data Processing**
- **Scalability:** Adding new data sources (e.g., more sensors) can be straightforward
- **High Flexibility:** It is easy to introduce new processing stages, such as advanced traffic prediction models, without disrupting the flow of other components.
- **Concurrency:** Components can be designed to process data concurrently, allowing the system to handle large volumes of real-time data streams (e.g., from sensors, traffic lights, cameras).
- **Modular Updates:** Individual processing units (e.g., accident detection, traffic light control) can be updated independently without affecting the overall data flow.

Cons:

- **Complex Debugging:** Since data flows continuously between components, it can be difficult to pinpoint where issues arise when the system malfunctions.
- **Latency Concerns:** If not carefully managed, the data flow may experience delays when there's a bottleneck in any component, which could negatively affect time-sensitive actions like real-time traffic light control.
- **Tight Coupling:** The components can become tightly coupled if they are directly dependent on the data output of other components. This can make the system harder to maintain as it grows.
- **Data Integrity Risks:** Since data flows rapidly between components, ensuring consistent data at all stages becomes difficult. Real-time systems that process concurrent data streams are susceptible to inconsistencies without careful synchronization.

Real-Time Smart Traffic Management System (STMS)

Layered approach

Presentation Layer:

- **Mobile App**: Provides real-time traffic updates to users. It communicates with the application layer to fetch traffic information and alerts.
- **Dashboard Interface**: For traffic control authorities to monitor the system in real time. It displays traffic conditions, sensor data, incidents, and overall system health.
- **Alerts and Reports**: Displays alerts for traffic light malfunctions, accidents, and sensor issues. Also generates reports for historical data analysis.

Challenges: The interface must be lightweight for users while handling a large amount of real-time data coming from multiple locations. Designing an intuitive and responsive UI for both citizens and authorities is crucial.

Application Layer:

- **Traffic Data Processing Module**: Responsible for processing real-time traffic data from sensors, detecting incidents (e.g., accidents or congestion), and making decisions about traffic light adjustments.
- **Traffic Light Control System**: Sends commands to traffic lights to adapt to real-time traffic conditions. This module should have low latency to ensure quick response times.
- **User Interaction Module**: Communicates with the mobile app and dashboard to fetch and display processed traffic data. Also handles routing algorithms to suggest alternative paths based on real-time conditions.

- **Challenges:** It must handle concurrency for managing multiple data streams from sensors and handling user interactions without compromising performance. Achieving minimal latency in traffic light control is vital.

Service Layer:

- **Real-Time Data Aggregation Service**: Aggregates real-time data from various sensors, such as cameras, speed detectors, and weather monitors. It provides a clean data stream to the application layer for further processing.
- **Routing and ETA Calculation Service**: Uses the aggregated data to compute optimal routes for users and estimate their arrival times.
- **Incident Detection Service**: Continuously monitors the incoming data to detect anomalies such as traffic jams or accidents using predefined algorithms and machine learning models.

Challenges: Scalability is important here, as the system needs to handle a potentially enormous amount of sensor data. Ensuring the real-time processing of this data while minimizing delays is critical.

Data Access Layer:

- **Sensor Data Management**: Manages incoming data from various sensors spread across the city. This includes data from cameras, speed detectors, and weather monitors.
- **Traffic Light Control Logs**: Keeps a log of every traffic light's status and the commands sent to it, allowing for auditing and debugging.
- **Historical Traffic Data Storage**: Stores historical traffic and incident data for generating reports and trend analysis. This storage is used for long-term planning and optimization.

Challenges: Ensuring reliable data flow from sensors and managing any potential data inconsistencies or outages is crucial. The system must maintain an efficient database structure for storing and querying large amounts of real-time and historical data.

Integration Layer:

- **API Gateway**: Provides a secure and scalable interface between the system and external components (e.g., third-party applications, neighboring city systems). It also facilitates communication between layers by abstracting the complexity of services.
- **External Data Integration**: Integrates with external systems like weather forecasts, mapping services, and emergency response systems. For example, severe weather data could influence traffic light behavior.

Challenges: Handling real-time external data while ensuring that the system remains responsive is a key concern here. API scalability is essential as the system grows to serve more users and integrate more data sources.

Infrastructure Layer:

- **Message Queues**: Manages real-time data streams from sensors. It helps decouple data producers (sensors) and consumers (data aggregation and processing services) to avoid overwhelming the system.
- **Load Balancing**: Ensures high availability and reliability of the system, particularly when handling high traffic loads from sensors or user requests.
- **Scalable Cloud Storage**: Stores both real-time and historical traffic data. The infrastructure should support the elasticity required to scale up as the number of sensors or volume of data grows.

Challenges: Providing the infrastructure that supports both real-time and scalable data processing while ensuring fault tolerance and recovery is crucial. Additionally, managing high availability is key, especially during peak traffic periods.

Pros:

- **Separation of Concerns:** Each layer handles a specific function (e.g., sensor data processing, traffic management, user interaction). This separation enhances maintainability and makes the system easier to understand and modify.
- **Modular Design:** Changes in one layer (e.g., the presentation layer, to improve the dashboard UI) do not directly affect other layers, which makes the system more adaptable and maintainable.
- **Security:** Layered architecture provides natural security boundaries between components, making it easier to control access to sensitive parts of the system (e.g., traffic light control).
- **Testability:** Individual layers can be tested in isolation, allowing for better unit and integration testing, which is useful in mission-critical systems like traffic management.
- **Fault Isolation:** If one layer fails (e.g., sensor data aggregation), other layers can continue functioning to some extent (e.g., displaying cached data on the user app).

