# Architectural Patterns

An architectural pattern defines a reusable solution to a recurring design problem at the system-level. It describes a context, a problem, and a general solution made up of component types, interaction mechanisms, topologies, and constraints.

---

## Layered Pattern

Divides the system into layers where each layer only communicates with the one directly below it. Every layer exposes a public interface. This pattern supports separation of concerns and enables independent development and testing per layer.

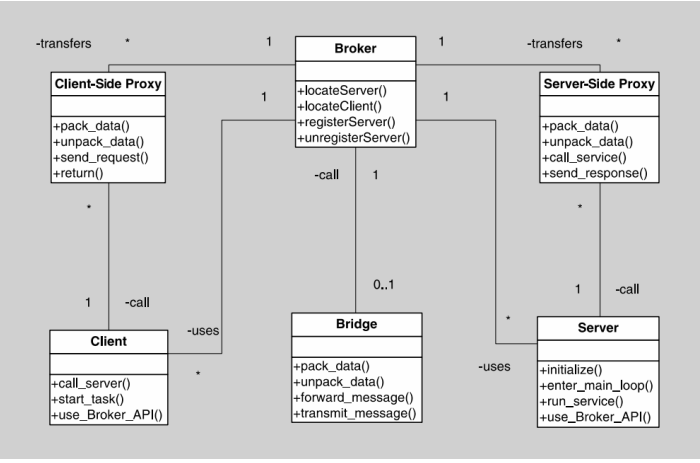
**When to use?** System includes stacked processing stages (e.g., data validation → logic → persistence). It is especially useful when responsibilities can be grouped hierarchically (e.g. UI, logic, data).

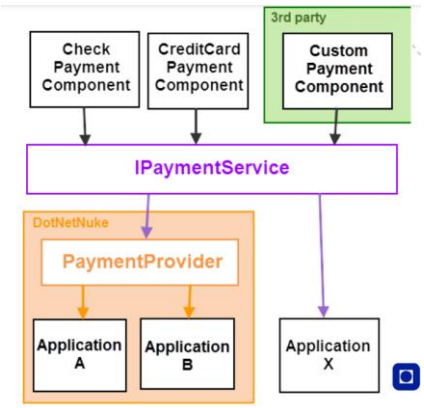
---

## Broker Pattern

Introduces a broker component that mediates between clients and servers, allowing them to interact without knowing about each other directly. Clients send requests to the broker, which forwards them to appropriate services. This supports distribution, scalability, and platform independence.

**When to use?** Need to connect heterogeneous systems or enable remote service discovery and communication without tight coupling.



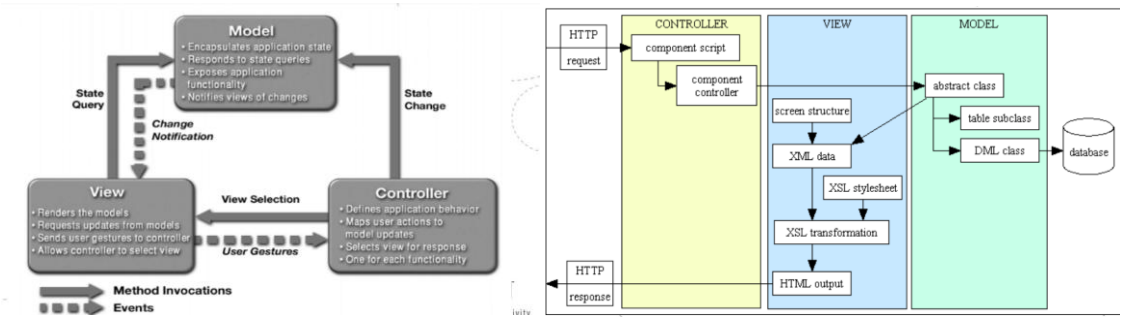




---

## Model-View-Controller (MVC)

Separates the user interface (View), business logic (Model), and interaction handling (Controller). The View observes the Model and updates when the Model changes; the Controller updates the Model based on user input.

**When to use?** System has distinct user interaction and business logic components that evolve independently. Improves testability and UI flexibility, especially for interactive applications with multiple frontends.

