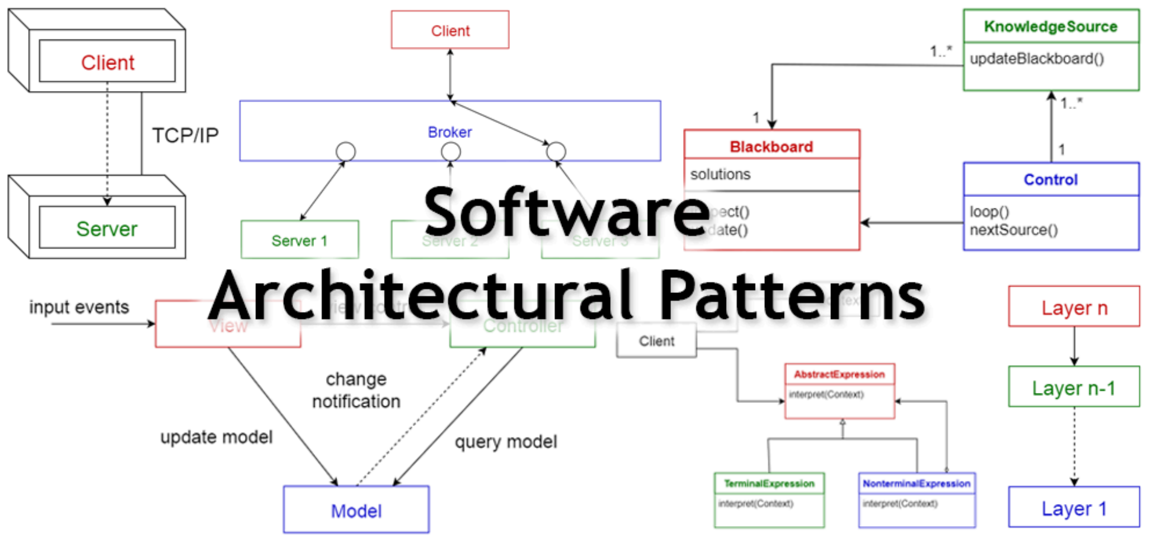
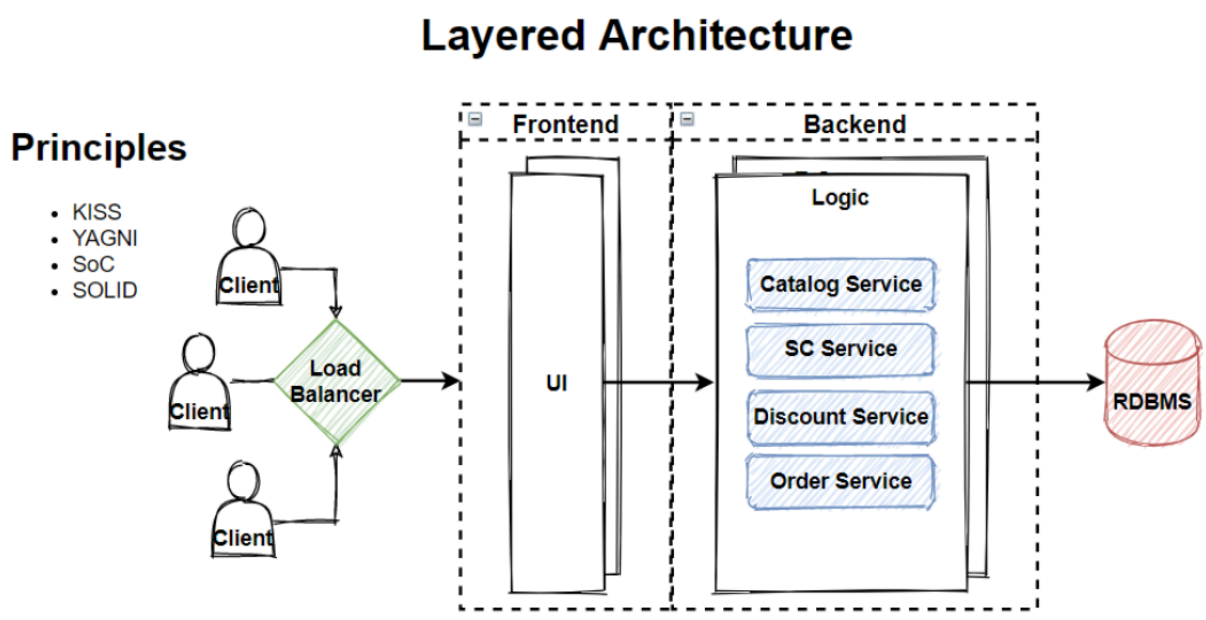
Architectural Patterns



Definition:
Architectural patterns are reusable solutions to common problems in software architecture. They define the overall structure and interaction of components in a system.

