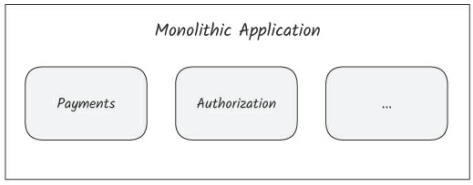
Introduction to Architectural Styles

Ahmad Hamo
2nd Semester 2024-2025

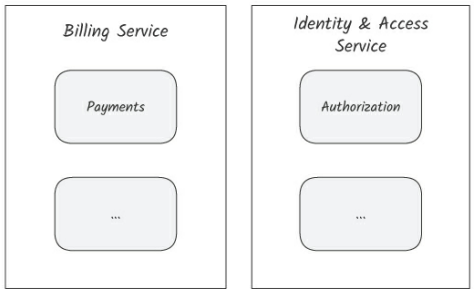
Coupling

A.



(A) Collocating modules in the same encapsulation boundary increases their lifecycle coupling,

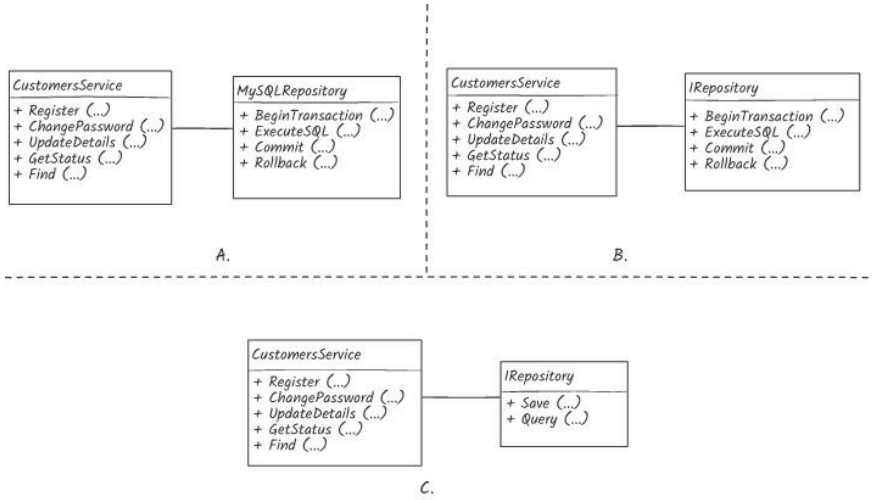
B.



(B) Extracting the modules into different services reduces their lifecycle coupling.

2

Shared Knowledge



A) shares the most knowledge: the **concrete database** that is being used, which is MySQL.

B) reduces the knowledge to the **database family.**

C) encapsulates further and exposes only the minimal knowledge needed for the *CustomersService* module to implement its functionality.

3

Monolithic Versus Distributed Architectures

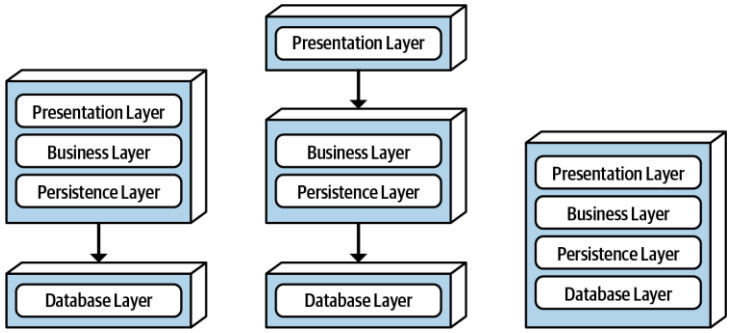
- **monolithic** *:* single deployment unit of all code
  - Layered architecture
  - Pipeline architecture
  - Microkernel architecture

- **distributed** : multiple deployment units connected through remote access protocols:
  - Service-based architecture
  - Event-driven architecture
  - Space-based architecture
  - Orchestration-Driven Service-Oriented Architecture
  - Microservices architecture

distributed architectures all share a common set of challenges and issues not found in the monolithic architecture styles

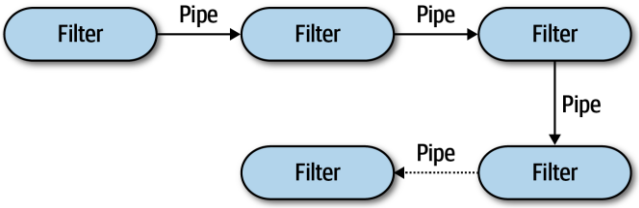
1. Layered Architecture Style

•Organizes software into layers to improve separation of concerns
•**When to Use** : Best for monolithic applications where clear separation of responsibilities is needed.
•**Example** : A web application with a frontend (UI), middleware (business logic), and a backend (database).