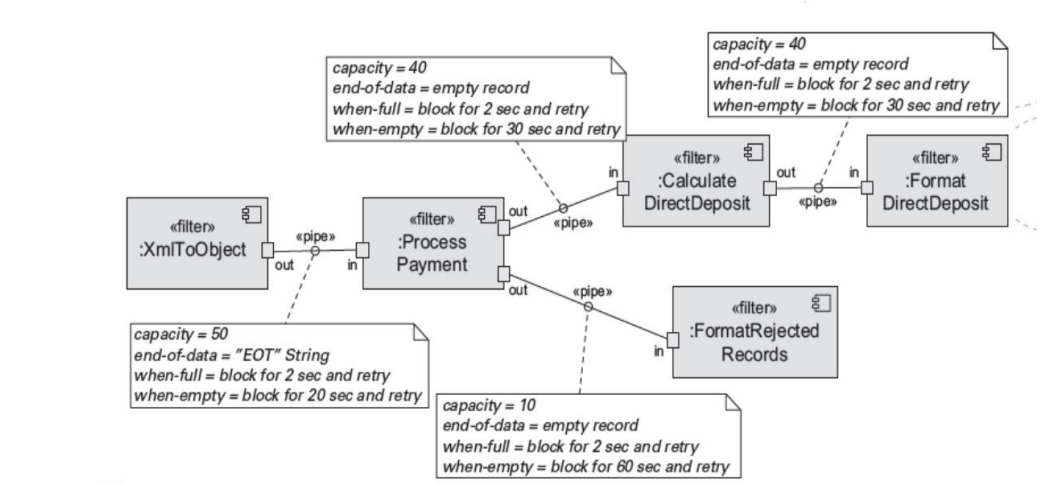



---

## Pipe and Filter

Processes data through a sequence of independent components (filters), each transforming the data and passing it to the next through pipes. Filters do not retain state between executions, and each step can be reused.

**When to use?** System processes data in clear sequential stages. Systems with linear data transformations such as compilers, audio/video processing, or data pipelines.




---

## Client-Server

Divides the system into clients (which request services) and servers (which provide services). Clients and servers can evolve independently. It allows central management of logic or data while enabling multiple user interfaces.

**When to use?** System requires centralized services accessed by distributed users. Network systems, web applications, or services with many users accessing shared resources.

## Peer-to-Peer

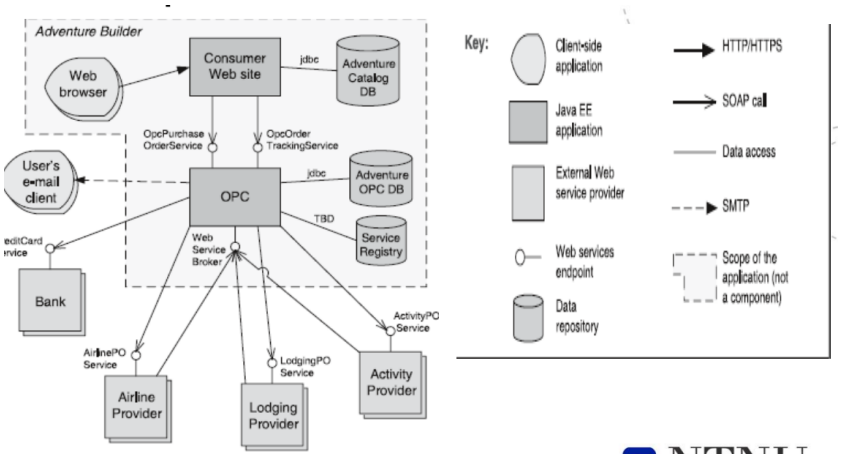
All components are peers — they act both as clients and servers. This avoids single points of failure and promotes scalability and resilience. Communication is symmetric and decentralized.

**When to use?** System needs high resilience, redundancy, and decentralization. Common in file-sharing systems, collaborative tools, and blockchain-based networks.

## Service-Oriented Architecture (SOA)

Organizes the system into distributed, loosely coupled services that communicate over a shared messaging infrastructure (e.g., a service bus). Each service can be developed and deployed independently.

