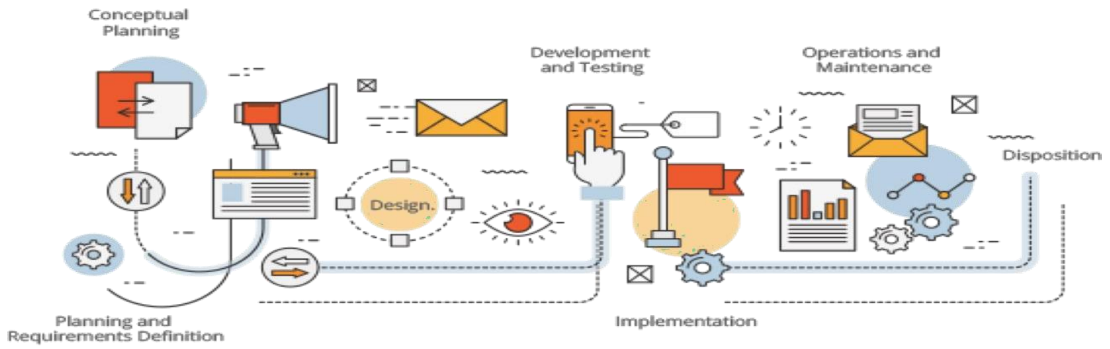
# Software Engineering

Requirement Engineering

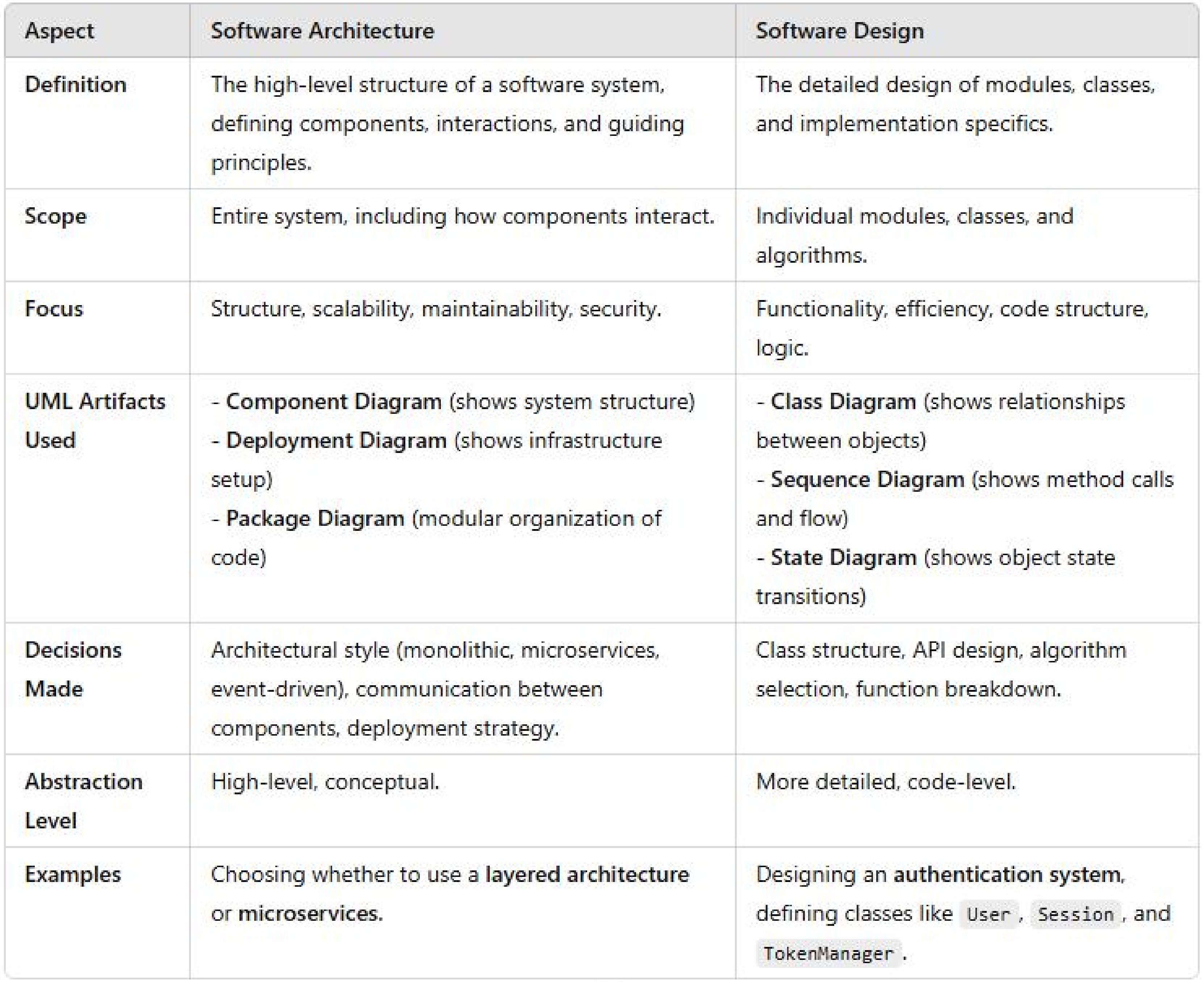


## Design and Architecture

- The terms "design" and "architecture" are often used interchangeably in software development, but they refer to different aspects of the software development process.

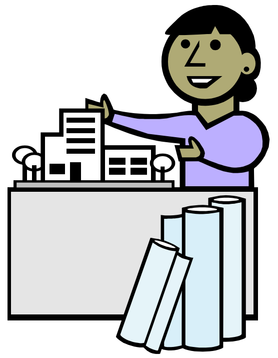
| Design | Architecture |
|---|---|
| • Design focuses on the detailed specification and implementation of individual components within a software system. | • Architecture focuses on the high-level structure and organization of the system as a whole. |
| • Design deals with the "how" of software development | • architecture deals with the "what" and "why," |
| • Examples of design activities include creating class diagrams, sequence diagrams, database schemas, user interface wireframes, and detailed algorithms for implementing specific functionality | • Examples of architectural decisions include choosing between monolithic and microservices architectures, selecting appropriate architectural patterns (e.g., MVC, layered architecture), defining system boundaries and interfaces, and designing the overall system structure. |

Ø Architecture defines the system’s overall structure, guiding design decisions.
Ø Design refines architecture into implementable code, ensuring maintainability.



| Aspect | Software Architecture | Software Design |
|---|---|---|
| **Definition** | The high-level structure of a software system, defining components, interactions, and guiding principles. | The detailed design of modules, classes, and implementation specifics. |
| **Scope** | Entire system, including how components interact. | Individual modules, classes, and algorithms. |
| **Focus** | Structure, scalability, maintainability, security. | Functionality, efficiency, code structure, logic. |
| **UML Artifacts Used** | - **Component Diagram** (shows system structure)<br>- **Deployment Diagram** (shows infrastructure setup)<br>- **Package Diagram** (modular organization of code) | - **Class Diagram** (shows relationships between objects)<br>- **Sequence Diagram** (shows method calls and flow)<br>- **State Diagram** (shows object state transitions) |
| **Decisions Made** | Architectural style (monolithic, microservices, event-driven), communication between components, deployment strategy. | Class structure, API design, algorithm selection, function breakdown. |
| **Abstraction Level** | High-level, conceptual. | More detailed, code-level. |
| **Examples** | Choosing whether to use a **layered architecture** or **microservices**. | Designing an **authentication system**, defining classes like `User`, `Session`, and `TokenManager`. |

# Software Architecture



- Software architecture refers to the high level structures of a software system, the discipline of creating such structures, and the documentation of these structures. These structures are needed to reason about the software system.