Cons:

- **Latency:** Each layer adds additional overhead due to inter-layer communication. In a real-time system, this can introduce latency, which is undesirable when processing time-sensitive data.
- **Less Flexible for Real-Time Data Processing:** The rigid structure of layers may not be ideal for real-time systems where data must flow continuously and quickly. Real-time operations (e.g., immediate traffic light adjustments) may face delays as they must pass through multiple layers.
- **Complex Integration:** The integration between layers, especially in large systems like a smart transportation system, can become complex. Inter-layer communication protocols, data transformations, and handling edge cases between layers may introduce development overhead.
- **Scalability Issues:** A layered system may face difficulties in scaling horizontally for handling large volumes of data, as adding more layers or services can increase communication overhead.

Which Architecture Best Suits the Transportation System?

Best Fit: Data Flow Architecture

Real-Time Data Handling:

•A transportation system requires real-time processing of data from numerous sensors, cameras, and traffic lights.

•Data flow architecture is inherently designed to process continuous streams of data, allowing the system to react quickly to traffic changes and incidents without unnecessary delays from inter-layer communication.

Low Latency Requirements:

•Traffic light control and congestion management are time-sensitive operations.

•The layered architecture introduces overhead by forcing data to pass through multiple layers, while the data flow architecture can directly connect components, minimizing latency.

Concurrency and Scalability:

- Smart transportation systems involve multiple parallel data streams (e.g., traffic sensor data, weather data, user location data).
- A data flow system allows for parallel processing of these streams without the hierarchical constraints of layered architecture, making it easier to scale as new sensors or data types are added.

Flexibility and Real-Time Adjustments:

- In a data flow architecture, adding new processing stages (e.g., for more sophisticated traffic prediction or advanced anomaly detection) can be done with minimal disruption to the system.
- This adaptability is key for continuously improving smart transportation systems.

When to Consider Layered Architecture?

A **layered architecture** may still be useful for specific subsystems that are less time-sensitive

•**Historical Data Analysis**: If the system needs to generate reports on traffic trends and incidents, a layered approach would isolate the user interface, business logic, and data access layers effectively.

•**User-Facing Applications**: The mobile app and dashboard can adopt a layered approach to separate concerns like user interface, traffic data processing, and data storage without affecting real-time traffic operations.

Non-Functional Requirement	Data Flow Architecture	Layered Architecture	Best Fit
Flexibility	Highly flexible; new processing stages can be integrated easily to adapt to changing requirements.	Moderate flexibility; changes in one layer may require adjustments in others, affecting overall adaptability.	Data Flow Architecture
Scalability	High scalability; easily accommodates new data streams (e.g., sensors) without affecting existing flows.	Moderate scalability; adding layers or services can introduce complexity and overhead, potentially slowing down the system.	Data Flow Architecture

Non-Functional Requirement	Data Flow Architecture	Layered Architecture	Best Fit
Maintainability	Moderate maintainability; debugging can be complex due to tightly coupled components and real-time data flow.	High maintainability; clear separation of concerns allows for easier updates and modifications in individual layers.	Layered Architecture
Performance	Optimized for real-time processing; can handle high volumes of data with low latency if well-designed.	Potentially slower due to inter-layer communication; performance can degrade with increased layers and complexity.	Data Flow Architecture

Non-Functional Requirement	Data Flow Architecture	Layered Architecture	Best Fit
Fault Tolerance	Moderate fault tolerance; failure in one component can affect data flow but can be mitigated with proper design.	High fault tolerance; issues in one layer can be isolated, allowing other layers to function normally.	Layered Architecture
Testability	Moderate testability; unit testing individual components can be challenging due to the continuous flow of data.	High testability; each layer can be tested independently, ensuring more robust unit and integration tests.	Layered Architecture

Non-Functional Requirement	Data Flow Architecture	Layered Architecture	Best Fit
Data Consistency	Challenging to maintain data consistency due to rapid data flow between components; potential for real-time data inconsistencies.	Generally better at maintaining data consistency due to structured processing and validation at each layer.	Layered Architecture
Security	Moderate security; while individual components can be secured, the open flow of data may pose risks.	High security; defined boundaries between layers help enforce access control and data protection measures.	Layered Architecture

Data Flow Architecture excels in **scalability, flexibility, performance**, and is well-suited for real-time processing in a smart transportation system. It allows for quick adjustments and easy integration of new data sources, making it ideal for dynamic environments.

Layered Architecture offers advantages in **maintainability, testability, fault tolerance, data consistency, and security**. Its clear separation of concerns facilitates easier debugging, updating, and securing of different parts of the system, making it a better fit for subsystems that require more stability and robustness.

Few real-life applications that effectively utilize a **layered architecture**:

---

1. Online Banking System
2. Content Management System (CMS)
3. E-Learning Platform
4. Customer Relationship Management (CRM) System
5. Healthcare Management System

That’s it