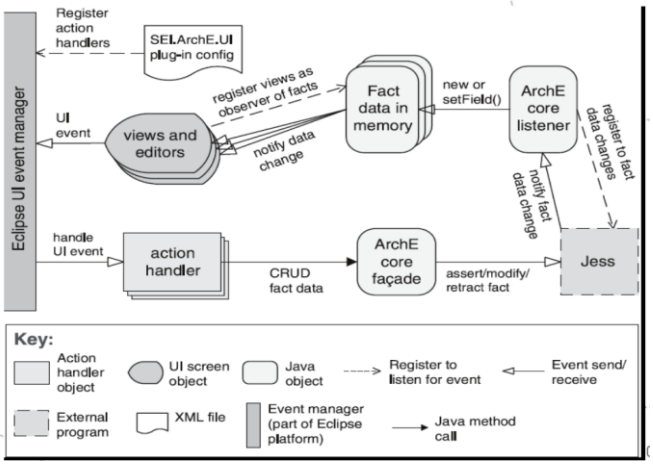
**When to use?** When integrating autonomous systems across different platforms or domains.




## Publish-Subscribe

Components interact via events — publishers broadcast messages without knowing which components (subscribers) will receive them. Subscribers register interest in specific events. Promotes decoupling between producers and consumers.

**When to use?** Want to separate event producers from consumers and avoid direct dependencies. Useful for notification systems, sensor networks, or applications with loosely coupled modules reacting to changes.

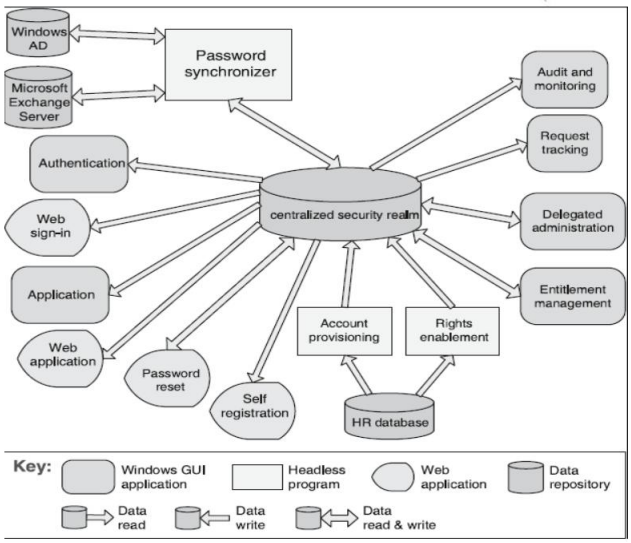



---

## Shared Data

Components share access to a central data store, which acts as the communication medium. Each component reads and writes data independently. This pattern simplifies coordination when data is the main integration point.

**When to use?** Only when concurrent access is well controlled or isolation is not a concern. Found in traditional database-centric applications, but considered risky in highly concurrent or modular systems.

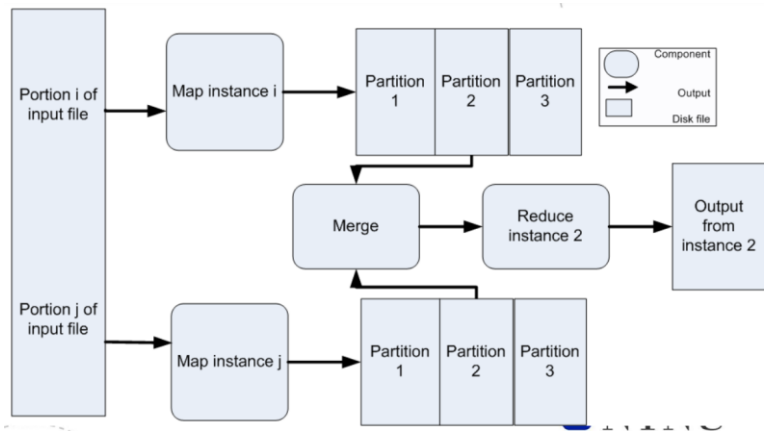



---

## Map-Reduce

Splits large-scale data processing into two main operations: Map (filter and transform input data) and Reduce (aggregate results). A coordinating infrastructure distributes the computation across multiple nodes.

**When to use?** When your problem involves processing huge datasets in parallel. Highly effective for batch processing in large-scale data analytics, machine learning pipelines, or indexing.




---

## Multi-Tier (n-tier)

Divides the system into logical tiers such as Presentation, Application Logic, and Data. Each tier is responsible for a specific set of concerns and may be deployed independently. Improves scalability and maintainability by isolating responsibilities.

**When to use?** When organizing business logic, UI, and data into distinct layers improves system structure. Standard in enterprise and web architectures.