- Architectural patterns provide reusable solutions to common high-level software design problems. They define the **structure, behavior, and interactions** between components in a system.

## Architectural Patterns

- The fundamental problem to be solved with a **large system is how to break it into chunks** manageable for human programmers to understand, implement, and maintain.

- Architectural patterns serve as standardized **solutions to common design challenges** in software engineering, abstracting away implementation details and promoting scalability, flexibility, and maintainability.

- By facilitating modular design, clear separation of concerns, and **reuse of design components**, these patterns address various quality attributes such as performance, reliability, security, and usability. Architectural patterns also promote interoperability and enable seamless integration between heterogeneous systems.

- Save time and effort by providing **proven solutions to recurring design problems**, allowing developers to focus on high-level design decisions and adapt their systems to changing requirements with ease.

## The Impact of Architecture Styles on Quality Attributes in Software Development

- Architecture styles exert significant control over the quality attributes of a software system.
- By shaping the system's structure, interactions, and components, architecture styles directly influence key aspects such as performance, reliability, security, scalability, and maintainability.
- Decisions regarding architecture styles, including patterns and paradigms, play a critical role in defining and prioritizing these quality attributes throughout the software development process.

- **Layered Architecture:** This pattern organizes the system into multiple layers (e.g., presentation layer, business logic layer, data access layer) where each layer has a specific responsibility. It promotes <u>modularity, scalability, and maintainability</u> by separating concerns and enforcing clear dependencies between layers.

- **Client-Server Architecture:** The client-server architectural pattern offers a balance of <u>scalability, reliability, performance, security, maintainability, and interoperability</u>, making it suitable for a wide range of distributed systems and networked applications.

- **Event-Driven Architecture (EDA):** EDA decouples system components by using asynchronous communication through events. It promotes <u>scalability, flexibility, and responsiveness</u> by allowing components to react to events and handle them independently.

- **Service-Oriented Architecture (SOA):** This pattern organizes the system into reusable services that are loosely coupled and interoperable. It promotes <u>reusability, interoperability, and flexibility</u> by encapsulating business logic into modular services with well-defined interfaces.

- **Pipe and Filter architectural**: The Pipe and Filter architectural pattern offers a balance of <u>modifiability, reusability, scalability, flexibility, performance, testability, and maintainability</u>, making it suitable for systems that require efficient data processing with clear separation of concerns.

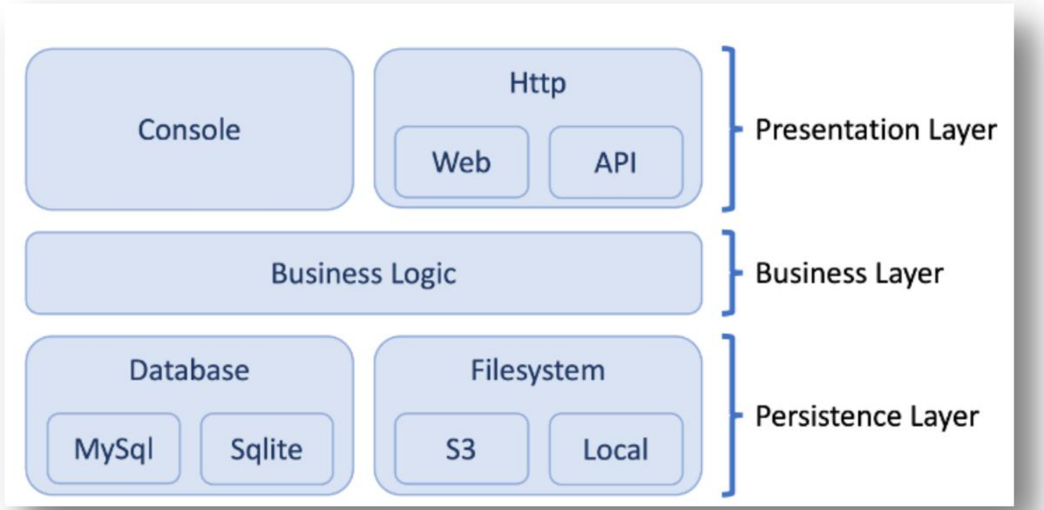
# Architecture Patterns

1. Layered pattern
2. Client-server pattern
3. Micro service architecture
4. Pipe-filter pattern
5. Model-view controller architecture
6. Repository Architecture
7. Event-based Architecture

## 1. Layered pattern

- Layered architecture is a design pattern commonly used in software development to organize a system's components into distinct layers, each responsible for a specific set of functionalities.
- This pattern promotes **modularity, scalability, and maintainability** by separating concerns and enforcing clear dependencies between layers.
- The layered architecture pattern typically consists of the following layers, from bottom to top:

  - Ø **Presentation layer** (also known as **UI layer**)
  - Ø **Business logic layer** (also known as **domain layer**)
  - Ø **Data access layer** (also known as **persistence layer**)



## 1. Layered Architecture (N-Tier Architecture)

🔷 **Concept**: Organizes software into layers, where each layer has a specific responsibility.

🔷 **Layers:**

- **Presentation Layer** – UI/Frontend (e.g., React, Angular).
- **Business Logic Layer** – Handles core processing (e.g., Spring Boot, Django).
- **Data Access Layer** – Communicates with databases (e.g., SQL, NoSQL).
- **Database Layer** – Stores and manages data.

🔷 **Use Cases:** Enterprise applications, web apps, and banking systems.

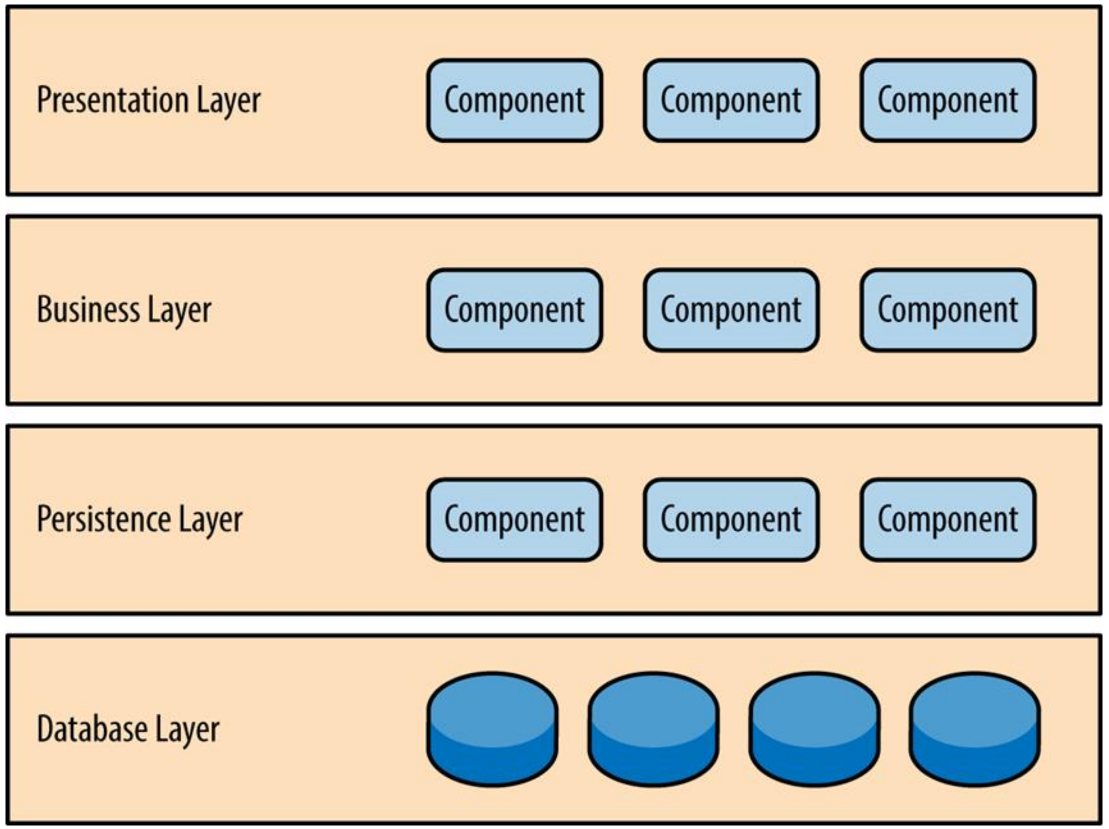
🔷 **Pros:** Easy to maintain, modular, separation of concerns.

