05 Architecture Styles

INT210 Architecture, Integration and Deployment

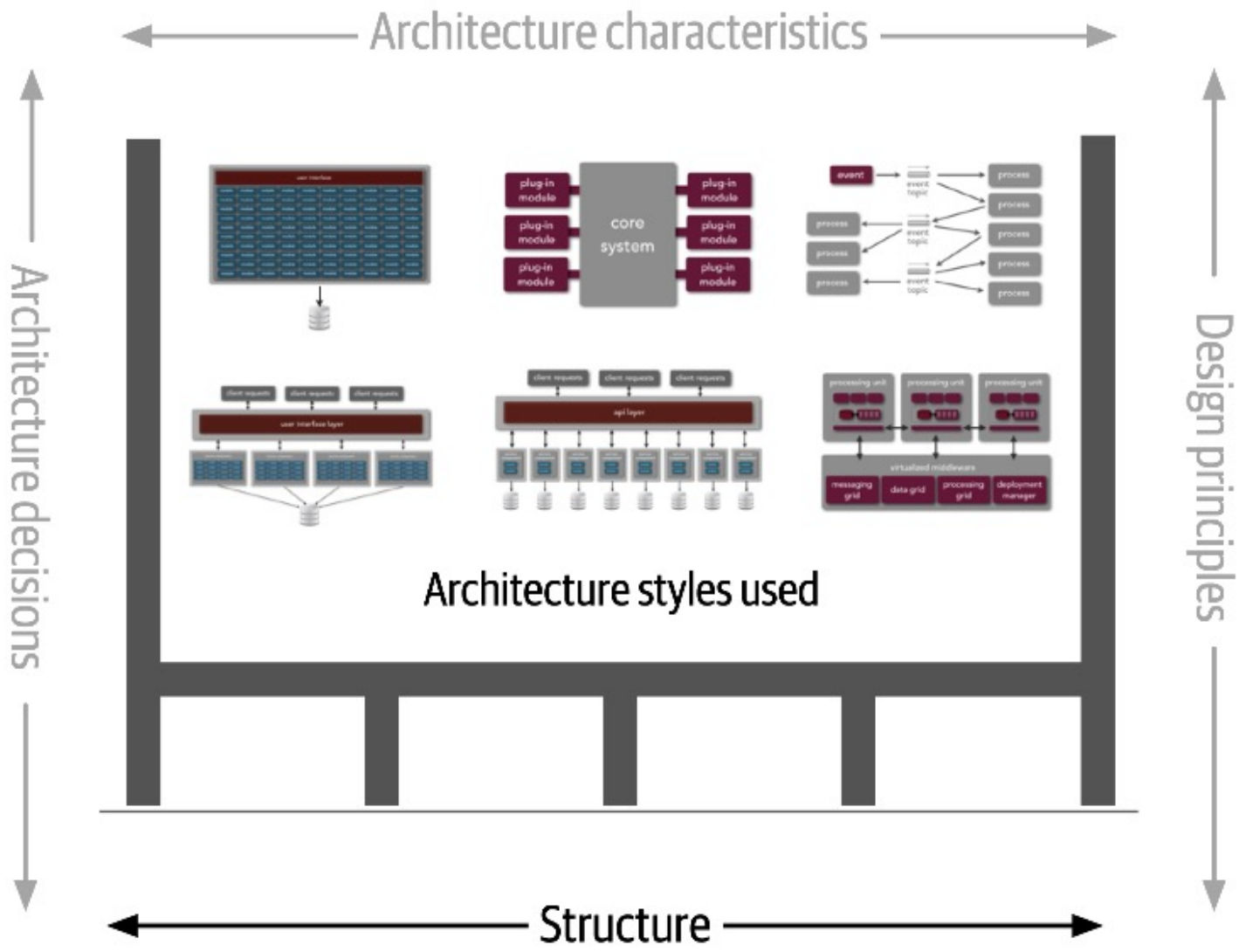


Figure 1-3. Structure refers to the type of architecture styles used in the system

Architectural Styles: foundations

- Big ball of mud – no architecture, no governance of code quality and structure
- Unitary architecture: embedded, desktop apps
- Distributed architecture
  - Client/server: desktop + DB, browser + web server, three-tier, etc.