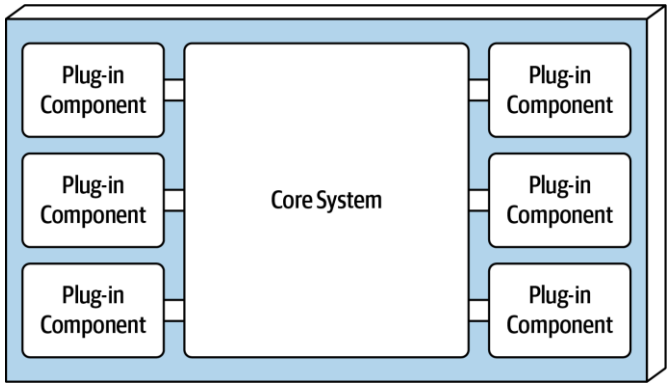
2. Pipeline Architecture Style

•Processes data in stages (pipeline), where each stage transforms the input and passes it to the next stage.
•**When to Use:** Suitable for stream processing or data-intensive applications.
•**Example2:** A video processing application where frames go through decoding, filtering, and encoding steps.
•**Example2**: A data processing pipeline that reads raw data, cleans it, transforms it, and then stores it in a database.



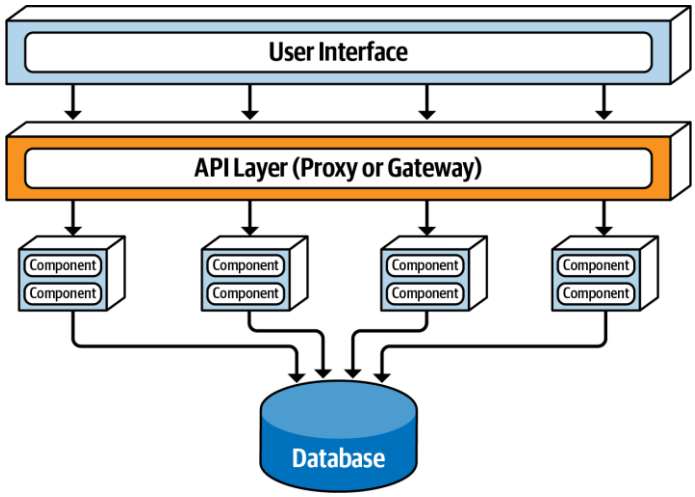
3. Microkernel architecture

•Consists of a minimal core system with plug-in modules for extending functionality.
•**When to Use**: Useful for applications that require a stable core with flexible extensions.
•**Example**: Eclipse IDE, where the core platform remains the same while developers add plugins.

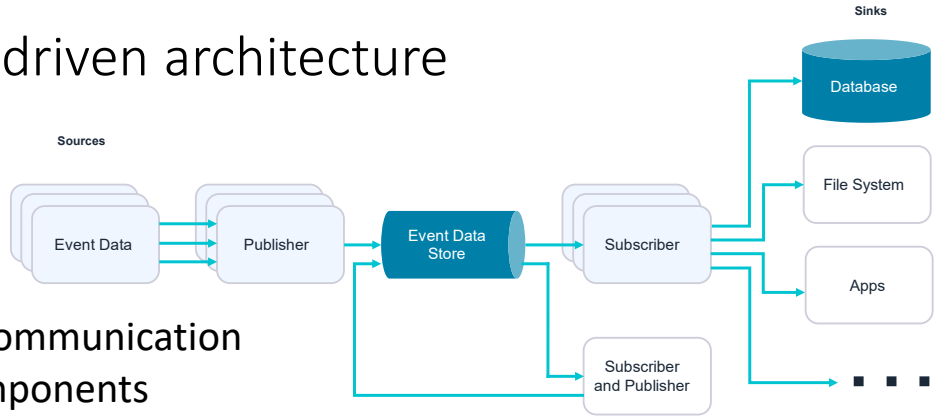


4. Service-Based Architecture

•A variation of Service-Oriented Architecture (**SOA**) that focuses on lightweight services.
•**When to Use** : When you want to build loosely coupled services that can be independently deployed and scaled, but are still part of a larger monolithic system.
•**Example** : A banking system where different services handle account management, transactions, and reporting.



5. Event-driven architecture



asynchronous communication style where components communicate through the production (Publisher) and consumption (Subscriber) of events.

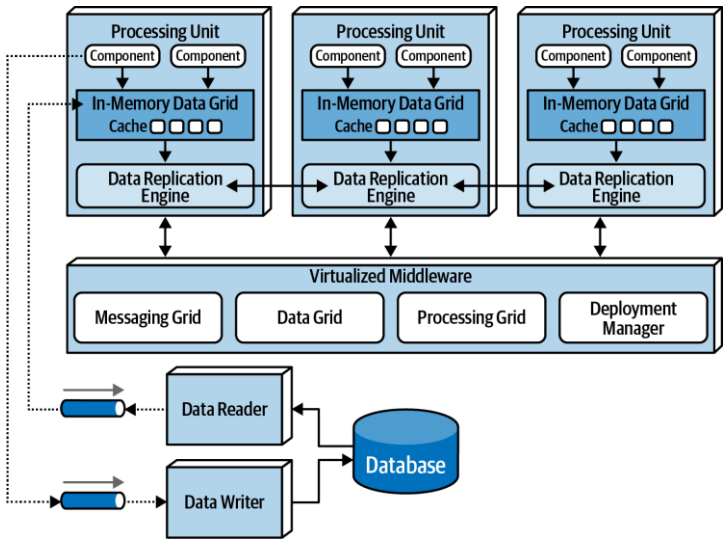
- **When to Use**: Best for real-time systems needing scalability and responsiveness.
- **Example**: A stock trading system where price changes trigger buy/sell orders.