🔷 **Cons:** Can be slow due to multiple layers, rigid for scaling.

- The **presentation layer** contains all of the classes responsible for presenting the UI/visualization to the end-user and handling bowser communication logic. Ideally, this is the only layer that customers interact with.

- The **business/logic layer** contains all the logic that is required by the application to meet its functional requirements. This layer usually deals with data aggregation, computation and query requests. This is where the main logic of the application is represented.

- The **data/physical + persistence layer** is where retrievable information is stored. This layer consists of both logical and physical aspects. While the logical schema specifies conceptual model of data, the physical schema implements the logical model into physical database platform



## Applicable problems

- Software that requires separate layers of processing and security.
- Data driven software, CRUD applications.

## Resilience to change

- Since the separation of concerns is the main property of the architecture, each layer of software has a specific function. This makes updating individual layers easy, and also allows teams to separate workloads.

## Negative behaviours

- Layered software usually results in tightly coupled software components, and monolithic applications.
- Security can become an issue if bypassing layers is allowed. The Business Layer usually acts as an integrity check for passing data.
- If not properly designed and managed, communication between layers can become complicated.

## Supported NFPs

- Easy to implement/test: Layered architecture is one of the easiest structures to implement. Since every layer has a specific function, testing is easy since layers can be mocked
- Flexibility: In some sense, any software can be abstracted into layers. Layered architecture is very open and broad in its implementation, thus leading to many practical applications.
- Security: Security can be implemented at every layer. If layers are skipped, extra precautions must be made.

## Inhibited NFPs

- Low scalability: Layered architecture results in a rigid structure of software implementation and highly coupled software groups, resulting in a system that is hard to scale and hard to update. A change to a single layer must be verified that it does not crash the entire system.
- Low Performance: Data must travel through every layer and processed, slowing down the performance.

## Comparison with other architectures

- Client-server, layered, and pipe and filter architectures are similar in their objective.
  - Client-server can be thought of as a variation of layered architecture with two layers.
  - Pipe and filter only allows unidirectional flow of information, whereas client-server and  layered architectures allow bidirectional flow.

**Layered Architecture:**

•Example: Online Banking System

1. Presentation Layer: Web interface for users to interact with the banking system.
2. Business Logic Layer: Handles transactions, account management, and authentication.
3. Data Access Layer: Interfaces with databases to retrieve and update customer account information.

## 2. Client-server architecture

- Client-server architecture is a computing model in which the server hosts, delivers and manages most of the resources and services to be consumed by the client.
- This type of architecture has one or more client computers connected to a central server over a network or internet connection.
- Client-server architecture is also known as a networking computing model or client-server network because all the requests and services are delivered over a network.

## 2. Client-Server Architecture

- **Concept**: A centralized **server** serves multiple **clients** that request data.
- **Examples**: Web applications (browser as client, backend as server).
- **Use Cases:** Websites, mobile applications, databases.
- **Pros:** Centralized control, scalable with load balancing.
- **Cons:** Server can become a bottleneck, requires good security.

## Client server architecture example

Client-server architecture is used in many different types of applications. Here are two examples:

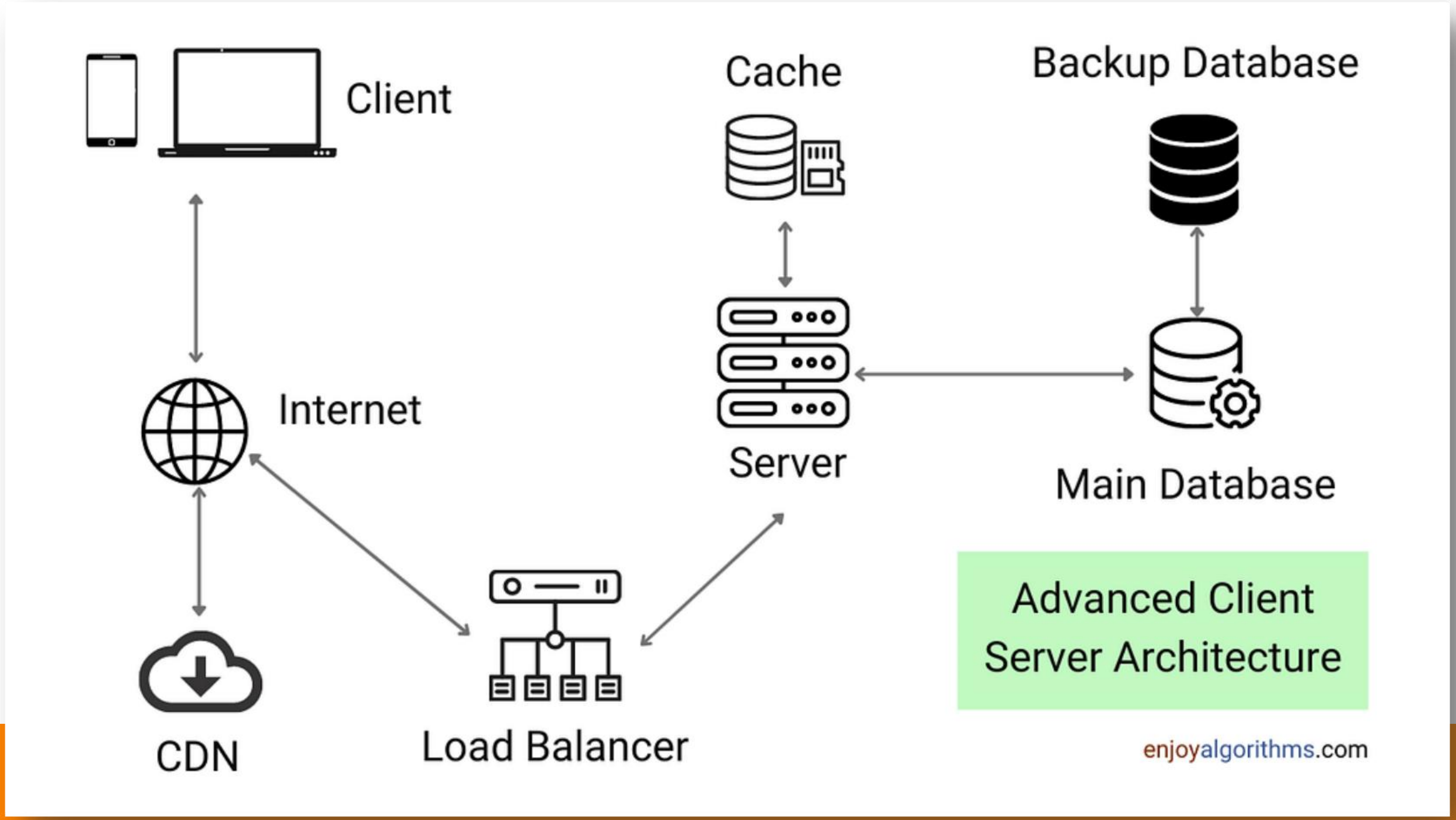
**Healthcare application:** A client computer will be running an application to enter patient information. At the same time, a server computer will be running another code to retrieve and manage database system.

