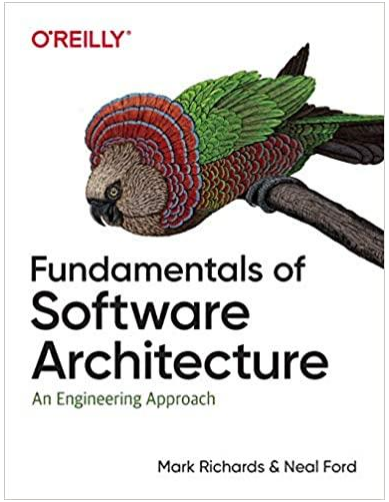
Part II. Architecture Styles



List of Architecture Styles Introduced:

Monolithic

- Layered Architecture Style
- Pipeline Architecture Style
- Microkernel Architecture Style

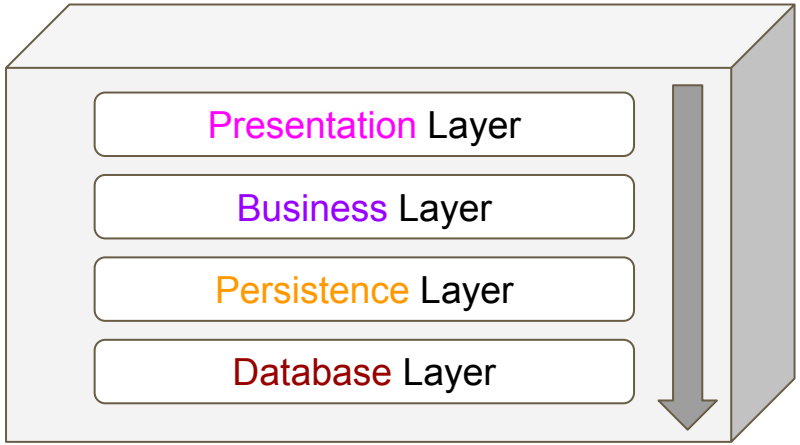
Distributed

- Service-Based Architecture Style
- Event-Driven Architecture Style
- Space-Based Architecture Style
- Orchestration-Driven Service-Oriented Architecture
- Microservices Architecture

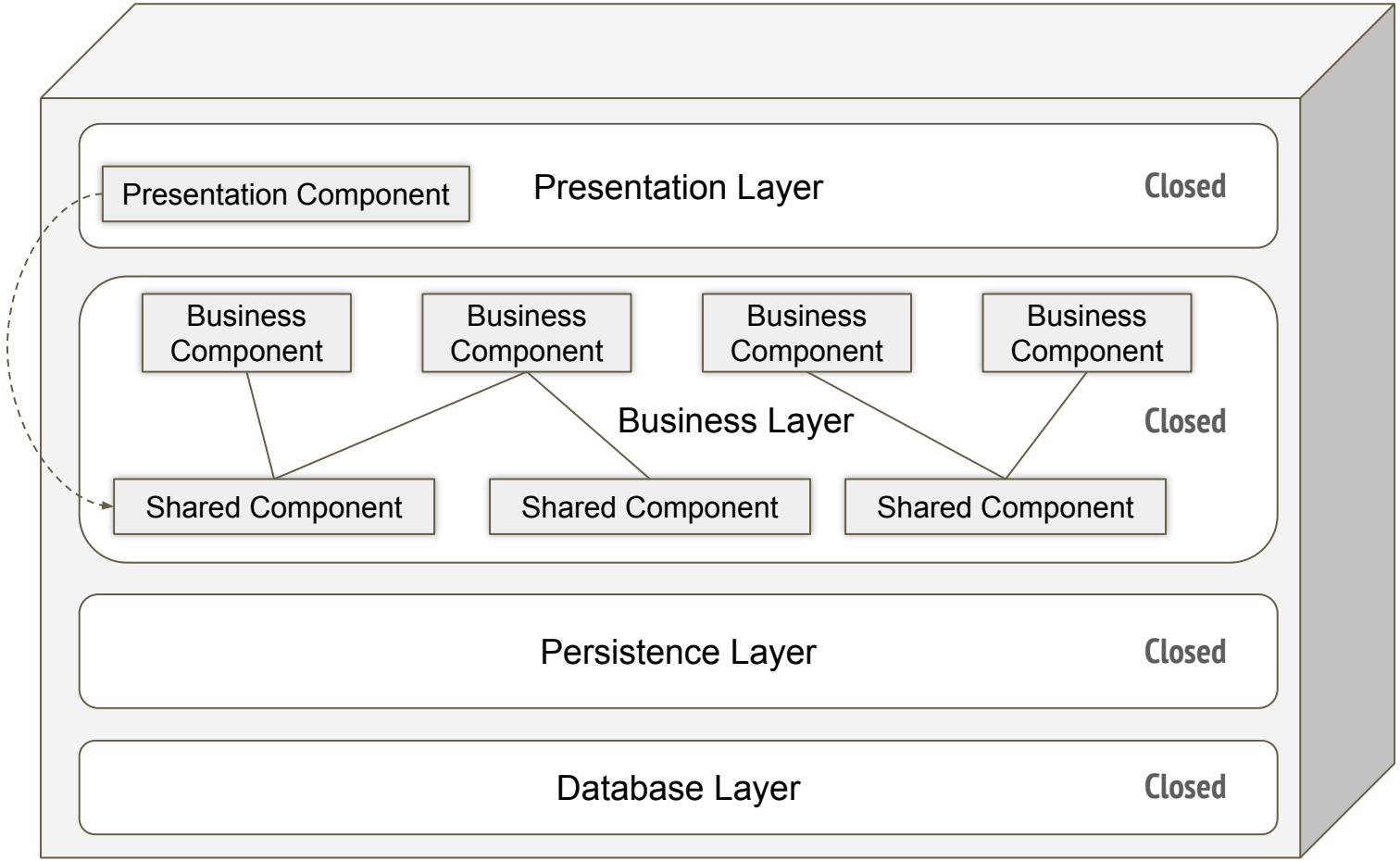
Layered Architecture Style

Logical horizontal layers, each performing a specific role within the application.

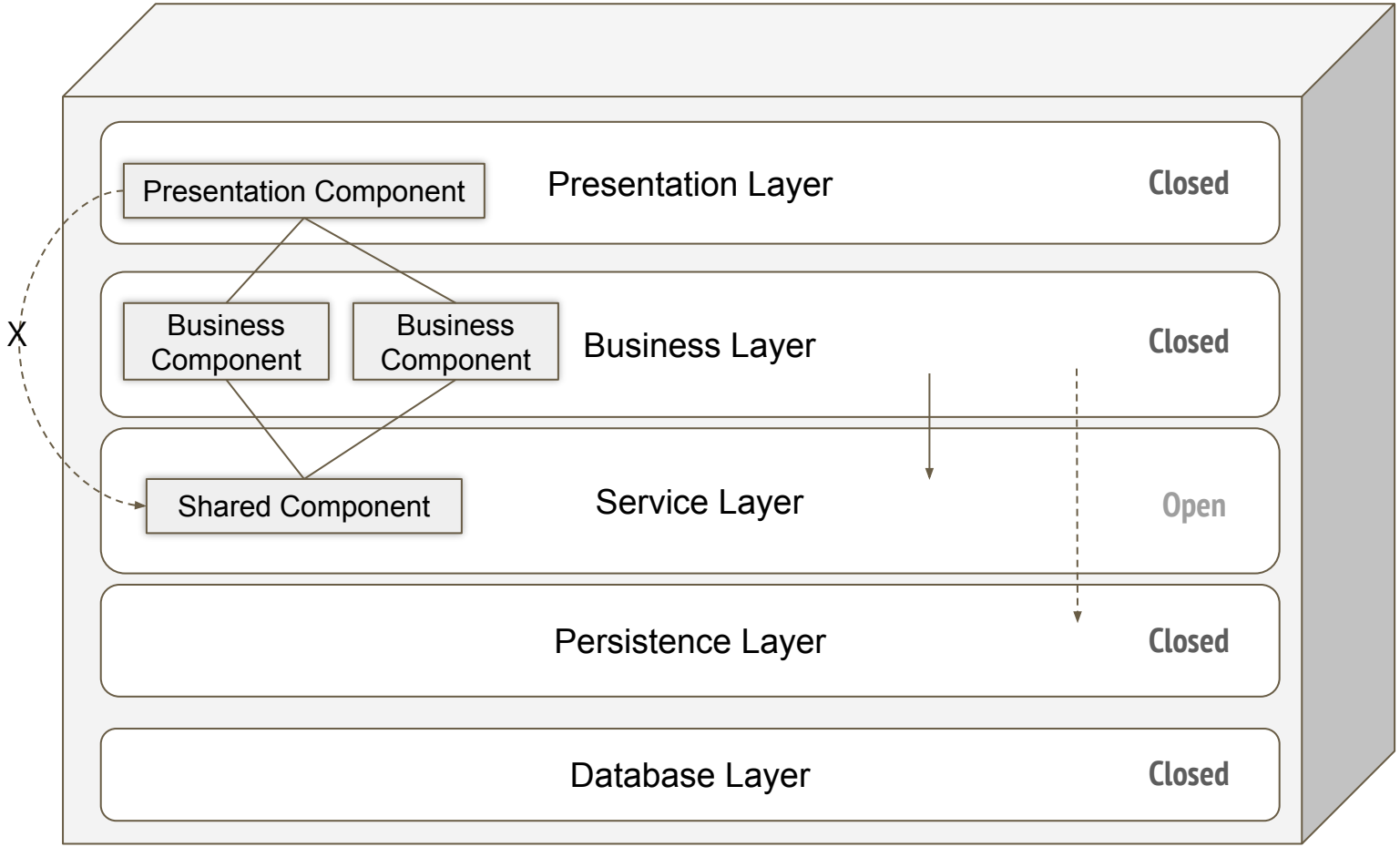
- The presentation layer contains the graphical design of the application, as well as any code to handle user interaction
- The business layer is where you put the models and logic that is specific to the business problem you are trying to solve.
- The persistence layer contains the code to access the database layer.
- The database layer is the underlying database technology (e.g. SQL Server, MongoDB).