1.1 Layered Architecture (n-tier architecture)



Description:

- Organizes the system into layers, each with specific responsibilities.

- Common layers: **Presentation**, **Business Logic**, **Data Access**, and **Database**.
- Data flows from the top layer (UI) to the bottom (database) and vice versa.

Example: Web applications where the UI interacts with business logic, which communicates with the database.

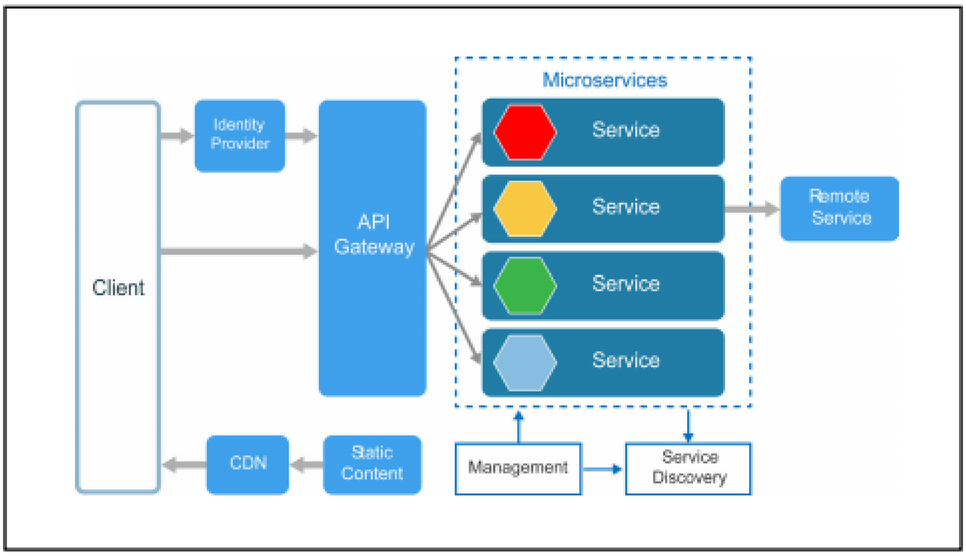
Benefits:

- Separation of concerns.
- Easy to maintain and test.
- Reusable layers across applications.
- Supports incremental development.

Trade-offs:

- Can become inefficient due to multiple layer crossings.
- Hard to change dependencies between layers.
- Not ideal for high-performance systems.

1.2 Microservices Architecture



Description:

- System is divided into small, independent services that communicate via APIs (often REST or message queues).
- Each service performs a specific business function and can be deployed independently.

Example: Netflix, Amazon, and Uber systems.