**Banking application:** When a bank customer accesses banking services with a web browser (client), web browser initiates a request to the webserver. Customer's login credentials will be stored in a database, and webserver accesses database server as a client. Now application server process the returned data by applying the business logic of banking application and provide output to the webserver. Finally, webserver returns the result to the client (web browser) for display.

Other applications that use the client-server architecture include email, the World Wide Web, and network printing.

A typical topological data flow goes as follows:

1. Client requests data from server
2. Load balancer routes the request to the appropriate server
3. Server processes the request client
4. Server queries appropriate database for some data
5. Database returns the queried data back to the server
6. The server processes the data and sends the data back to the client.
7. This process repeats



## Negative behaviors

- Could be a greater potential for failure because of centralized control: If the servers go down then entire service goes down.

- Particularly vulnerable to Denial of Service (DOS) attacks because the number of servers is considerably smaller than the number of clients.

- Very expensive to install and manage the network. Server machines are much more powerful than standard client machines, which means that they are more expensive. Requires hiring employees with networking and infrastructure knowledge, in order to manage the system

## Supported NFPs

- Scalability: capable of horizontal or vertical scaling of the system
  - Horizontal scaling: adding or removing servers in the network
  - Vertical scaling: migrating to larger and faster server machines
- Availability: servers typically do not have to shutdown or restart for many days
  - Server uptime is possible during maintenance, with server duplication
- Dependability: processing and resource allocation is done by a dedicated set of machines
  - These servers can be optimized to complete a certain task quickly and efficiently
- Heterogeneity: clients are consumers of services and servers are providers of services
  - Clear separation of concerns results in the system being composed of disparate parts
  - For example, one or more servers may fail, but the system can still function as long as the other servers offer the same services

## Client-Server Architecture:

- Example: Email Services (e.g., Gmail)
    - Client: Web or mobile application used by users to access emails.
    - Server: Backend infrastructure responsible for storing emails, managing user accounts, and delivering emails to clients.

## clarity on scalability aspect of layered architecture

Scalability in Layered Architecture refers to the ability of an application to handle increasing workloads efficiently by scaling different layers independently. This ensures that performance remains optimal as user demands grow.

**Types of Scalability**

1. Vertical Scalability (Scaling Up)
   - Increases the power of existing servers (e.g., adding more RAM, CPU).
   - Suitable for small-scale applications but has hardware limitations.

2. Horizontal Scalability (Scaling Out)
   - Adds more servers or instances to distribute the load.
   - Common in cloud-based applications using load balancers.

## How It Works?

- The **Presentation Layer** (UI) can scale horizontally by using multiple web servers with a load balancer (e.g., AWS ELB, Nginx).
- The **Business Logic Layer** (Backend) can scale by deploying multiple application servers.
- The **Data Layer** (Database) can use replication, caching, or sharding for scalability.

## Why Does Layered Architecture Have Low Scalability?

- Layered Architecture is widely used in software design, but it has scalability limitations due to its monolithic nature and **tight coupling between layers**.

### 1. Tight Coupling Between Layers

- Each layer is dependent on the layer below it. Scaling one layer (e.g., business logic) often requires changes in other layers, making it harder to scale efficiently.
- If the business logic layer becomes a bottleneck, scaling it independently is difficult without also scaling the data access layer and database.

### 2. Single Point of Failure

- If one layer fails (e.g., database layer), the entire system is affected.
- A database crash can bring down the entire application, even if the frontend and backend are running smoothly.

## Performance Bottlenecks

- Each request must pass through multiple layers, increasing response time.
- More layers = more processing overhead (especially if each layer calls multiple services).

◈ Example:

A simple user request like "fetch user details" must go through:

UI Layer → Business Layer → Data Access Layer → Database → Response

This extra processing slows down performance under heavy traffic.

## Conclusion: When to Use Layered Architecture?

- Good for small to medium applications where scaling isn’t a priority.
- Not ideal for large-scale applications that require handling millions of users.
- If scalability is a key requirement, microservices or event-driven architecture is a better choice.

## 3. Pipe and Filter architecture pattern

- This software architecture pattern decomposes a task that performs complex processing into a series of separate elements that can be reused, where processing is executed sequentially step by step.

- Pipe and filter is a component based architectural style that allows for the deconstructing of monolithic processes into small independent components.

- The pipe and filter architecture is a software architecture that is data-centric, structured around how data flows through the application. In this architecture, applications employ a pipeline as follows:

  - Firstly, the application takes in data as input.
  - Next, a series of transformations on the data are applied sequentially.
  - Finally, the application returns the processed data as output.

## Example

- Imagine you're cooking in a big kitchen. You have different ingredients and tools to prepare a meal. Now, think of each ingredient or tool as a piece of information or data, and the cooking process as some task or operation you want to perform on that data.

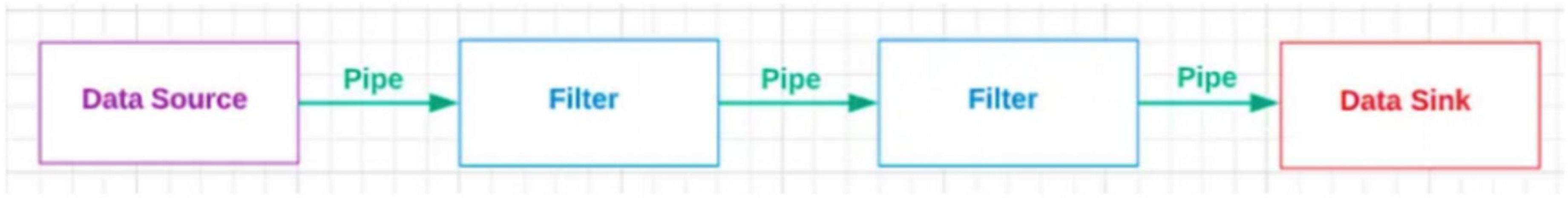
- In a pipe and filter architecture, you organize your kitchen (or your computer program) into different stations or filters. Each filter does one specific job or operation on the data that passes through it. For example, one filter might chop vegetables, another might cook them, and another might mix sauces.

- Now, imagine pipes connecting these filters. These pipes carry the data from one filter to another, just like how ingredients move from one station to another in a kitchen. Each filter only does its own job and doesn't worry about what happens before or after it. It just receives data from the previous filter through a pipe, processes it, and sends it along to the next filter.

- So, in simple terms, a pipe and filter architecture is like a kitchen assembly line where each station (filter) does one specific task, and the ingredients (data) flow through pipes from one station to another until the final dish (output) is ready.