Shared objects within the business layer



Adding a new service layer to the layer architecture



Pipeline Architecture Style

Pipes

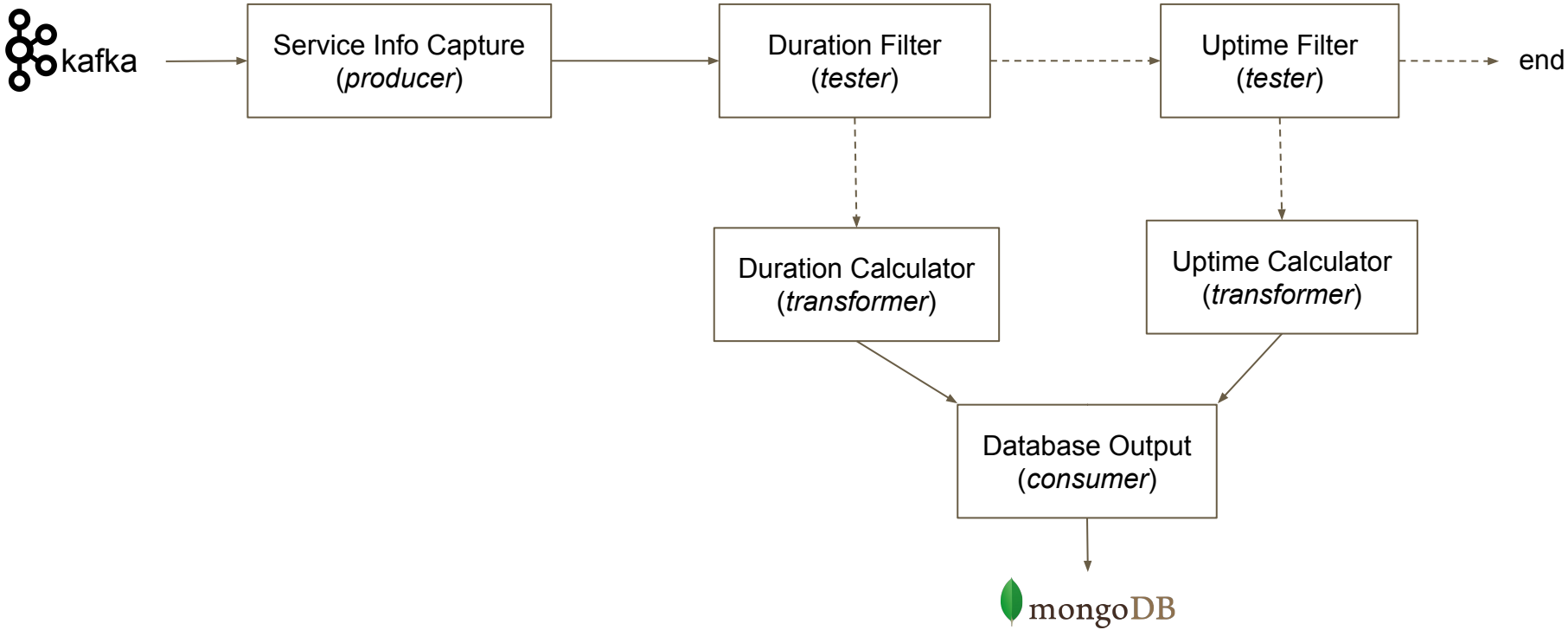
Pipes form the communication channel between filters.

Filters

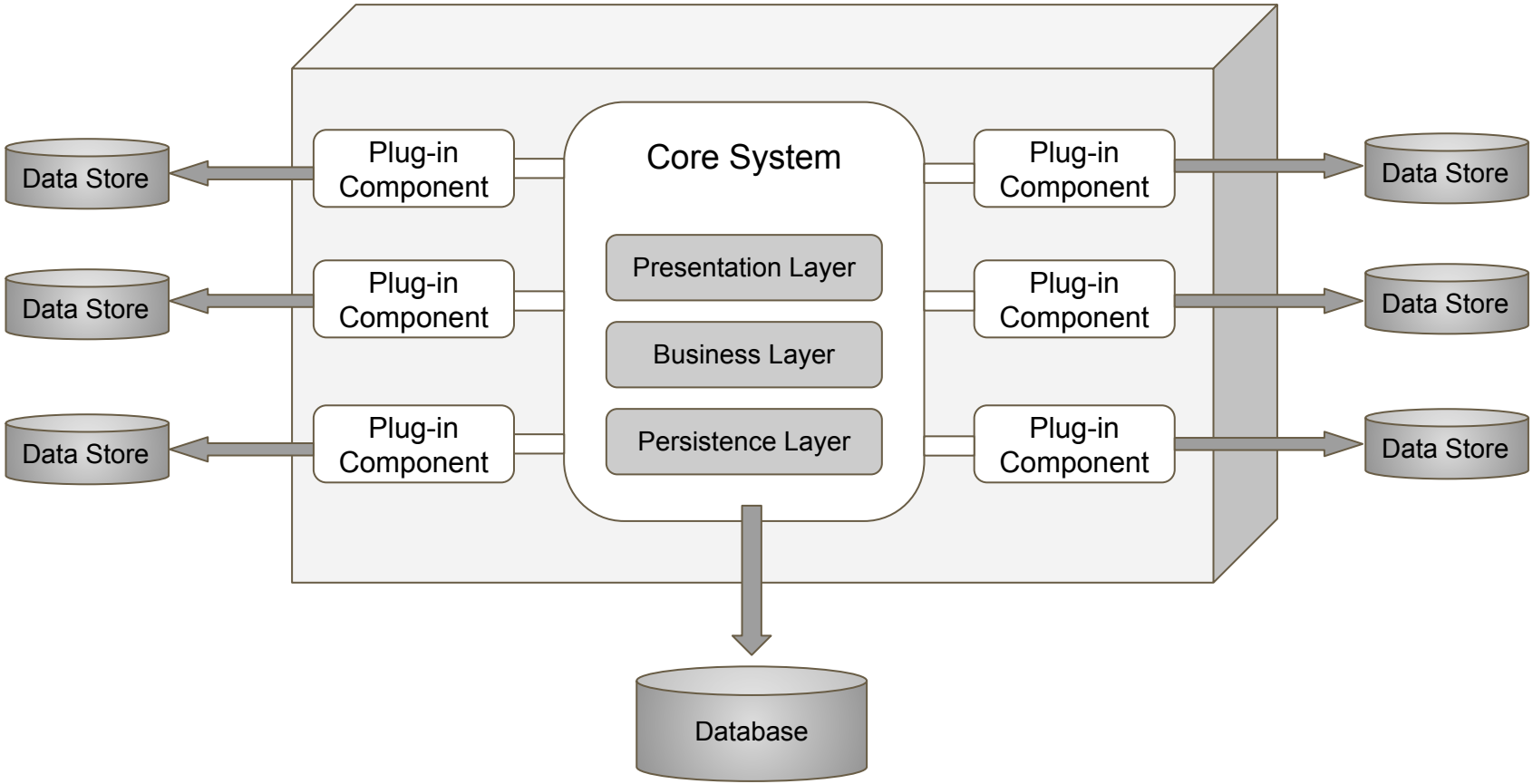
- *Producer*
  - the starting point of a process, also called source.
- *Transformer*
  - Accepts input, optionally performs a transformation on data, then forwards it to the outbound pip. This feature is also called *map*.
- *Tester*
  - Accepts input, tests one or more criteria, then optionally produces output, based on the test. This feature is also called *reduce*.
- Consumer
  - The termination point for the pipeline flow. It could persist the final result to a database, or may display the final result on a UI.