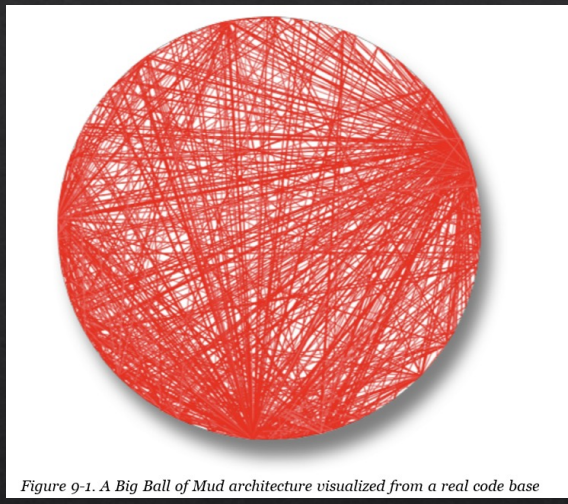


Figure 9-1. A Big Ball of Mud architecture visualized from a real code base

Monolithic styles

- Layered architecture
- Pipeline architecture
- Microkernel architecture

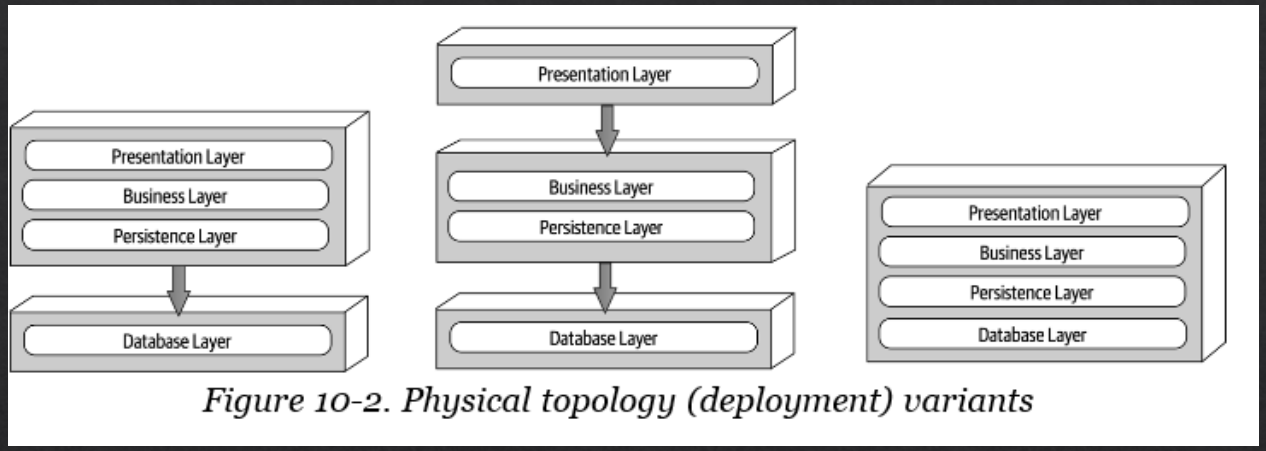


Figure 10-2. Physical topology (deployment) variants

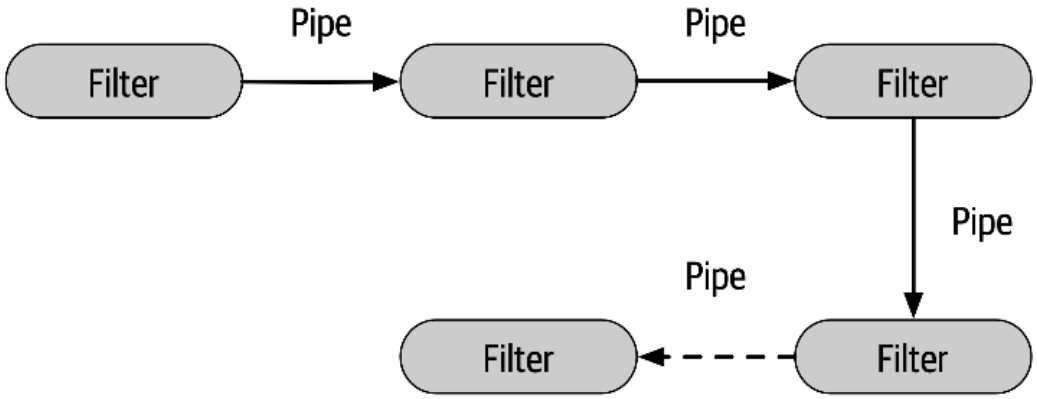


Figure 11-1. Basic topology for pipeline architecture

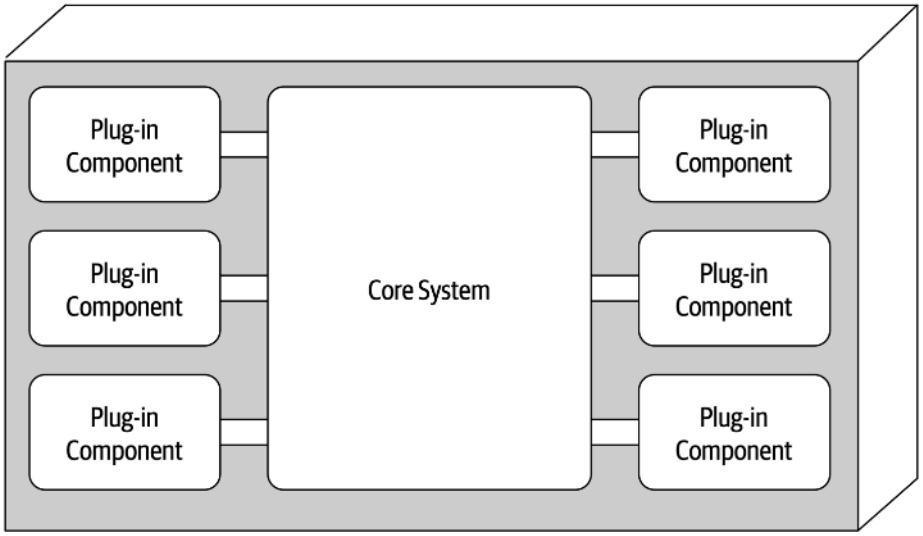


Figure 12-1. Basic components of the microkernel architecture style

Layered architecture (n-tier)

- May be physically deployed as a single unit or as separate units
- Is this technically-partitioned or domain-partitioned?