There are four main components:

1. **Data Source**: The original, unprocessed data
2. **Data Sink**: The final processed data
3. **Filter**: Components that perform processing
4. **Pipe**: Components that pass data from a data source to a filter, or from a filter to another filter, or from a filter to a data sink

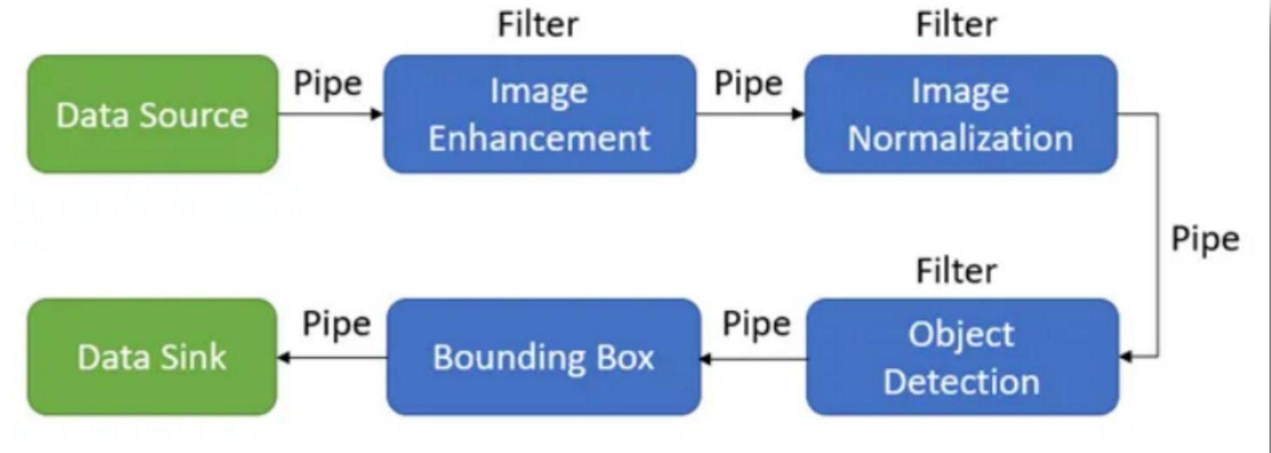


Generic Pipe & Filter Diagram

## Real World Application

- One scenario where the pipe and filter architecture is especially suitable is in the field of video processing, such as in the media-handling library, GStreamer.

- One example of this is in the real-time object detection from a live cameras. We can use this example to illustrate the application of the pipe and filter architecture.



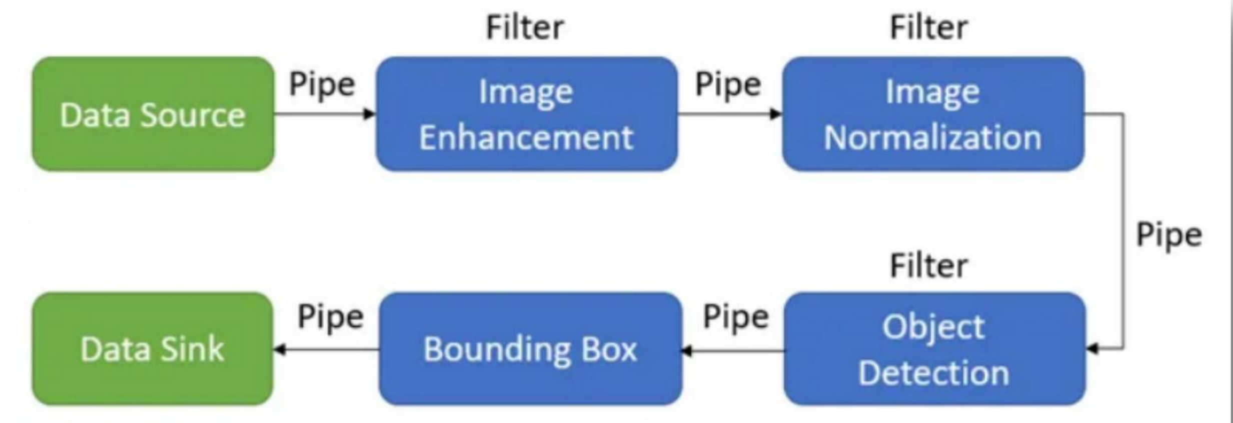
Pipe and Filter for Video Processing — Object Detection

1. The image frames from the live video recorded serve as the input data and are sent into the application via the data source. The data is transported via pipes between each component.

2. From the data source, the data goes through a series of filters sequentially, each processing the data to make it more useful for the next filter order to achieve the eventual goal of object detection.

3. Eventually, the processed data, which in this case is the input image frame with bounding boxes drawn around objects of interest, is served as the application's output in the data sink.

**It is useful to note pipe and filter architecture's flexibility here: many other transformations in video processing can reuse filters implemented above.**



Pipe and Filter for Video Processing — Object Detection

## Advantages

- Suitable for processing that requires clear, systematic steps in order to **transform successive pieces of data**, because of the intuitive flow of processing.
- Each filter can be **modified easily** — as long as the data input and data output remain the same
- Filters are **reusable**, old filters can be replaced by new ones, or new filters can be inserted easily into the flow of processing — as long as the data input and output between filters are compatible.
- Each component is implemented as a separate, distinct task, hence having a natural **separation of concerns**

## Disadvantages

- **Inefficient** and inconvenient to pass around the full set of data throughout the entire pipe and filter system, because not every component will require the full set of data.

- **Reliability may be an issue** if data is lost on the way between components.

- Having too many filters can slow down your application, introducing **bottlenecks or deadlocks** if one particular filter processes slowly or fails

## 4. Microservices architecture

- Microservices architecture is an approach to building software systems as a suite of small, independently deployable services.
- Each service encapsulates a specific business capability and communicates with others over lightweight protocols (often HTTP/REST or messaging systems).
- Microservices architecture aims to improve scalability, flexibility, and maintainability by breaking down a monolithic application into smaller, more manageable services.
- Popular platforms and companies that have adopted microservices architecture include Netflix, Amazon, Uber, and Spotify.

## Core Principles

### 1. Single Responsibility

In an e-commerce platform, different services handle distinct functions. One microservice might manage user authentication, another handles product catalogs, and yet another processes payments. Each service is designed to perform a specific task, making it easier to develop, maintain, and scale independently.