Examples:

- Electronic Data Interchange (EDI)
- ETL tools (extract, transform, and load)
- Apache Camel: pass information from one step in a business process to another.



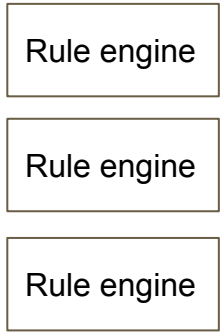
Microkernal (*plug-in*) Architecture Style



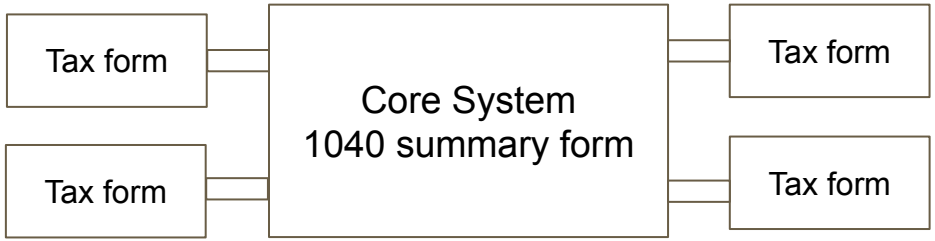
Examples:

- Eclipse IDE (https://www.eclipse.org/ide)
- PMD (https://pmd.github.io)
- Jira (https://www.atlassian.com/software/jira)
- Jenkins (https://jenkins.io)
- Chrome and Firefox

Auto insurance claims applications



Tax preparation software



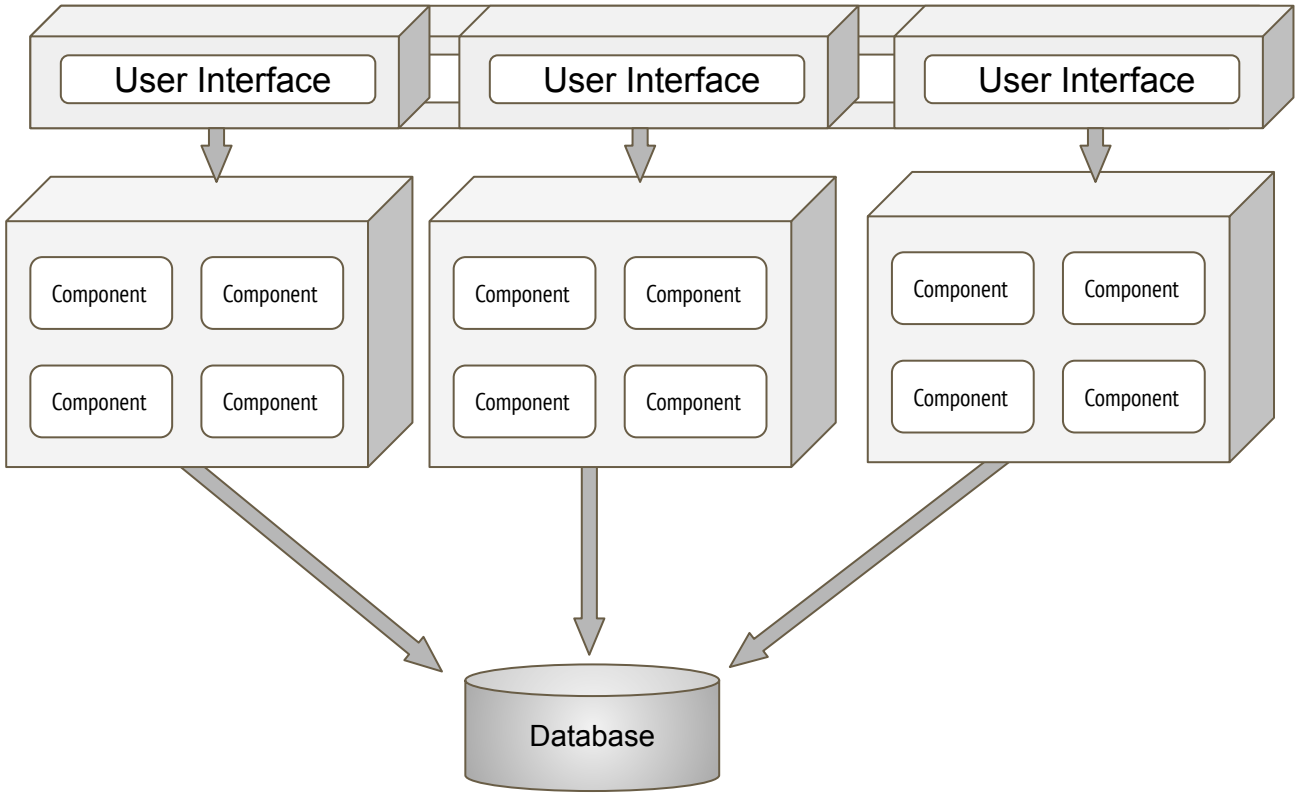
Some architecture characteristics

Availability	How long the system will need to be available
Performance	Includes stress testing, peak analysis, capacity required, and response times.
Reliability	Assess if the system needs to be fail-safe. If it fails, will it cost the company large sums of money?
Scalability	Ability to increase workload size within existing infrastructure (hardware, software, etc.) without impacting performance.
Fault tolerance	Ability to handle error and boundary conditions while running if the internet connections goes down or if there's a power outage or hardware failure.
Elasticity	Ability to grow or shrink infrastructure resources dynamically as needed to adapt to workload changes in an autonomic manner, maximizing the use of resources.
Recoverability	Business continuity requirements(e.g. In case of disaster, how quickly is the system required to be online again?). This will affect the backup strategy and requirements for duplicated hardware.
Security	Does the data need to be encrypted in the database? Encrypted for network communication between internal systems? What type of authentication needs to be in place for remote user access?

Architecture characteristics	Layer Architecture	Pipe Architecture	Kernel Architecture
Partitioning type	Technical	Technical	Domain and technical
Deployability	★	★★	★★★
Elasticity	★	★	★
Evolutionary	★	★★★	★★★
Fault tolerance	★	★	★
Modularity	★	★★★	★★★
Overall cost	★★★★★	★★★★★	★★★★★
Performance	★★	★★	★★★
Reliability	★★★	★★★	★★★
Scalability	★	★	★
Simplicity	★★★★★	★★★★★	★★★★★
Testability	★★	★★★	★★★