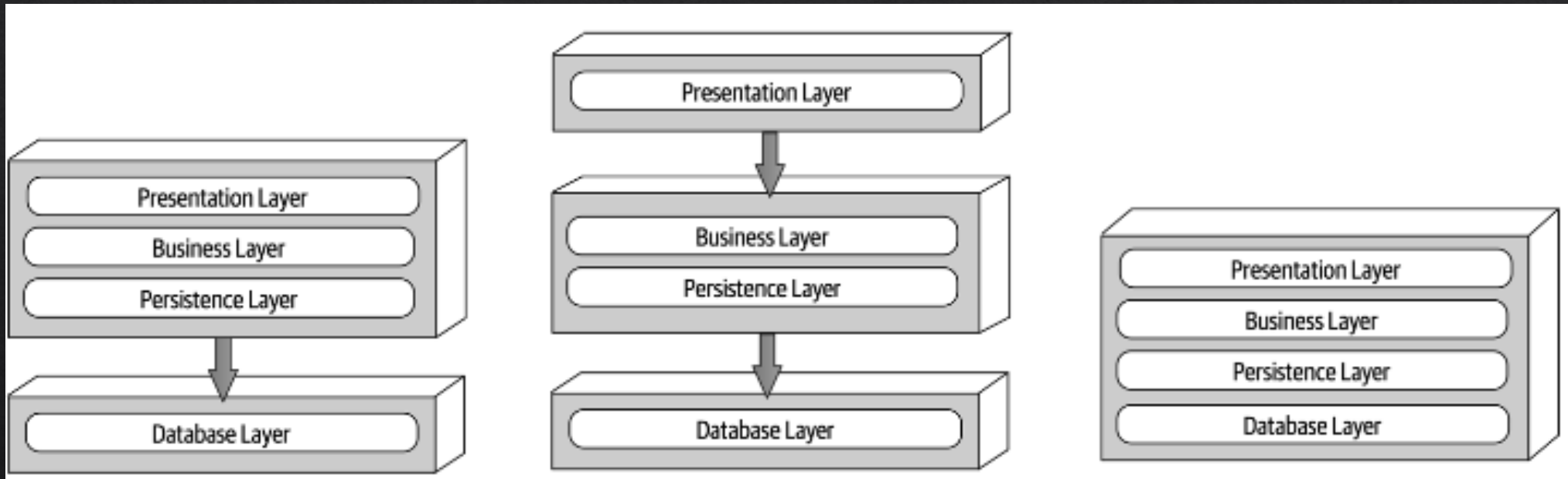


Figure 10-2. Physical topology (deployment) variants

Closed layers

- Separation of concerns
- Layers of isolation

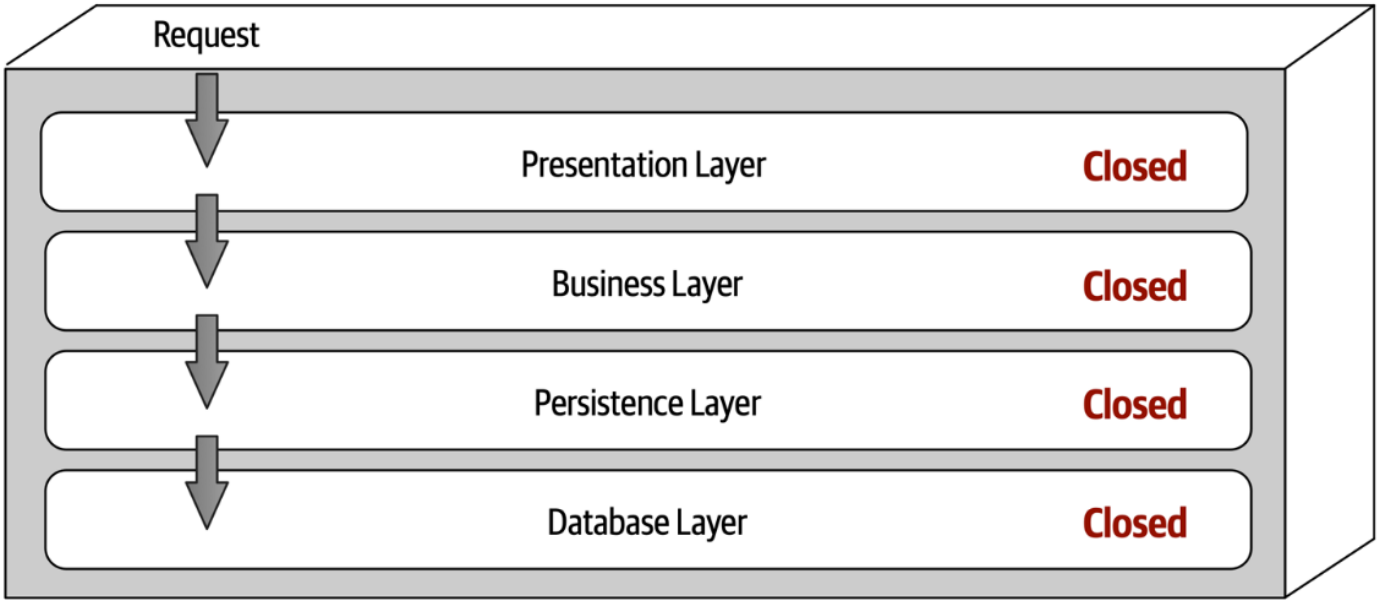


Figure 10-3. Closed layers within the layered architecture

Adding services layer to ‘open’ access

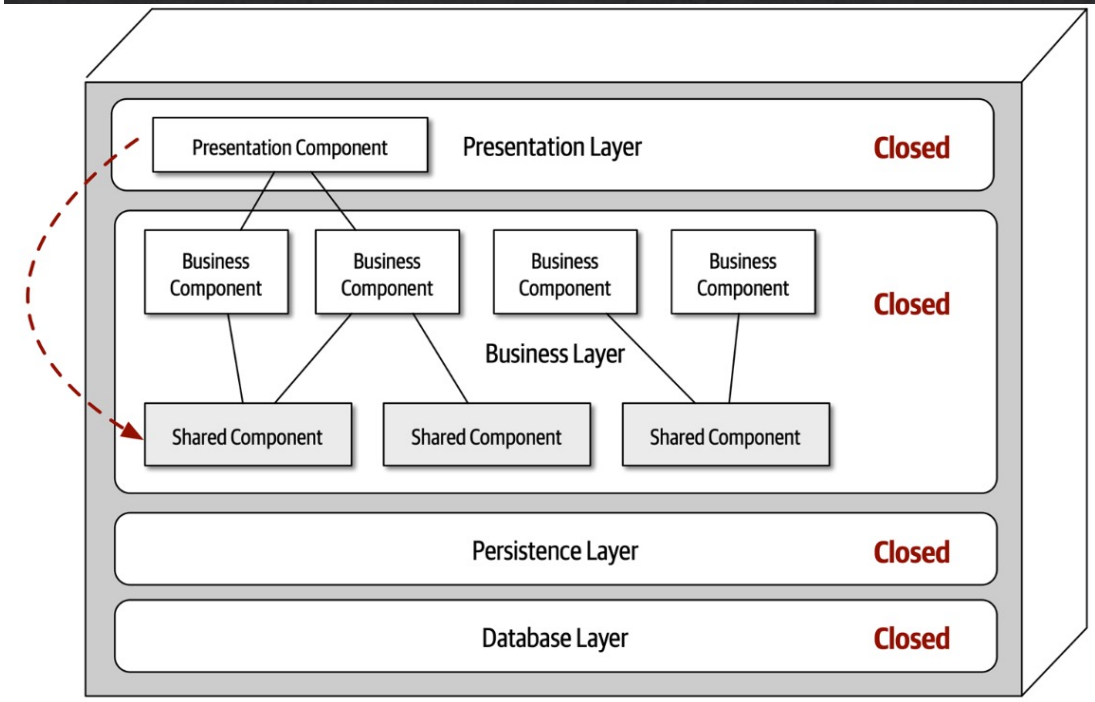


Figure 10-4. Shared objects within the business layer

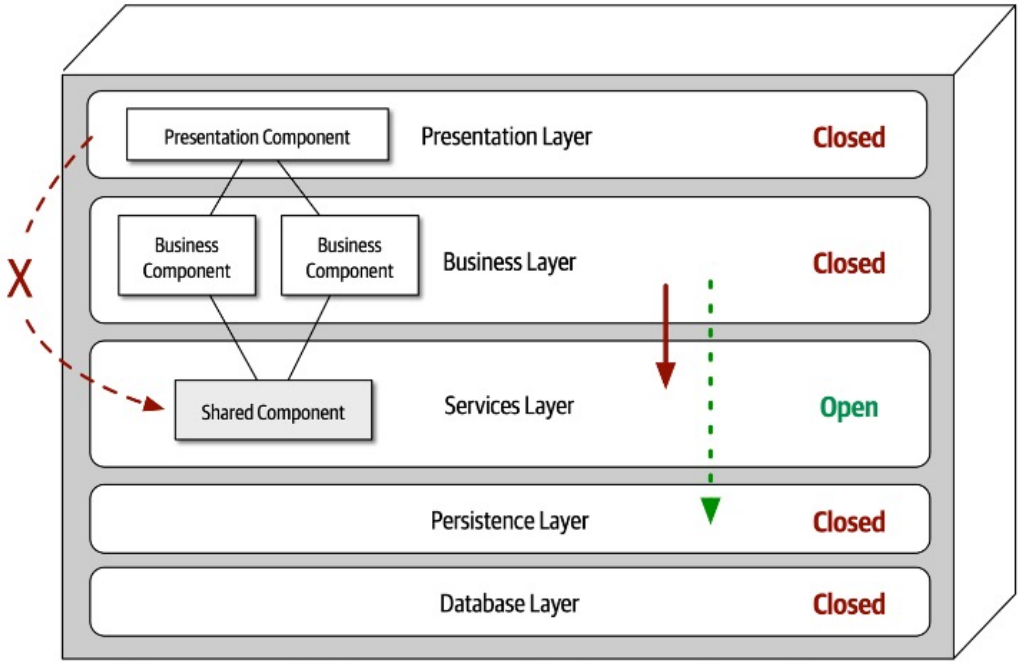


Figure 10-5. Adding a new services layer to the architecture

Layered architecture

- Good for small, simple apps
- Good starting arch
- Low cost/time
- Simple, familiar

Architecture characteristic	Star rating
Partitioning type	Technical
Number of quanta	1
Deployability	★
Elasticity	★
Evolutionary	★
Fault tolerance	★
Modularity	★
Overall cost	★★★★★
Performance	★★
Reliability	★★★
Scalability	★
Simplicity	★★★★★
Testability	★★

Figure 10-6. Layered architecture characteristics ratings

Pipeline architecture

- Filter types
  - Producers: starting point, outbound only