### 2. Decentralization

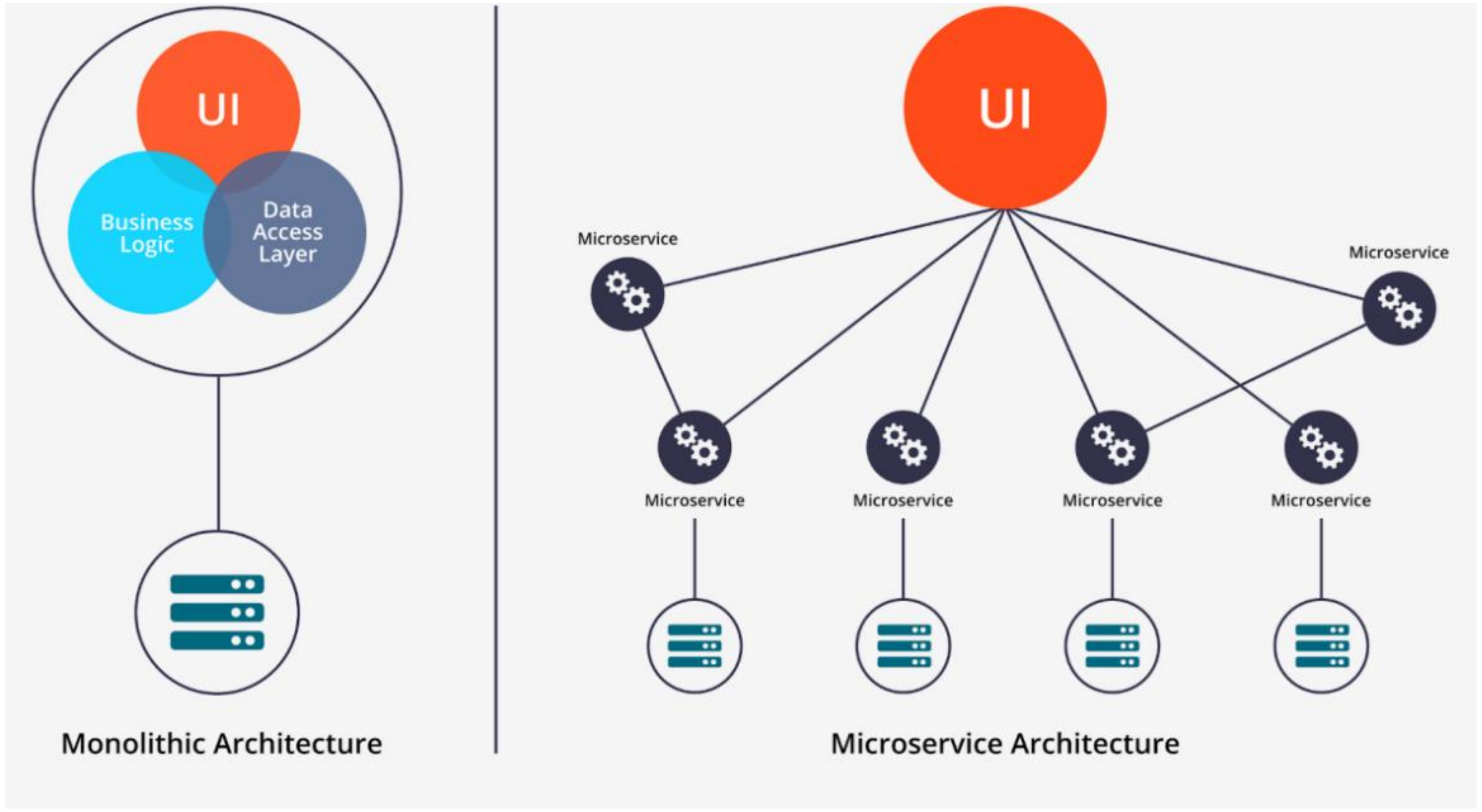
Instead of relying on a centralized database, each microservice can have its own dedicated datastore. For instance, the order management service might use a relational database optimized for transactions, while the product catalog service might use a NoSQL database for quick read operations. This decentralization avoids bottlenecks and allows each service to choose the best data storage solution for its needs.

### 3. Independent Deployment

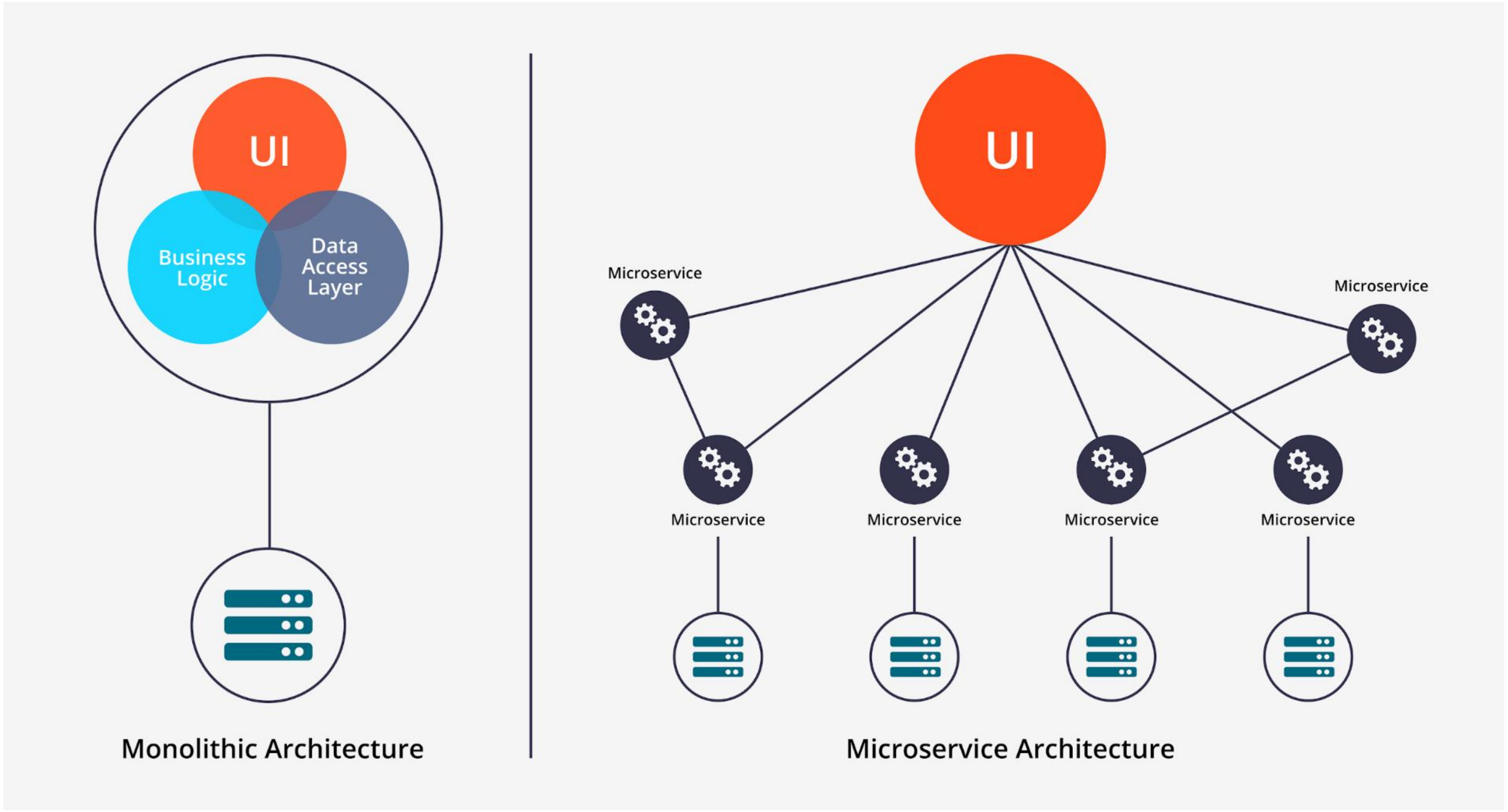
A company like Netflix deploys new features or fixes in one service (such as recommendations) without redeploying the entire system. This means that if there's an update to the billing service, only that microservice is deployed, minimizing risk and reducing downtime across the platform.

## 4. Technology Diversity

In a technology-diverse environment, each microservice can be implemented in the language or framework that best fits its purpose. For example, a high-performance service for real-time data processing might be built using Go, while a service with complex business logic might be implemented in Java or .NET. This allows teams to select the optimal tools for each service’s requirements.