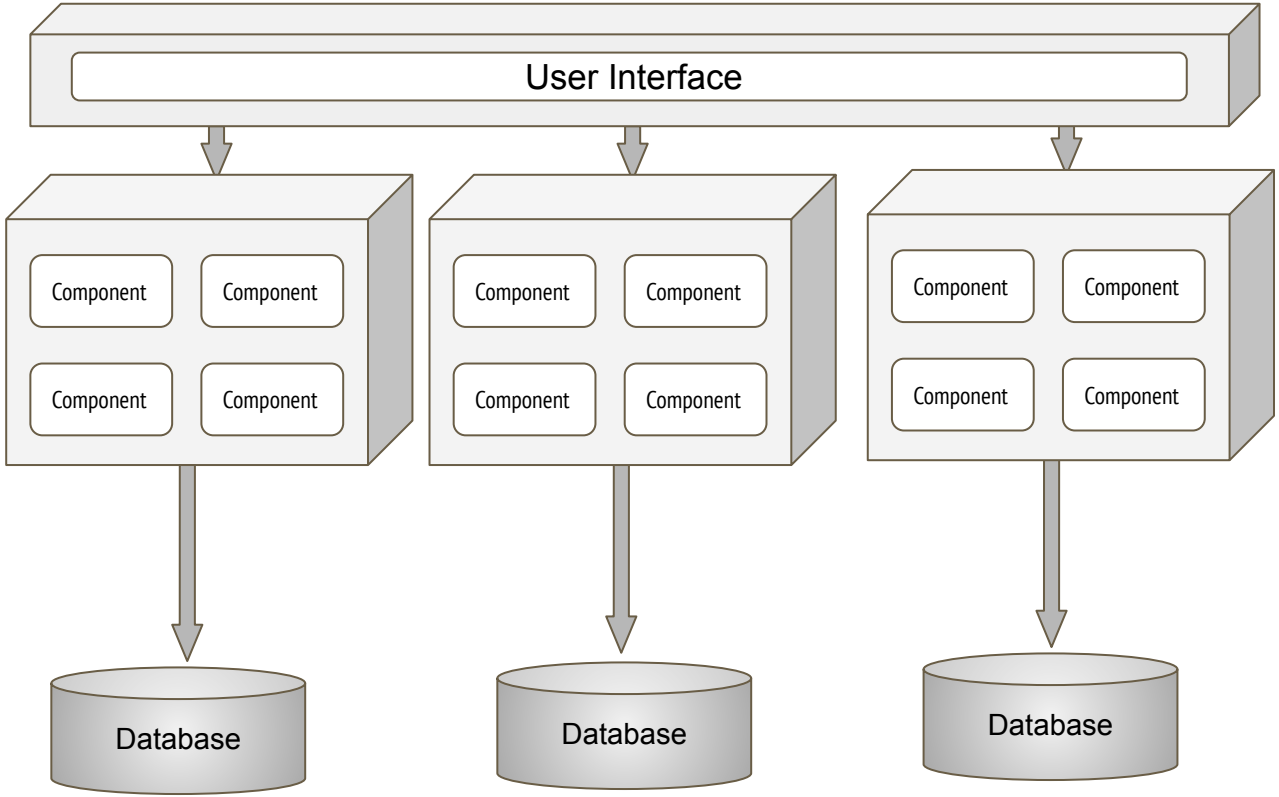
Service-Based Architecture Style

Basic topology



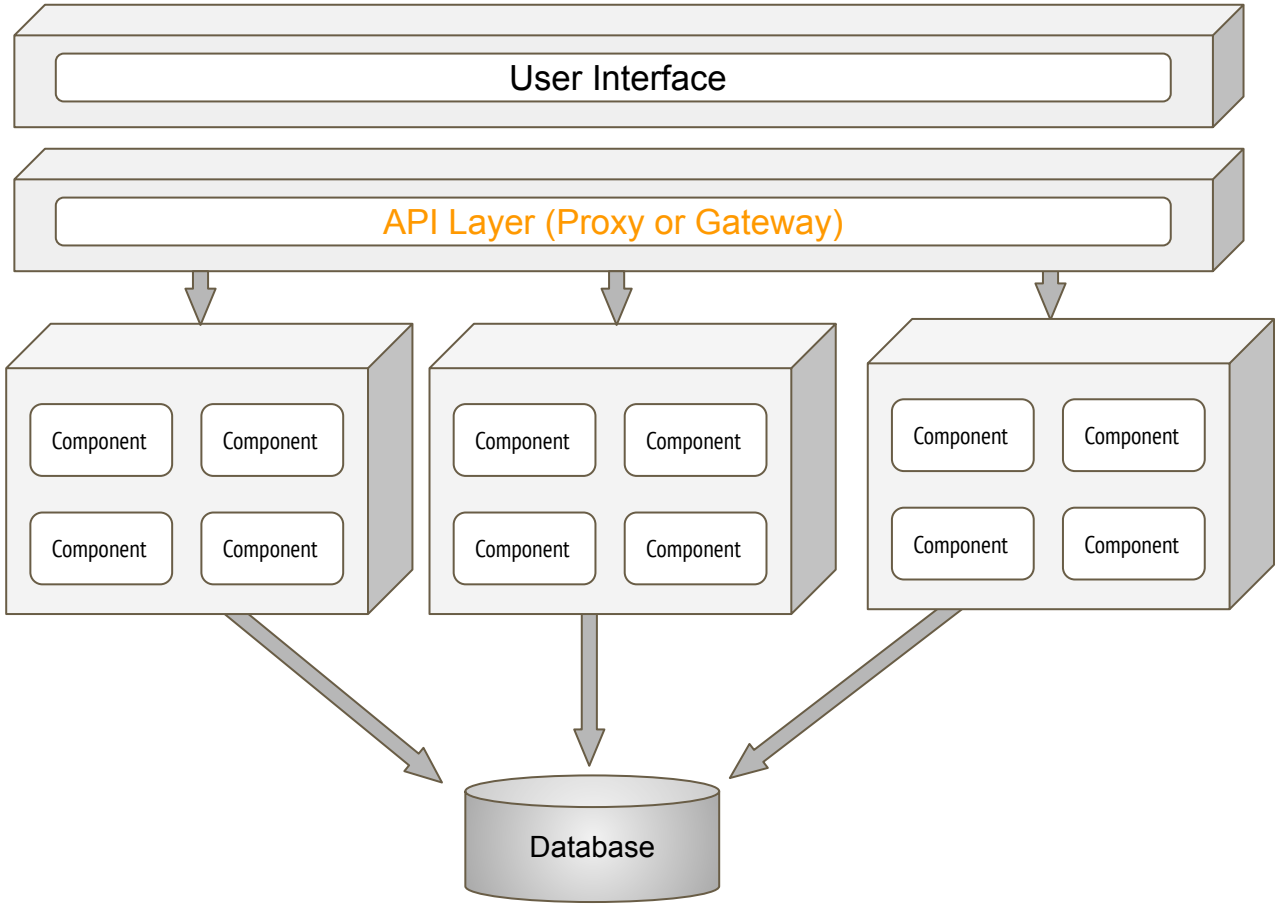
Service-Based Architecture Style

Database Variants

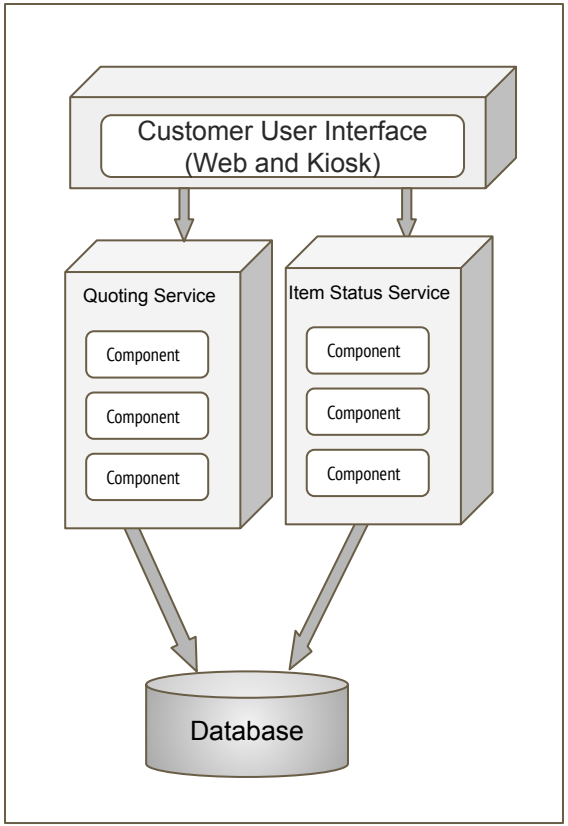


Service-Based Architecture Style

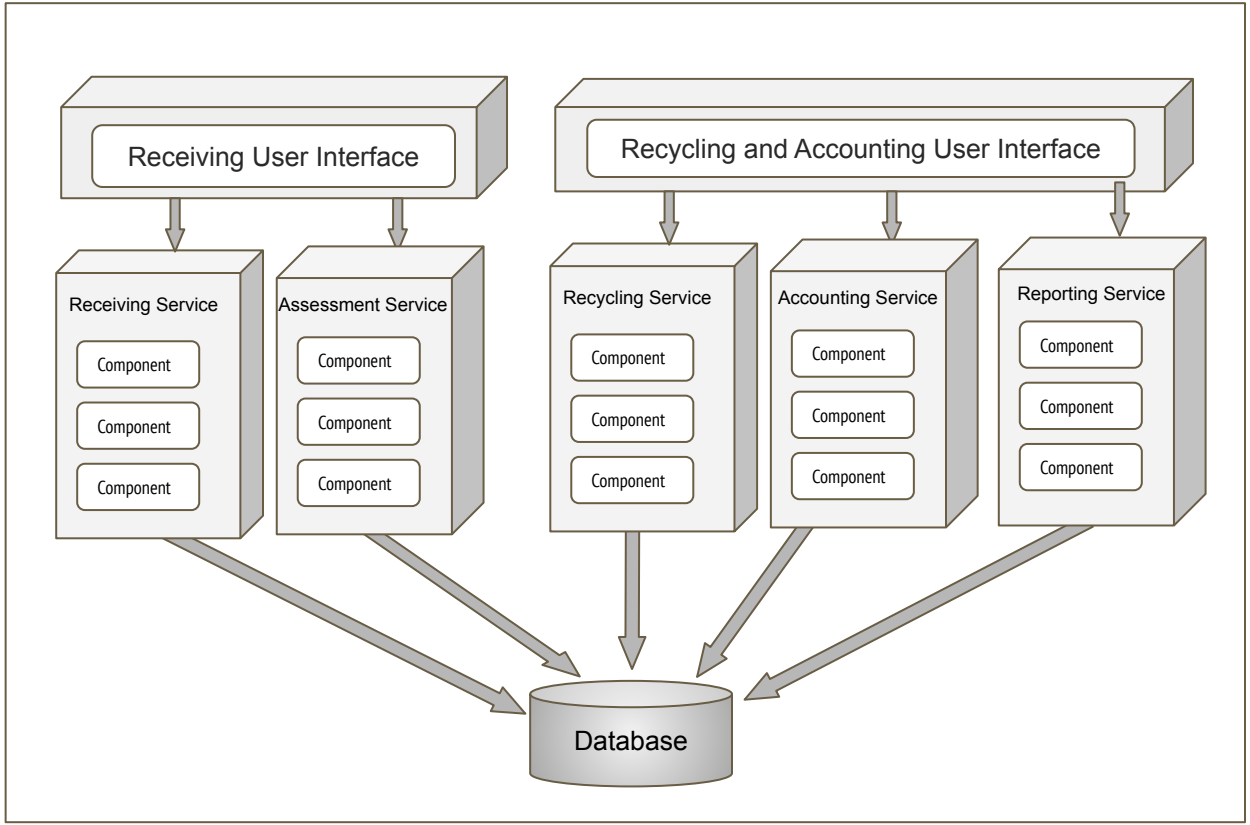
Adding an API Layer



An electronic recycling system to recycle old electronic devices



External customer-facing quantum



Internal operations quantum

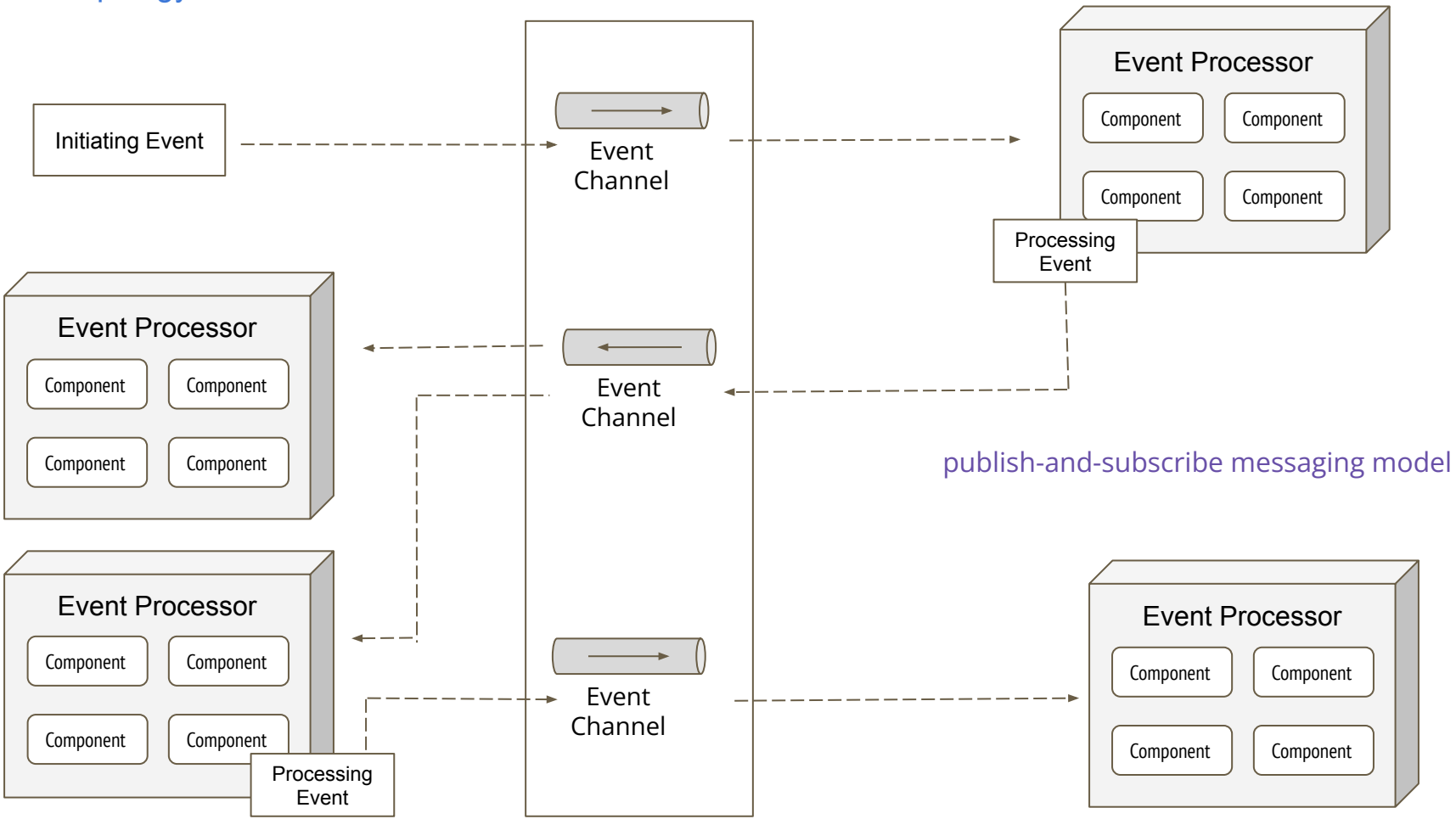
Event-Driven Architecture Style

Event-based Model

	When to use?
Broker Topology	Require a high degree of responsiveness and dynamic control over the processing of an event.
Mediator Topology	Require control over the workflow of an event process.

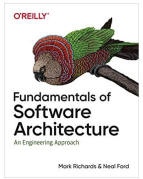
Author comments:
The choice between the broker and mediator topology comes down to a trade-off between workflow control and error handling capability versus high performance and scalability.