  - Transformer: accepts input, optionally transforms data and forward to outbound pipe
  - Tester: accepts and tests input, may produce output
  - Consumer: termination point of pipeline, persist result to database, display result

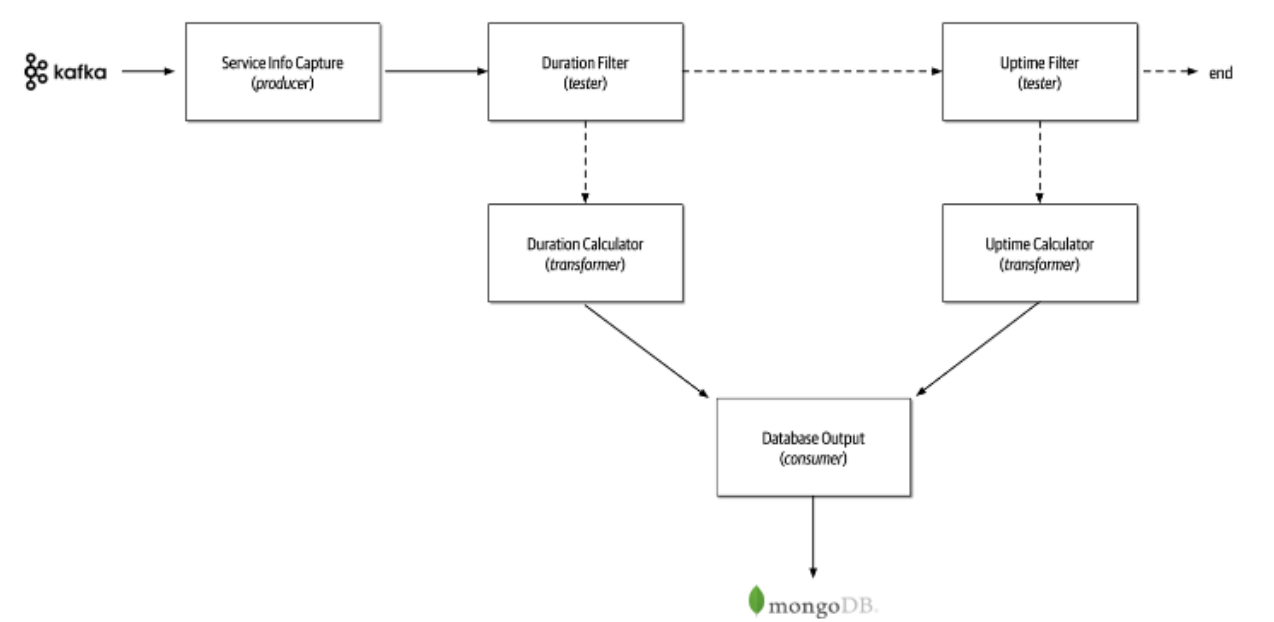


Figure 11-2. Pipeline architecture example

Pipeline architecture

- Technically partitioned
- Low cost, simplicity
- Better deployability and testability than layered

Architecture characteristic	Star rating
Partitioning type	Technical
Number of quanta	1
Deployability	★★
Elasticity	★
Evolutionary	★★★
Fault tolerance	★
Modularity	★★★
Overall cost	★★★★★
Performance	★★
Reliability	★★★
Scalability	★
Simplicity	★★★★★
Testability	★★★

Figure 11-3. Pipeline architecture characteristics ratings

Microkernel architecture

- Core system: minimal functionality required to run the system
- Plug-in components: standalone, independent components that contain specialized processing, additional features and custom code to enhance core