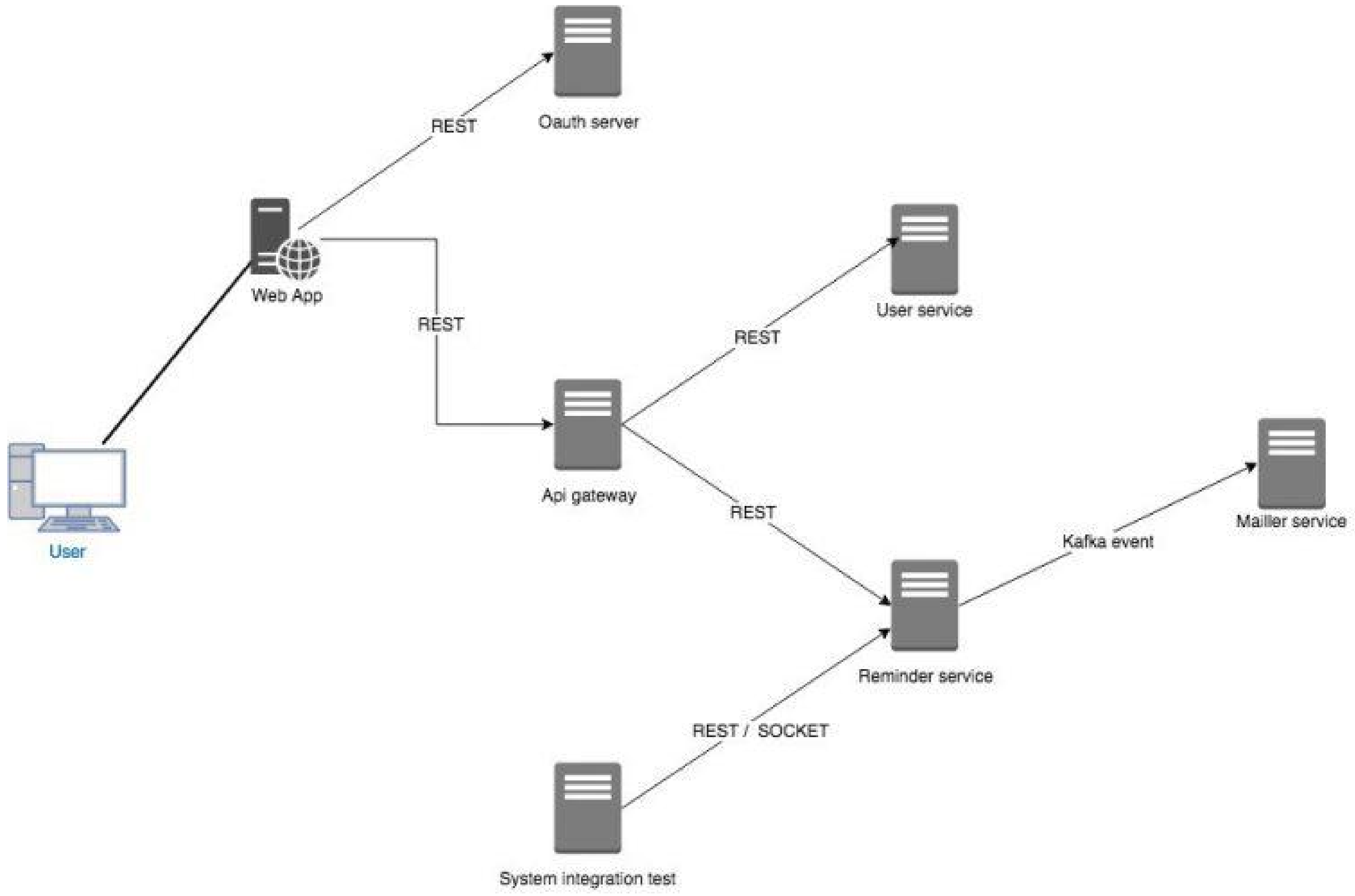
# What is Microservice



Microservice architecture split Business Logic and Data Access Layer into several small part called *Microservices*.
Real business logic and Data access are implemented by invoking multiple different microservices distributed on different nodes.

1. **Service Decomposition:**
    1. In a traditional monolithic e-commerce application, all functionality (e.g., user management, product catalog, order processing) is tightly coupled into a single codebase.
    2. With microservices architecture, the application is decomposed into smaller, independent services based on business capabilities. For example:
        1. User Service: Manages user authentication, registration, and profile management.
        2. Product Service: Handles product catalog management, including adding, updating, and deleting products.
        3. Order Service: Manages order processing, including creating, updating, and canceling orders.
        4. Payment Service: Handles payment processing, including payment validation and authorization.
2. **Independent Development and Deployment:**
    1. Each service is developed and deployed independently by a dedicated team using their preferred programming languages, frameworks, and databases.
    2. For example, the User Service team might choose Node.js with MongoDB, while the Order Service team might use Java with PostgreSQL.
3. **Service Communication:**
    1. Services communicate with each other through well-defined APIs, typically using lightweight protocols such as HTTP/REST or messaging queues.
    2. For example, when a user places an order, the Order Service communicates with the Payment Service to authorize the payment.
4. **Data Management:**
    1. Each service has its own database or data store, ensuring data isolation and minimizing dependencies between services.
    2. For example, the Order Service stores order details in its own database, and the Product Service stores product information in its database.
5. **API Gateway:**
    1. An API gateway acts as a single entry point for clients (e.g., web or mobile applications) to access the microservices. It handles routing, authentication, and other cross-cutting concerns.
    2. For example, the API gateway forwards requests from the client to the appropriate microservice (e.g., User Service, Product Service) based on predefined routing rules.

# Why should we use Microservice



## Why should we use Microservice

- Comparing to the traditional monolithic architecture, Microservice has much stronger **scalability**.

- First, It really benefit from the **distribution**. You dont need to have one very powerful server to run your application anymore. Instead you could use several relatively weaker servers and run different microservices on each of them.

- The system will have better IO performance since the tasks are distributed and can be done simultaneously. And by combinng microservice with edge computing, you can take advantage of IOT devices like raspberry pi or arduino to provide service locally for users.

## Containers are a well-suited microservices architecture example

- Microservices architecture is closely related to the concept of containerization because **containerization** provides a lightweight and efficient way to package, deploy, and manage microservices.
- Microservices are designed to be small, independent services that can be deployed and scaled independently. Containerization, using technologies like **Docker**, provides a way to isolate each microservice in its own container.
- Each container encapsulates the microservice along with its dependencies, libraries, and runtime environment, ensuring that it runs consistently across different environments.