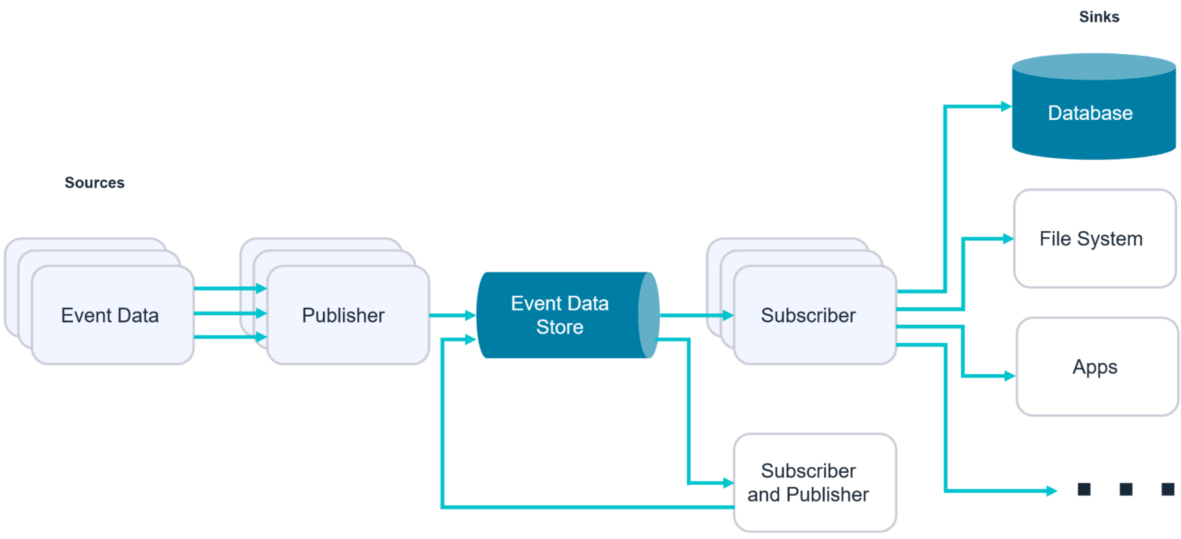
Benefits:

- Scalability — scale individual services as needed.
- Flexibility in using different technologies per service.
- Fault isolation — failure in one service doesn't affect the whole system.
- Continuous deployment and faster updates.

Trade-offs

- Complex deployment and communication management.
- Requires strong DevOps practices.
- Difficult debugging and monitoring.
- Data consistency challenges.

1.3 Event-Driven Architecture (EDA)



Description:

- Components communicate through events.
- Producers generate events, and consumers (subscribers) react to them asynchronously.
- Typically uses message brokers like Kafka, RabbitMQ, or AWS SNS.

Example: Real-time stock trading systems, IoT applications.

Benefits:

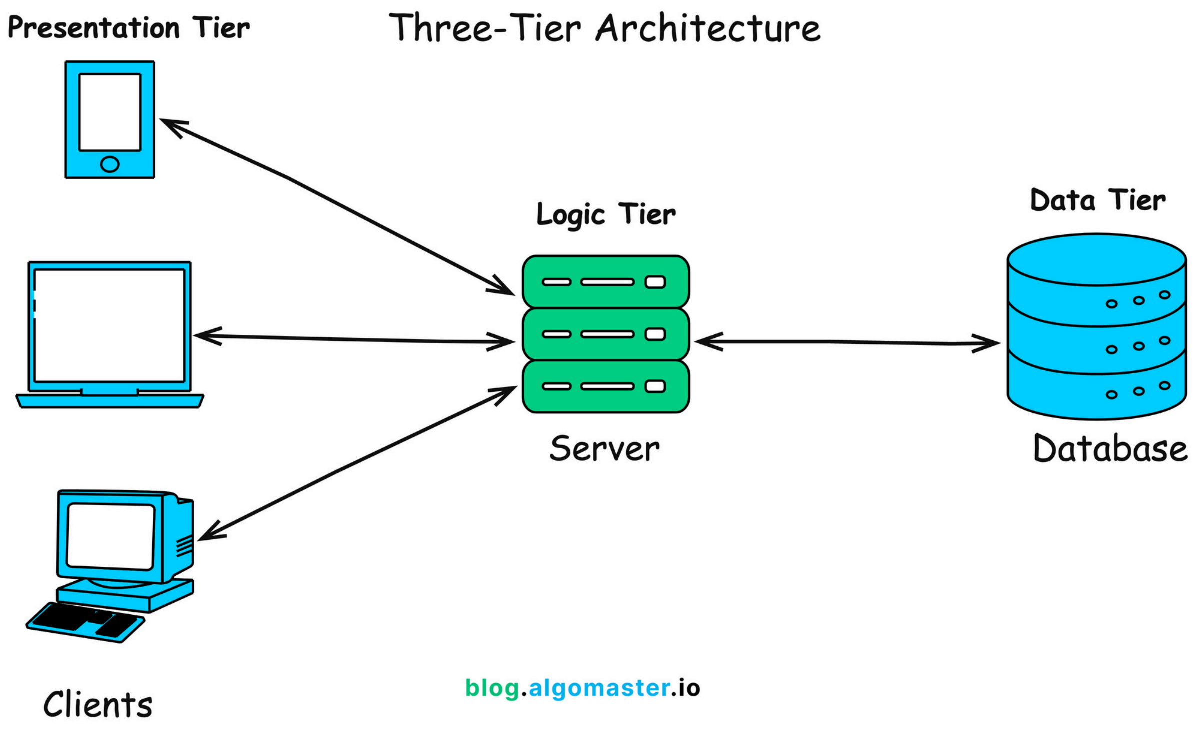
- High scalability and flexibility.
- Loose coupling between services.