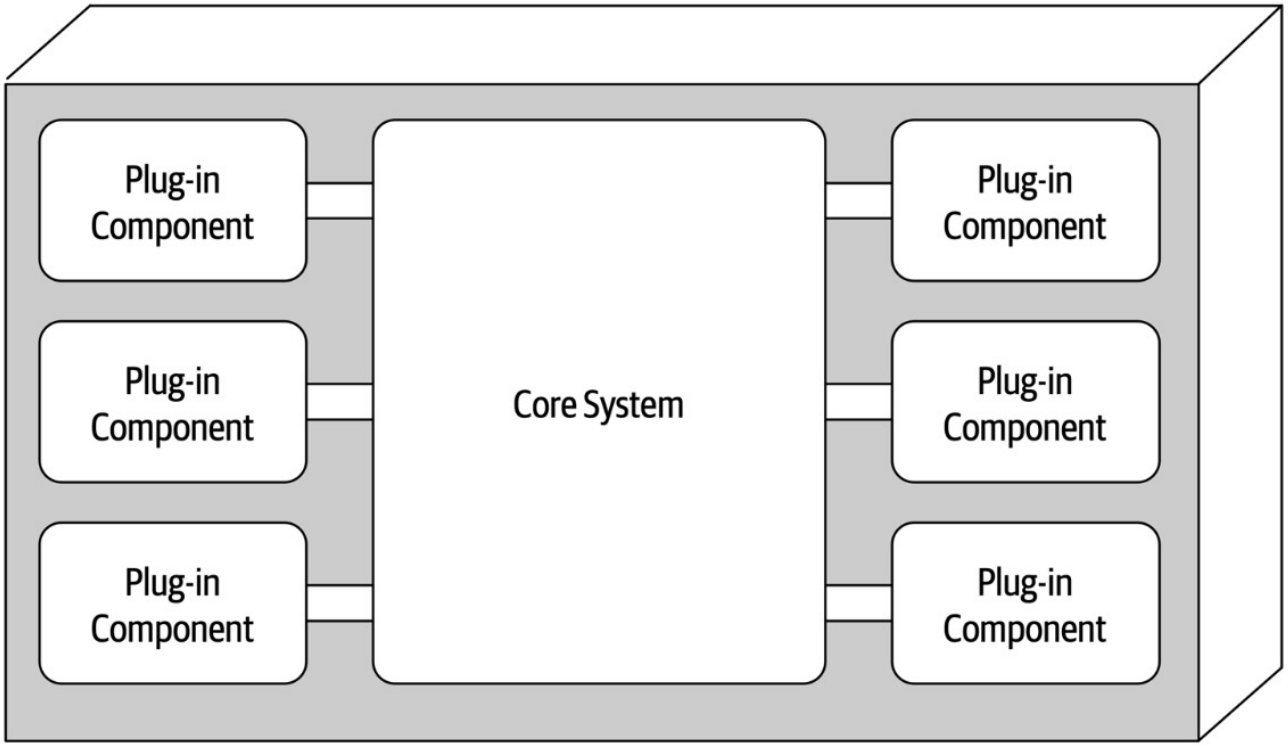


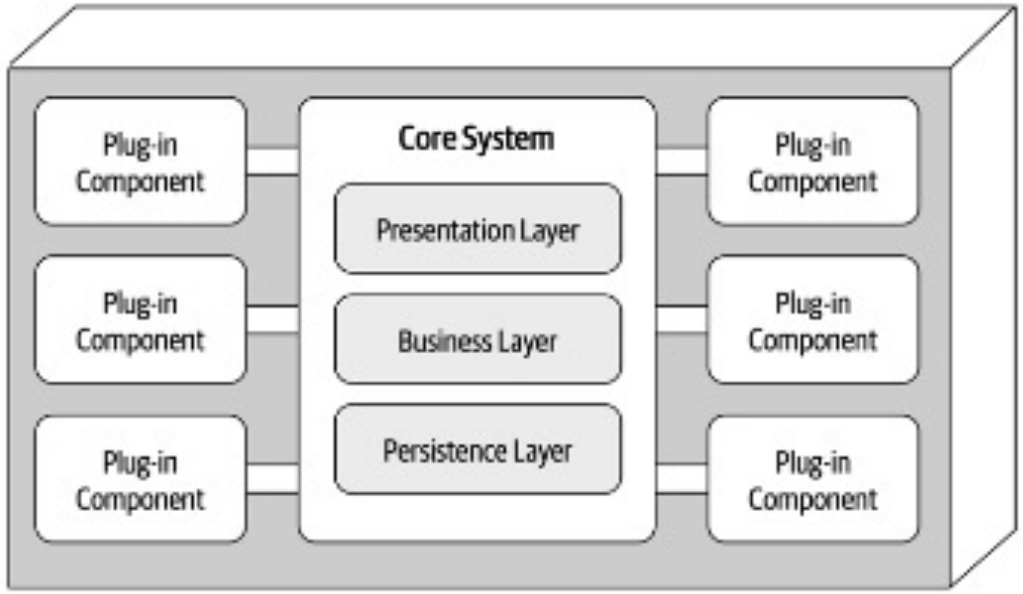
Figure 12-1. Basic components of the microkernel architecture style

Java constructors as components

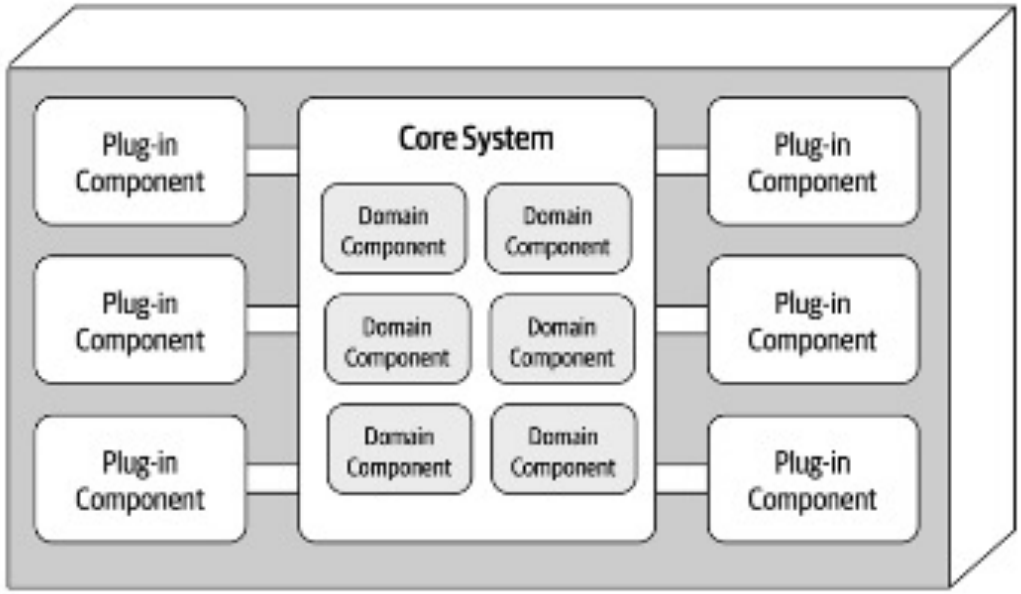
```
public void assessDevice ( String deviceID ) {
   if ( deviceID . equals ( "iPhone6s" )) {
      assessiPhone6s ();
   } else if ( deviceID . equals ( "iPad1" ))
      assessiPad1 ();
   } else if ( deviceID . equals ( "Galaxy5" ))
      assessGalaxy5 ();
   } else ...
      ...
   }
}
```

```
public void assessDevice ( String deviceID ) {
  String plugin = pluginRegistry . get ( deviceID );
  Class <?> theClass = Class . forName ( plugin );
  Constructor <?> constructor = theClass . getConstructor ();
  DevicePlugin devicePlugin =
   ( DevicePlugin ) constructor . newInstance ();
  DevicePlugin . assess ();
}
```

Microkernel architecture variants



Layered Core System (Technically Partitioned)



Modular Core System (Domain Partitioned)