# Model-View-Controller

| Name | MVC (Model-View-Controller) |
|---|---|
| Description | Separates presentation and interaction from the system data. The system is structured into three logical components that interact with each other. The Model component manages the system data and associated operations on that data. The View component defines and manages how the data is presented to the user. The Controller component manages user interaction (e.g., key presses, mouse clicks, etc.) and passes these interactions to the View and the Model. See Figure 6.3. |
| Example | The architecture of a web-based application system organized using the MVC pattern. |
| When used | Used when there are multiple ways to view and interact with data. Also used when the future requirements for interaction and presentation of data are unknown. |
| Advantages | Allows the data to change independently of its representation and vice versa. Supports presentation of the same data in different ways with changes made in one representation shown in all of them. |
| Disadvantages | Can involve additional code and code complexity when the data model and interactions are simple. |

## 5. Model-View-Controller (MVC) Architecture

◆ **Concept**: Separates application logic into three parts:

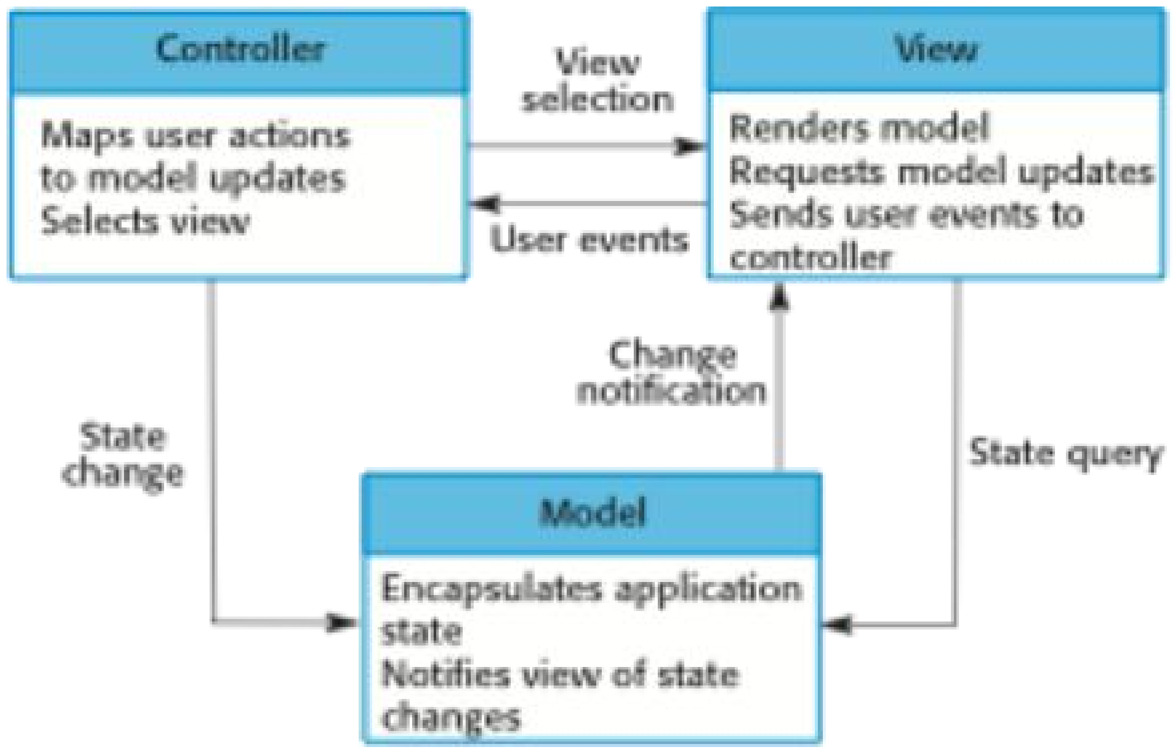
- **Model** – Manages data and logic.
- **View** – Handles UI representation.
- **Controller** – Handles user input and updates the model/view.

◆ **Use Cases:** Web frameworks (Django, Ruby on Rails, Spring MVC).

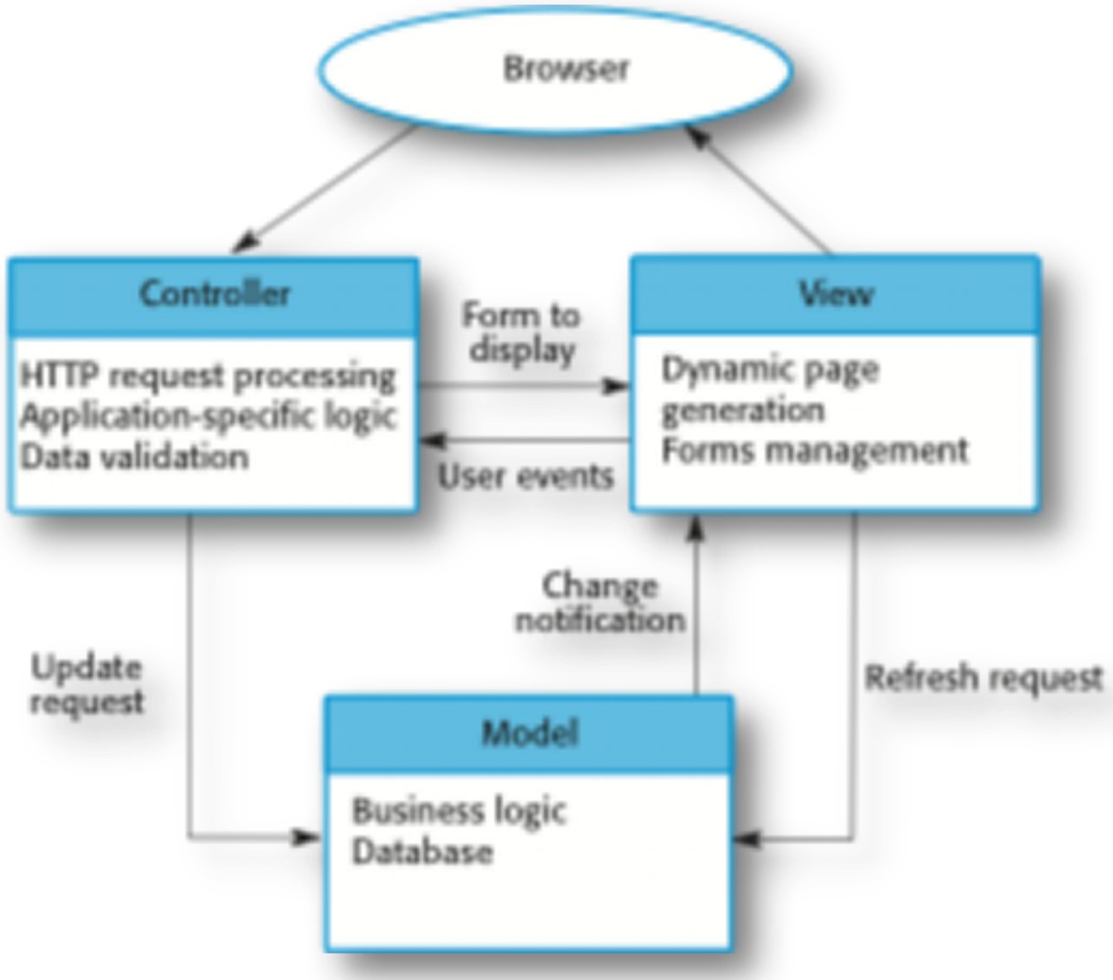
◆ **Pros:** Separation of concerns, easy to manage.

◆ **Cons:** Can become complex with large applications.

## The organization of the Model-View-Controller



# Web application architecture using the MVC pattern



- MVC and Layered Architecture are not direct replacements, but MVC can exist within a Layered Architecture or be adapted into a layered approach.

- **Understanding the Relationship**
- MVC is a design pattern that primarily focuses on separating UI, business logic, and data in user-facing applications.
- Layered Architecture is a broader architectural pattern that enforces strict separation of concerns in a system.
- Thus, instead of replacing MVC with Layered Architecture, you can integrate MVC within a Layered Architecture to get the best of both worlds.

## Task 04

- Repository Architecture
- Event-based Architecture