Broker Topology



publish-and-subscribe messaging model

Broker Topology Example: An retail order entry system



place order

Order Placement
Component
Component
Component
Component

Notification
Component
Component
Component
Component

order created

payment denied

Warehouse
Component
Component
Component
Component

Inventory updated

Inventory
Component
Component
Component
Component

Payment
Component
Component
Component
Component

Order Fulfillment
Component
Component
Component
Component

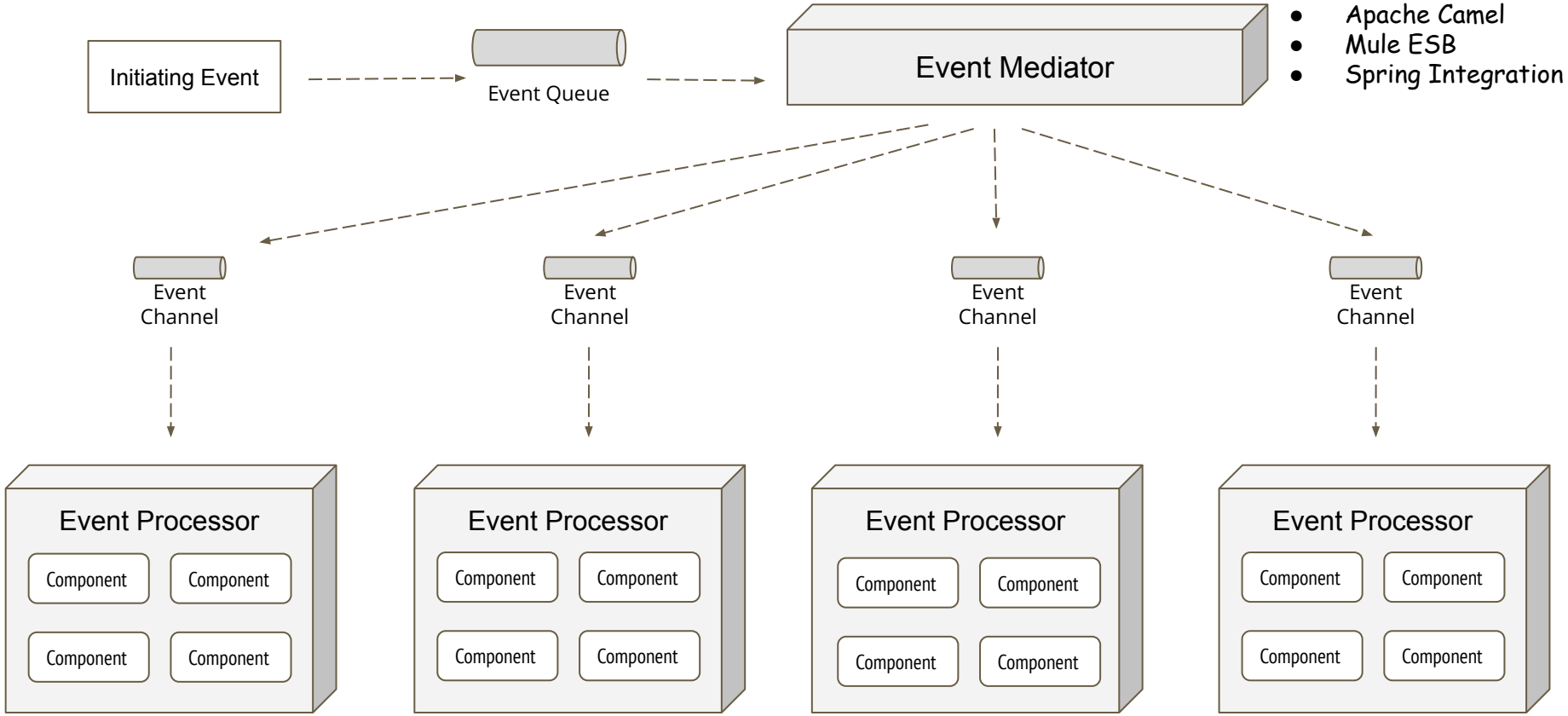
order-fulfilled

Shipping
Component
Component
Component
Component

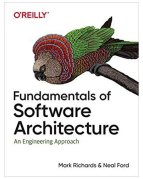
payment applied

order-shipped

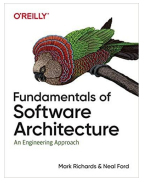
Mediator Topology



Mediator Topology Example: An retail order entry system



Mediator Topology Example: An retail order entry system



place order

Event Mediator

STEP1 | STEP2 | STEP3 | STEP4 | STEP5

Create Order | Email Customer | Apply Payment | Adjust Inventory | Fulfill Order | Replenish Stock | Email Customer | Ship Order | Email Customer

email customer

email customer

ship order

Order Placement
- Component
- Component

Notification
- Component
- Component

Payment
- Component
- Component

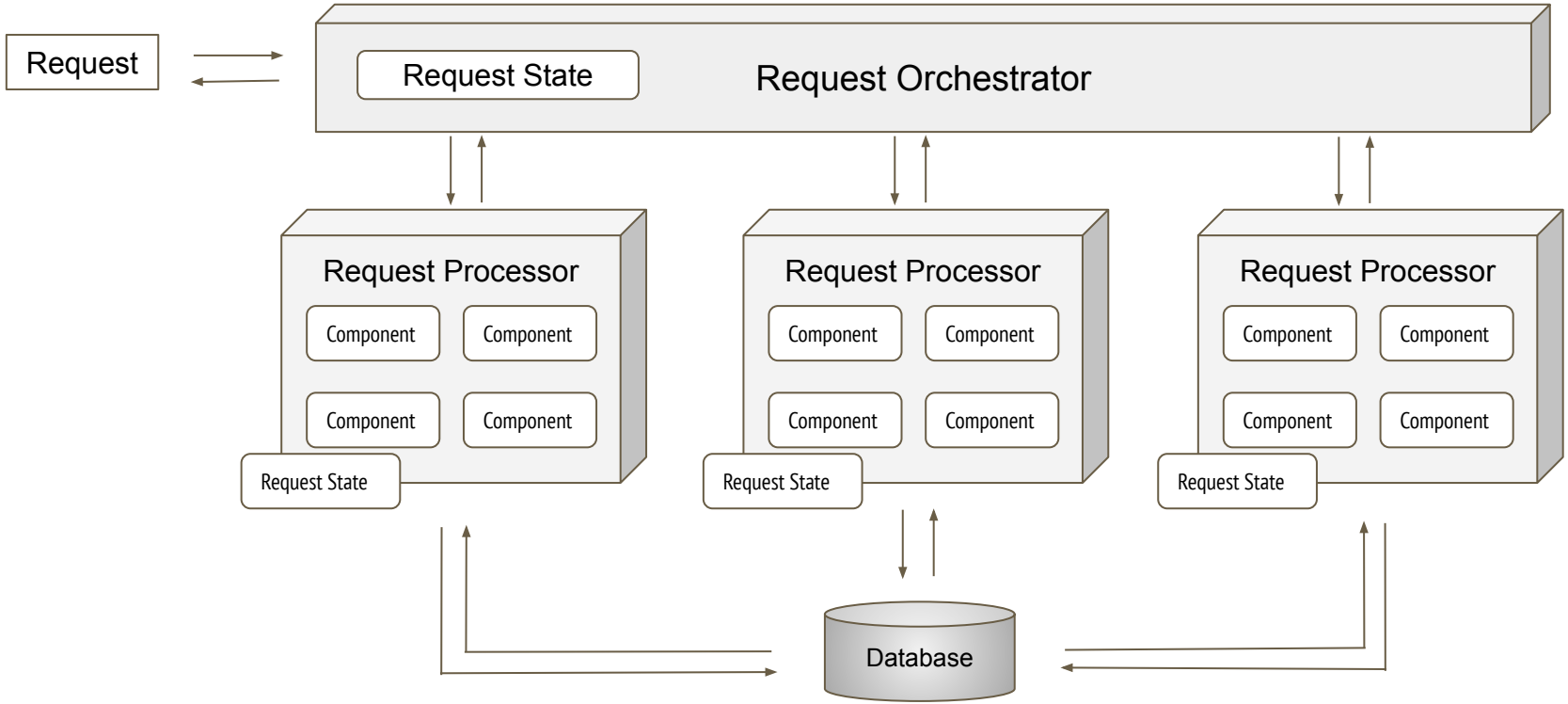
Inventory
- Component
- Component

Warehouse
- Component
- Component

Order Fulfillment
- Component
- Component

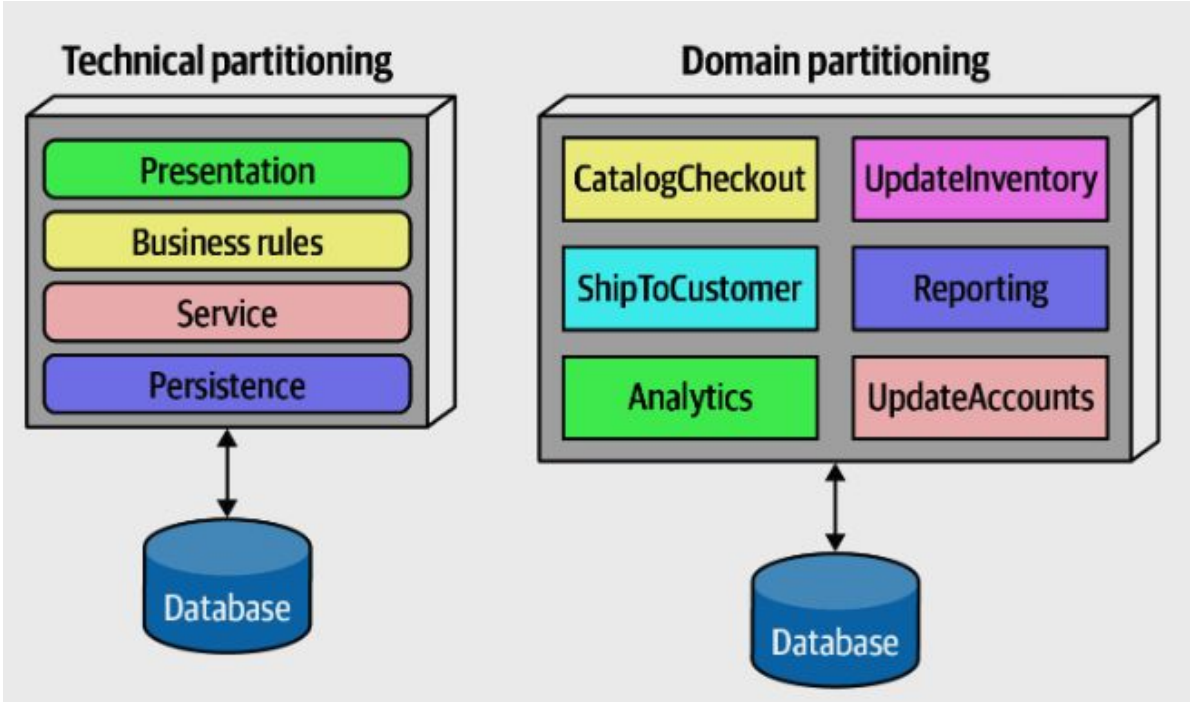
Shipping
- Component
- Component

Request-based Model can be seen in most applications



Architecture characteristics	Service-Based Architecture	Event-Driven Architecture
Partitioning type	Domain	Technical
Deployability	★★★★	★★★
Elasticity	★★	★★★
Evolutionary	★★★	★★★★★
Fault tolerance	★★★★	★★★★★
Modularity	★★★★	★★★★
Overall cost	★★★★	★★★
Performance	★★★	★★★★★
Reliability	★★★★	★★★
Scalability	★★★	★★★★★
Simplicity	★★★	★
Testability	★★★★	★★

Technical vs Domain partitioning



Architectural Quantum

Definition of Quantum in *Physics*: the minimum amount of any physical entity involved in an interaction.