- Real-time responsiveness.

Trade-offs:

- Debugging and tracing events can be difficult.
- Potential for duplicate or lost messages.
- Requires careful design for event ordering and consistency.

1.4 Client-Server Architecture



Description:

- System is divided into two parts: **clients** (requesters) and **servers** (responders).
- The server provides services or resources that clients consume.

Example: Web browsers (clients) communicating with web servers.

Benefits:

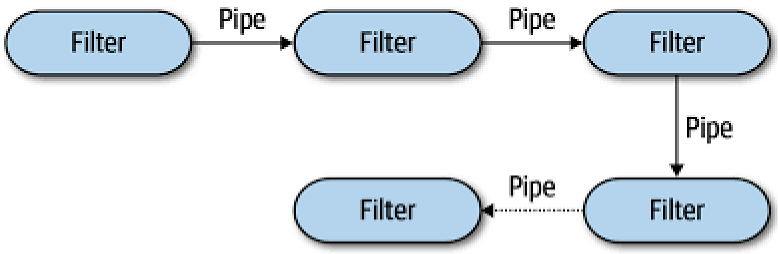
- Centralized control and data management.
- Scalable (by adding more servers).
- Easy to secure and update the server.

Trade-offs:

- Server becomes a single point of failure.

- High load can reduce server performance.
- Network dependency — clients need an active connection.

1.5 Pipe and Filter Architecture



Description:

- Data passes through a series of processing components (filters) connected by pipes.
- Each filter transforms data and passes it on.

Example: Compilers, data processing pipelines.

Benefits:

- Reusability of filters.
- Easy to understand and maintain.
- Supports parallel execution.

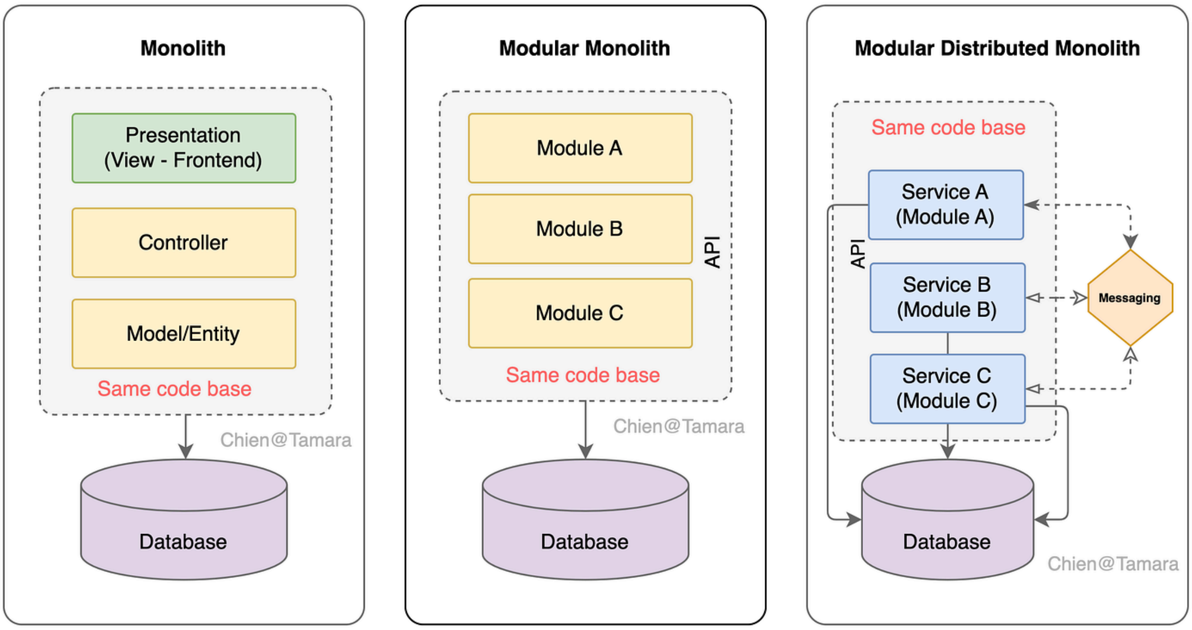
Trade-offs:

- Not suitable for systems requiring complex interactions.
- Data format compatibility between filters must be maintained.