6. Space-based architecture

Uses distributed memory (data grids) to handle high loads and avoid bottlenecks.

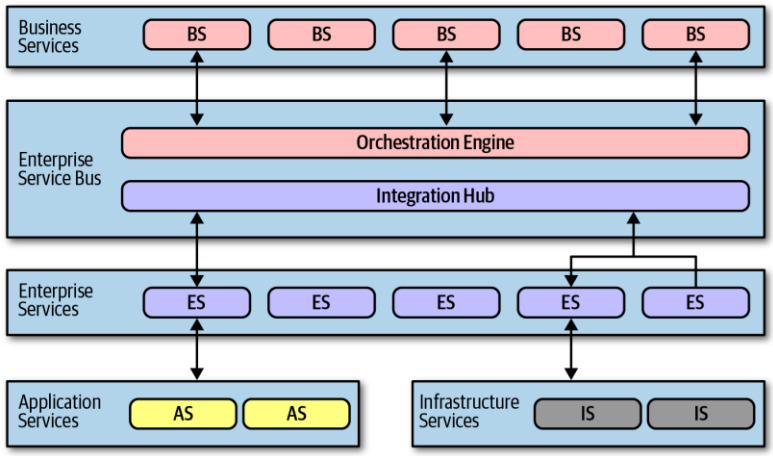
When to Use: Suitable for high-throughput applications that require scalability.

Example: Large-scale online gaming platforms handling millions of concurrent users.



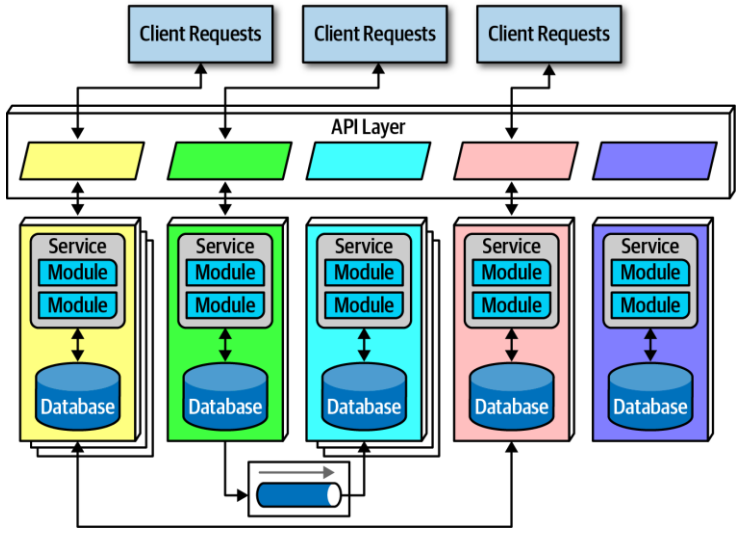
7. Orchestration-Driven Service-Oriented Architecture

A centralized component (orchestrator) coordinates and manages service interactions.
When to Use: Suitable for complex business processes that require workflow management.
Example: A banking system where loan approvals depend on multiple interacting services.



8. Microservices architecture

Breaks down applications into small, independently deployable services communicating via ***APIs***.
When to Use: Best for scalable, cloud-native applications with independent teams managing services.
Example: Netflix, where user management, recommendations, and streaming services run independently.



Comparison Table 1/2

Architecture Style	Key Concept	Best Use Case	Pros	Cons
Layered Architecture	Separation of concerns into layers	Monolithic applications with clear module separation	Easy to manage and test	Can become rigid and slow
Pipeline Architecture	Data processed in sequential stages	Stream processing, ETL pipelines	Modular and easy to scale	Harder to debug failures
Microkernel Architecture	Minimal core with plugins	Extensible applications (e.g., IDEs, OS)	Flexible and stable core	Plugins must be well-defined
Service-Based Architecture	Lightweight services interacting	Simple modular applications	Easier to manage than full SOA	Can grow into complex dependency chains
Event-Driven Architecture	Events trigger actions across services	Real-time, asynchronous systems	Highly scalable and reactive	Complex debugging

Comparison Table 2/2

Architecture Style	Key Concept	Best Use Case	Pros	Cons
Event-Driven Architecture	Events trigger actions across services	Real-time, asynchronous systems	Highly scalable and reactive	Complex debugging
Space-Based Architecture	Distributed memory for high load	High-performance applications	Scales horizontally	Harder to manage consistency
Orchestration-Driven SOA	Centralized service coordination	Complex workflows (e.g., enterprise apps)	Central control simplifies workflows	Orchestrator becomes a bottleneck
Microservices Architecture	Independent services communicating via APIs	Cloud-native, scalable apps	High agility and fault isolation	Requires complex service management