Figure 12-2. Variations of the microkernel architecture core system

UI variants

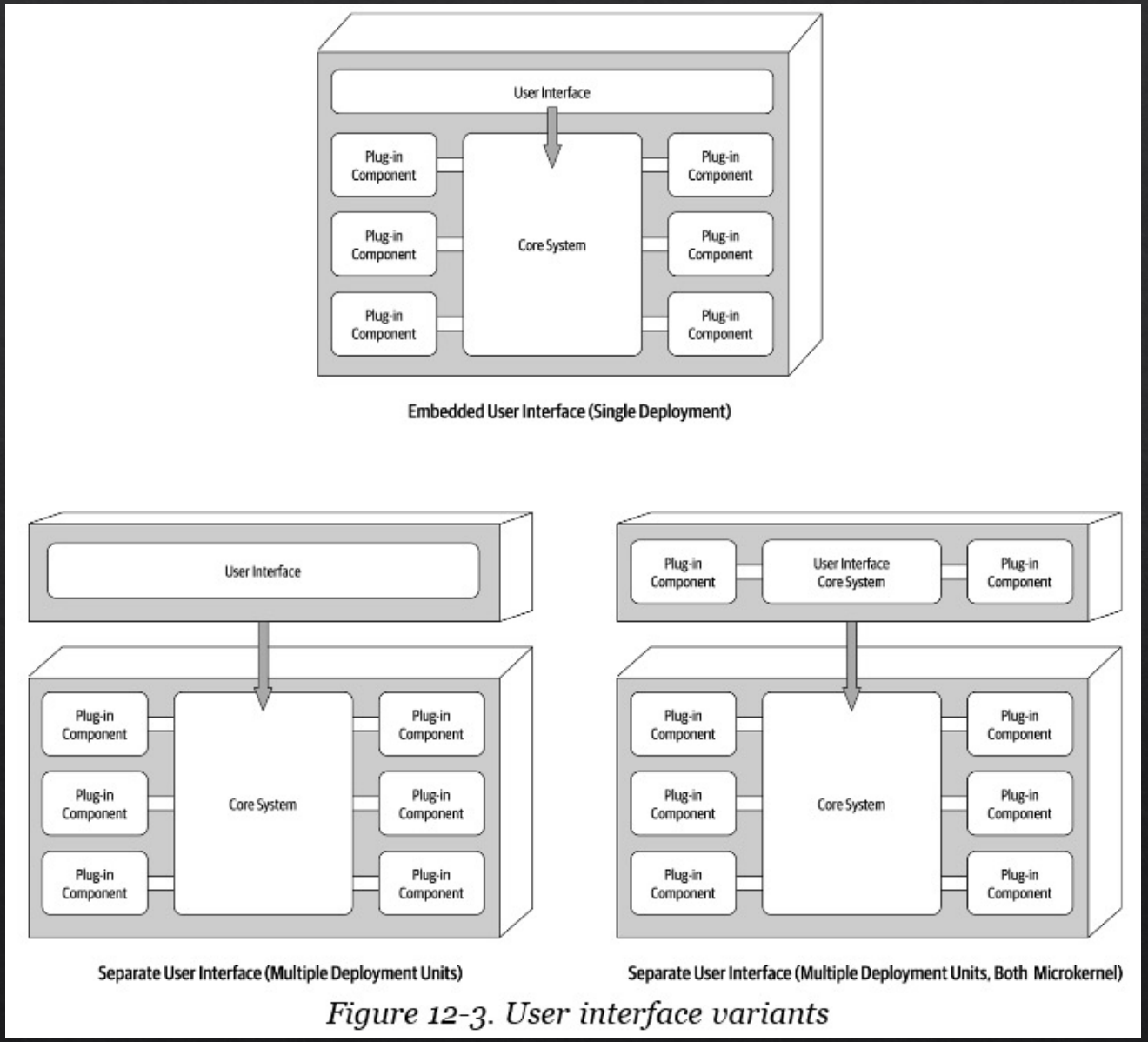


Figure 12-3. User interface variants

Registry

- Plug-in Registry: contains info about each plug-in module (name, data contract << I/O data, remote access protocols, etc.)

```
Map < String , String > registry = new HashMap < String , String >();
static {
  //point-to-point access example
  registry . put ( "iPhone6s" , "Iphone6sPlugin" );

  //messaging example
  registry . put ( "iPhone6s" , "iphone6s.queue" );

  //restful example
  registry . put ( "iPhone6s" , "https://atlas:443/assess/iphone6s" );
}
```

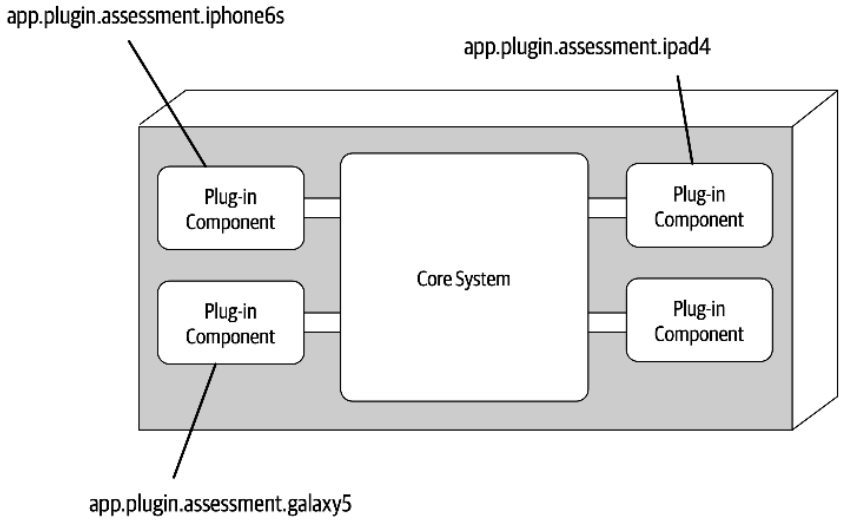


Figure 12-5. Package or namespace plug-in implementation

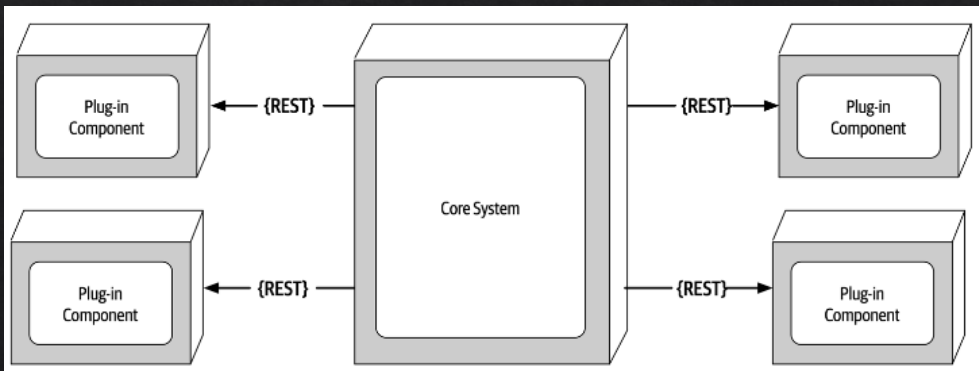


Figure 12-6. Remote plug-in access using REST

Contracts

- Contracts – defines overall behavior and output

```
public interface AssessmentPlugin {
  public AssessmentOutput assess ();
  public String register ();
  public String deregister ();
}
   public class AssessmentOutput {
  public String assessmentReport ;
  public Boolean resell ;
  public Double value ;
  public Double resellPrice ;
}
```