Architectural Styles

Definition:
Architectural styles are broader categories that define how components and connectors interact, influencing the system's structure and design philosophy.

2.1 Monolithic Architecture



Description:

- Entire application is built as a single unit.
- All components (UI, business logic, data access) are tightly coupled and deployed together.

Example: Traditional web applications or early enterprise systems.

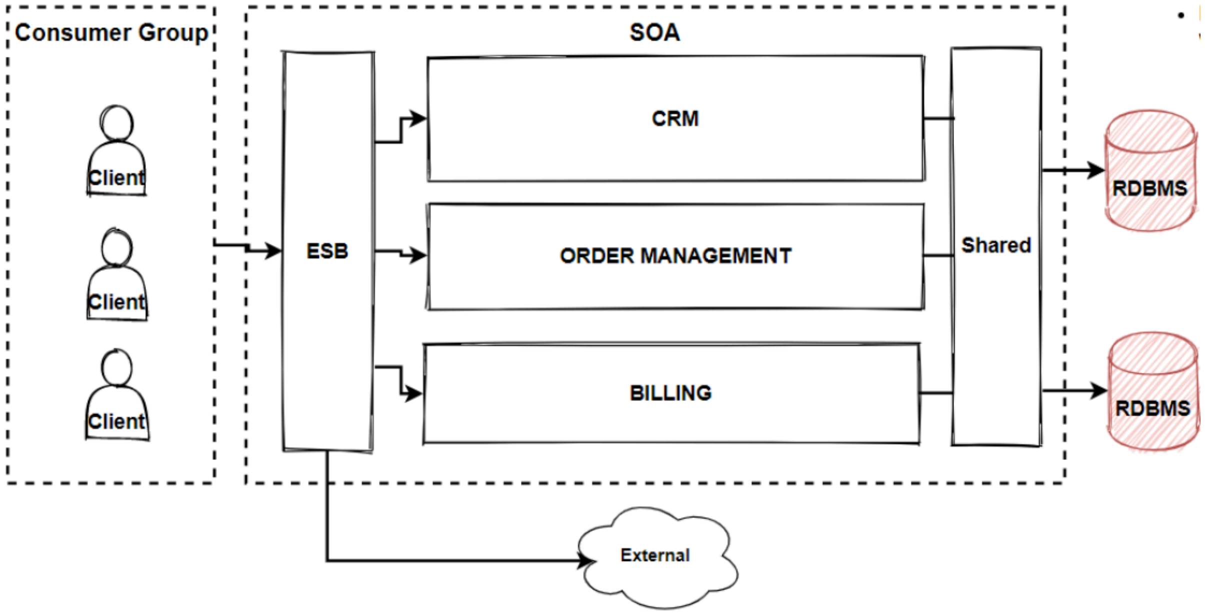
Benefits:

- Simple to develop, test, and deploy.
- Easier debugging (single codebase).
- Performance benefits (no inter-service latency).

Trade-offs:

- Hard to scale individual components.
- Difficult to maintain as the codebase grows.
- Any change requires redeployment of the entire system.

2.2 Service-Oriented Architecture (SOA)



Description:

- System is composed of reusable services that communicate via standard protocols (like SOAP, XML, HTTP).
- Focuses on interoperability and integration across platforms.

Example: Enterprise applications integrating banking, HR, and logistics services.