Architecture Quantum: An independently deployable artifact with high functional cohesion and synchronous connascence.

- Independently deployable

All the necessary components function independently from other parts of the architecture.

- High functional cohesion

Implies that an architecture quantum does something purposeful.

- Synchronous connascence

Implies synchronous calls within an application context or between distributed services that form this this architecture quantum.

Space-Based Architecture Style

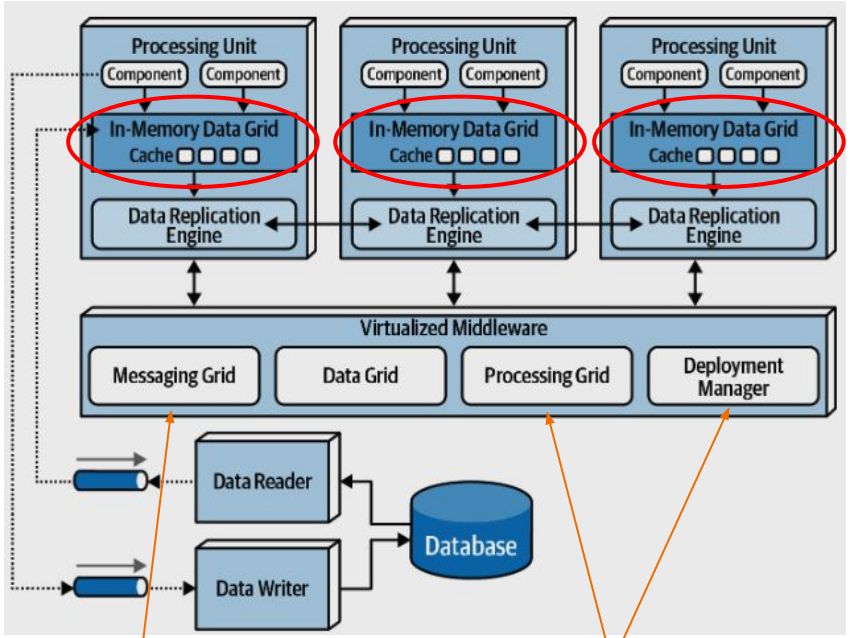
Space-based architecture gets its name from the concept of *tuple space*, the technique of using multiple parallel processors communicating through shared memory.

Removal central DB, but leveraging replicated in-memory data grids.

Processing Unit: application logic

Virtualized Middleware: control various aspects of data synchronization and request handling.

Data Pumps: asynchronous, providing eventual consistency with the in-memory cache and the database.



Manage orchestrated request processing

Manage input request and session state.

Dynamic startup and shutdown of processing unit instances based on load conditions.

Examples:

Spaced-based architecture is well suited for applications that experience high spikes in user or request volume and application that have throughput in excess of 10k concurrent users.

- ### Concert Ticketing System

  Limited tickets available
  Difficult for a typical database to handle high volume concurrent requests.

- ### Online Auction System

  Require high levels of performance and elasticity.
  Unpredictable spikes in user and request load.

Orchestration-Driven Service-Oriented Architecture

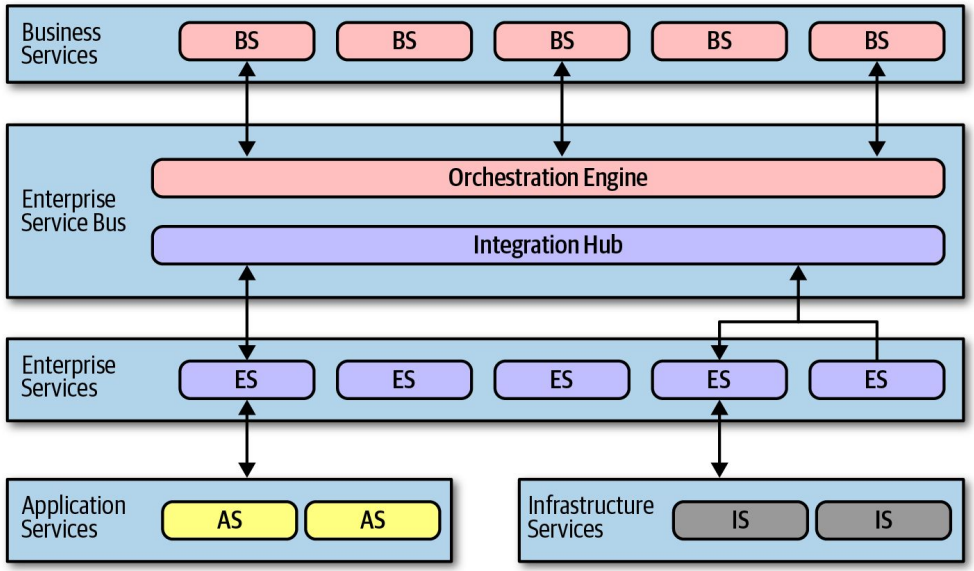
Late 1990s

Business Services: no code, just input, output and schema information.

Orchestration Engine: stitching together the business service using orchestration, like transactional coordination and message transformation.

Enterprise Services: fine-grained, shared implementations. Building blocks and tied together via the orchestration engine.

Application Services: one-off, single-implementation services. Not reusable.

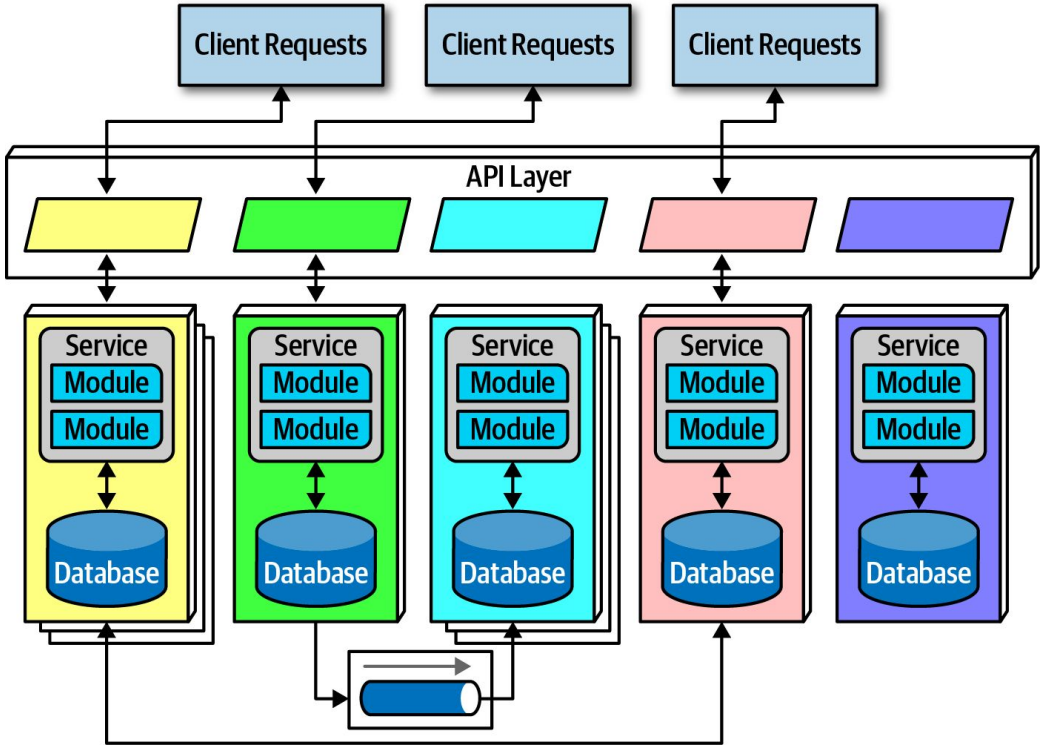


Infrastructure Services: monitoring, logging, authentication, and authorization.

A major goal of this architecture is reuse at the service level.