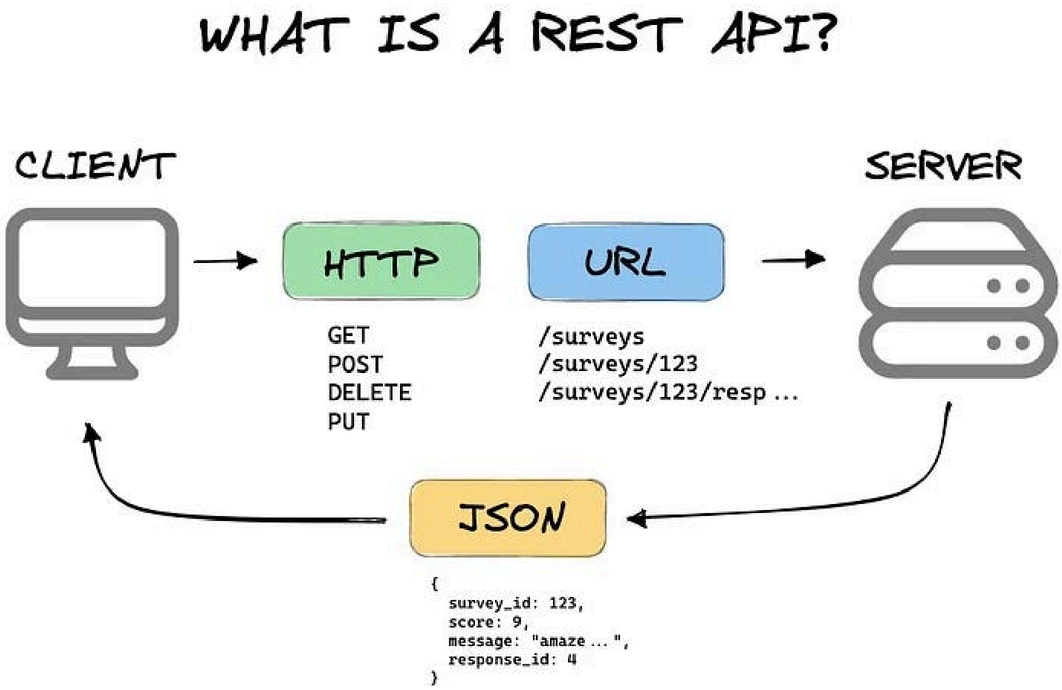
Benefits:

- Reusability of services across systems.
- Interoperability between heterogeneous systems.
- Centralized governance and control.

Trade-offs:

- Complex service management.
- Higher latency due to service calls.
- Governance overhead can slow down development.

2.3 RESTful Architecture



Description:

- Based on **Representational State Transfer (REST)** principles using HTTP methods (GET, POST, PUT, DELETE).
- Stateless client-server communication.

Example: Modern web and mobile APIs (e.g., Twitter API, GitHub API).

Benefits:

- Lightweight and scalable.
- Simple and widely adopted.
- Language-agnostic (works across platforms).

Trade-offs:

- Statelessness may lead to repeated data transfers.
- Limited to HTTP.
- Can be less efficient for complex, real-time interactions compared to GraphQL or gRPC.

2.4 Serverless Architecture

Description:

- Application runs on cloud functions triggered by events, without managing servers.
- The cloud provider handles scaling and infrastructure.

Example: AWS Lambda, Google Cloud Functions.

Benefits:

- Automatic scaling.
- Pay-per-use cost model.
- Simplifies operations and deployment.

Trade-offs

- Cold start latency issues.
- Limited execution time.
- Vendor lock-in risk.

Benefits and Trade-offs of Architectural Patterns

Pattern/Style	Key Benefits	Main Trade-offs
Layered	Easy maintenance, clear structure	Slower performance, rigid dependencies
Microservices	Scalability, flexibility, fault isolation	Complex management, communication overhead
Event-Driven	Real-time processing, loose coupling	Debugging difficulty, potential inconsistency
Client-Server	Centralized management, modular	Server bottlenecks, network dependency

Monolithic	Simple to develop/test	Hard to scale, tightly coupled
SOA	Reusability, interoperability	Governance overhead, complexity
RESTful	Lightweight, standardized	Statelessness, limited protocol support
Serverless	Auto-scaling, reduced ops cost	Cold starts, vendor dependency

Summary

- **Architectural patterns** define reusable solutions for structuring applications.

- **Architectural styles** define high-level philosophies or models of organizing systems.
- Choosing the right pattern or style depends on:
  - Project size and complexity
  - Scalability and maintainability needs
  - Team expertise
  - Deployment environment