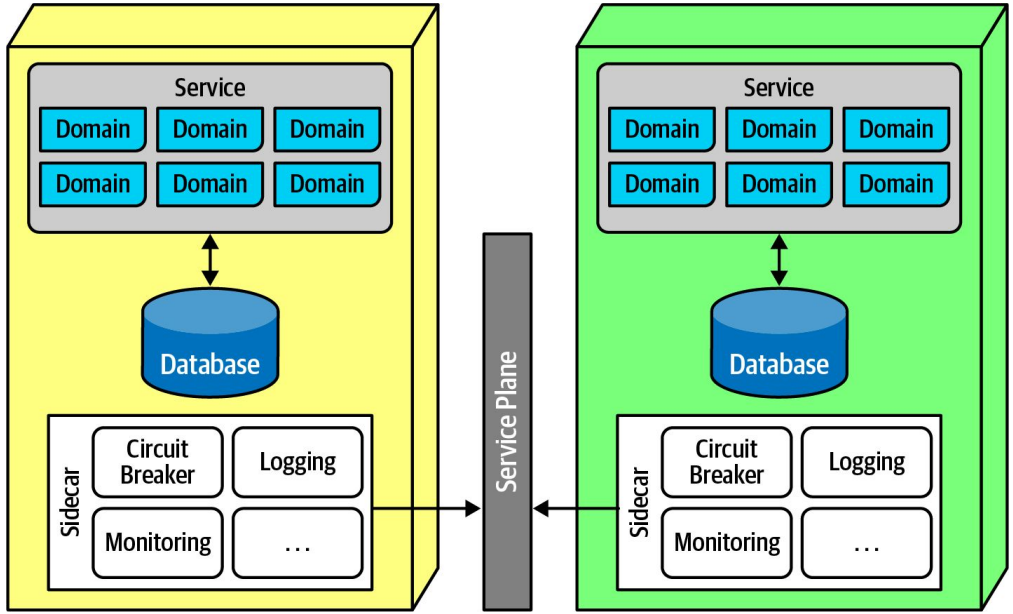
Microservices Architecture

- Decoupling services, both at domain and operational level.
- Performance down side: network calls, security verification at every endpoint.
- Hard to maintain transaction across service boundaries.
- Granularity of service is hard to achieve.
- Data isolation, data consistency issue.



Operational Reuse in Microservice Architecture

- The sidecar component handles all the operational concerns that teams benefit from coupling together.
- When it comes time to upgrade the monitoring tool, the shared infrastructure team can update the sidecar, and each microservices receives that new functionality.



Transaction and Sagas

The Saga design pattern is a way to manage data consistency across microservices in distributed transaction scenarios.

The Saga pattern provides transaction management using a sequence of *local transactions*.

Use the Saga pattern when you need to:

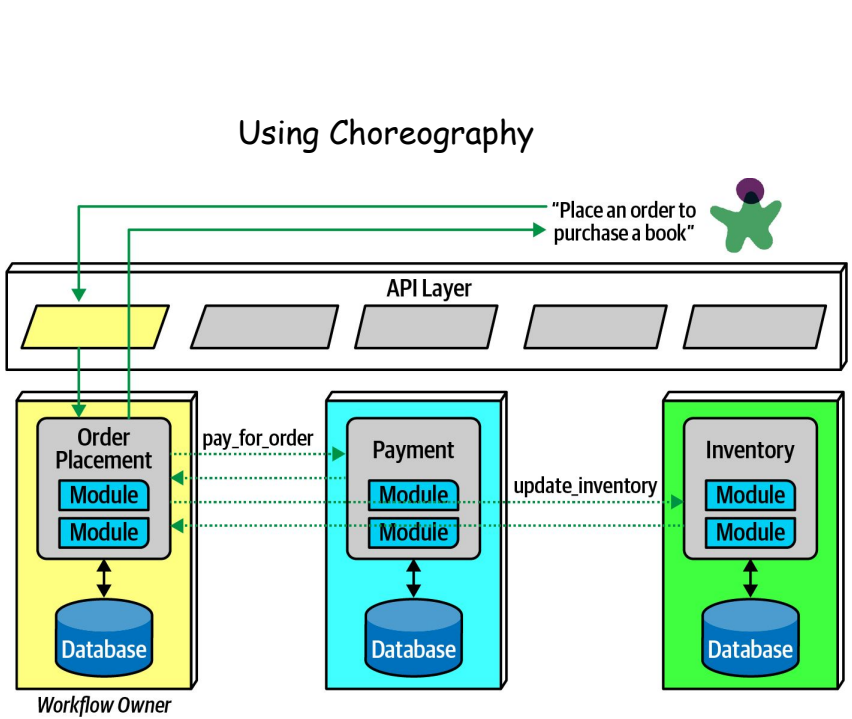
- Ensure data consistency in a distributed system without tight coupling.
- Roll back or compensate if one of the operations in the sequence fails.

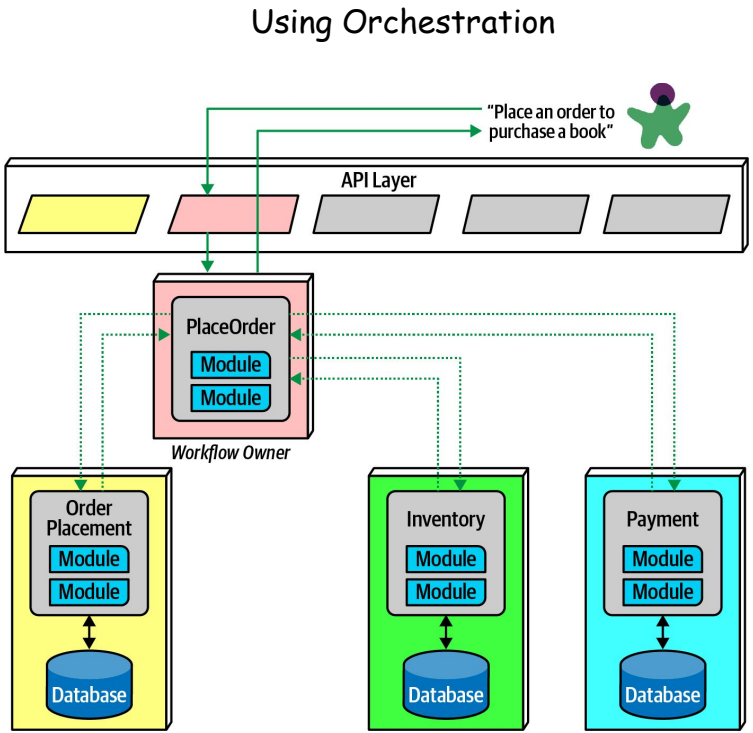
There are two common saga implementation approaches, *choreography* and *orchestration*. Each approach has its own set of challenges and technologies to coordinate the workflow.

Choreography and Orchestration

Choreography: no central coordinator exist, utilize the same communication style as a broker event-driven architecture.

Orchestration: coordinate across several services,

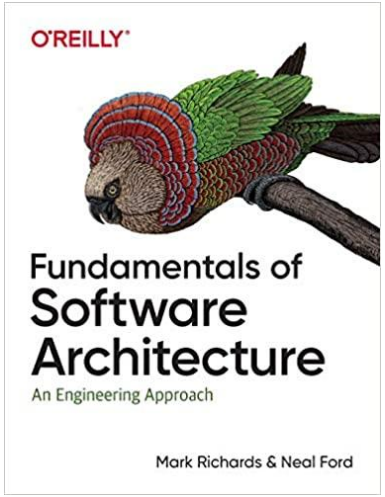




Architecture characteristics	Space-Based Architecture	Orchestration-Driven Service-Oriented Architecture	Microservice Architecture
Partitioning type	Domain and technical	Technical	Domain
Deployability	★★★	★	★★★★
Elasticity	★★★★★	★★★	★★★★★
Evolutionary	★★★	★	★★★★★
Fault tolerance	★★★	★★★	★★★★
Modularity	★★★	★★★	★★★★★
Overall cost	★★	★	★
Performance	★★★★★	★★	★★
Reliability	★★★★	★★	★★★★
Scalability	★★★★★	★★★★	★★★★★
Simplicity	★	★	★
Testability	★	★	★★★★

Book:

https://www.amazon.com/Fundamentals-Software-Architecture-Comprehensive-Characteristics/dp/1492043451



Reading notes from others:

- https://github.com/zhangjunhd/reading-notes/blob/master/software/FundamentalsOfSoftwareArchitecture.md#4architecture-characteristics-defined
- https://danlebrero.com/2021/11/17/fundamentals-of-software-architecture-summary/
- https://medium.com/interviewnoodle/the-notes-for-fundamentals-of-software-architecture-e8ab68e85c4a