Microkernel characteristics

- Low cost and simple
- Improved modularity and extendibility
- Improved testability, deployability and reliability
- Can be technically- and domain-partitioned

Architecture characteristic	Star rating
Partitioning type	Domain and technical
Number of quanta	1
Deployability	★★★
Elasticity	★
Evolutionary	★★★
Fault tolerance	★
Modularity	★★★
Overall cost	★★★★★
Performance	★★★
Reliability	★★★
Scalability	★
Simplicity	★★★★
Testability	★★★

Distributed architecture issues

- Network is NOT reliable: data/messages may be lost
- Network has latency: affects system performance, chaining multiple services, should know average and max (95^{th} percentile+) latency
- Bandwidth is not infinite: may need to service thousands of requests/sec
- Network is NOT secure: must implement security functions
- Network topology may change: affects latency assumptions, may cause timeouts
- There are many network administrators: ???
- The network adds to system/service costs: equipment, cloud, transactions
- Network equipment may not have the same performance/standards

Service-based architecture

- Microservices style architecture
- Not as complex as other distributed architecture
- Distributed macro layered structure (UI, components, single DB)
- Components cover domains of the app
- Not too many services (4-12)
- Single instance of each domain service

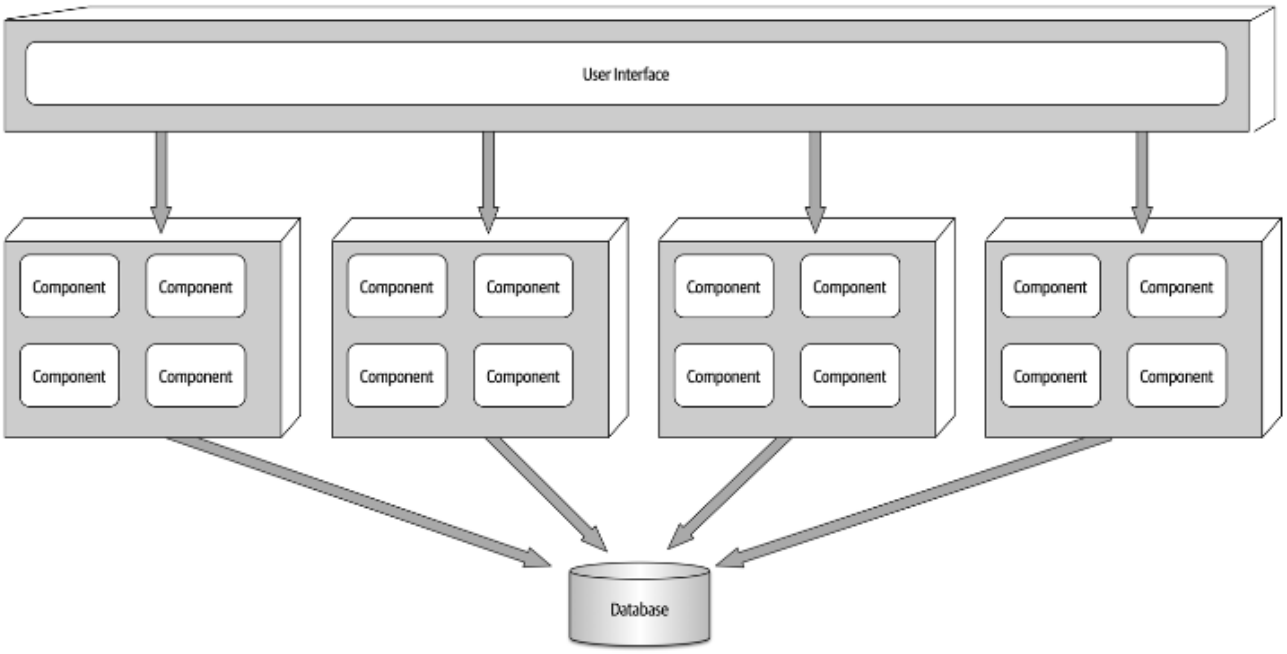


Figure 13-1. Basic topology of the service-based architecture style

Separate UI, DB

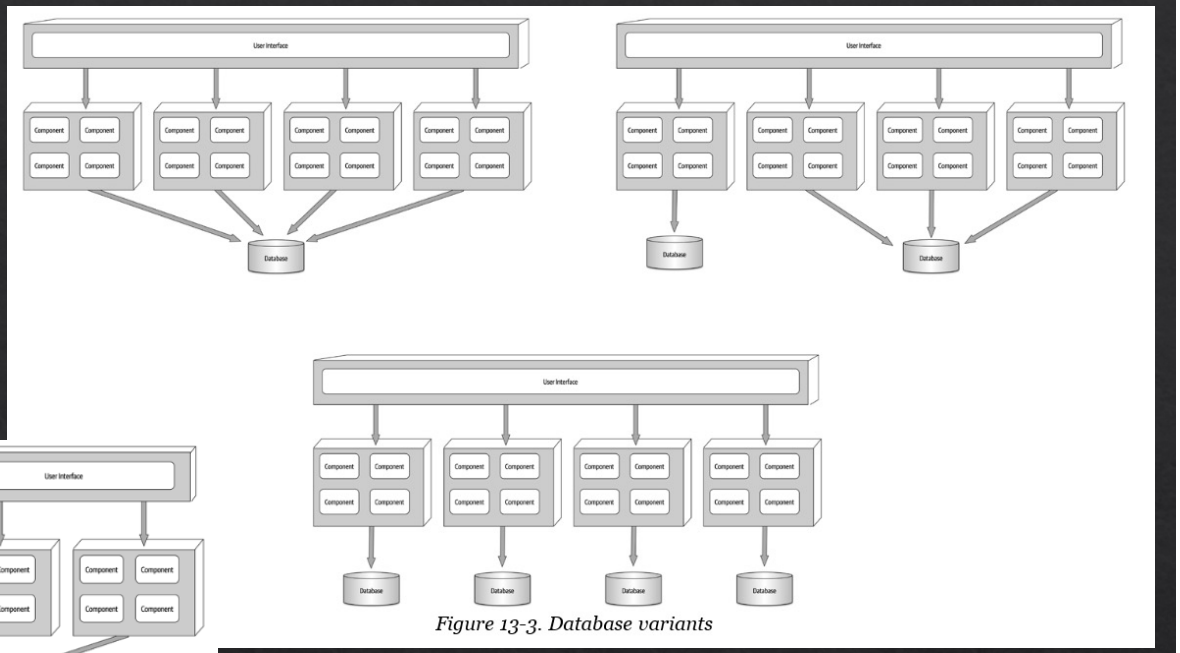
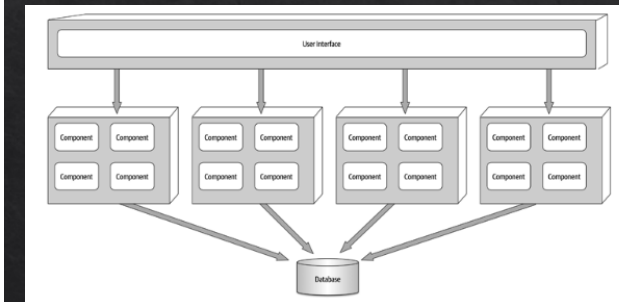
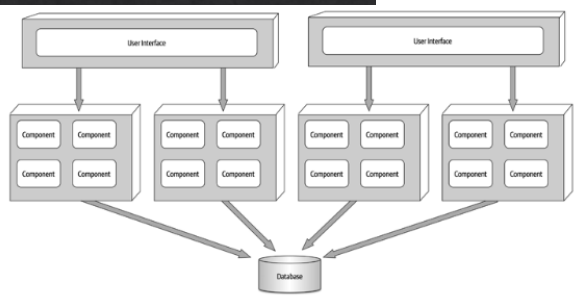


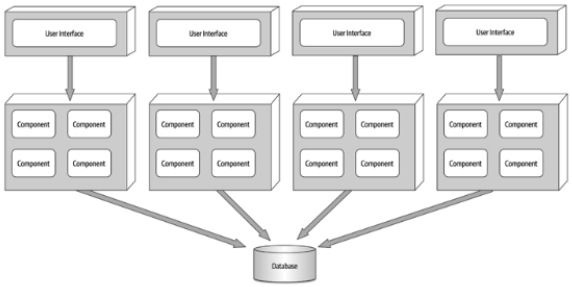
Figure 13-3. Database variants



Single Monolithic User Interface



Domain-Based User Interface



Service-Based User Interface

Can have additional API layer

- To provide service to external systems

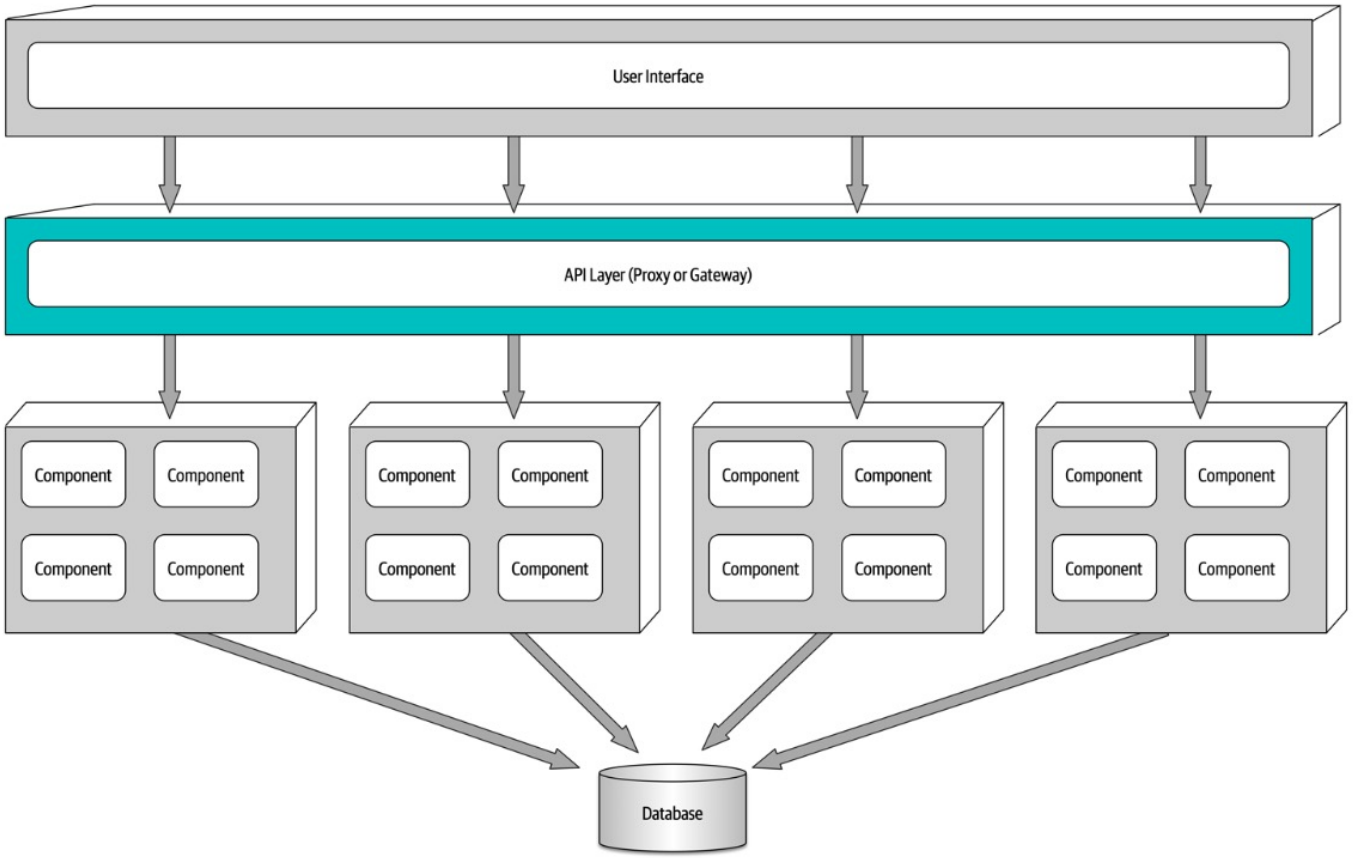


Figure 13-4. Adding an API layer between the user interface and domain services

Example architecture

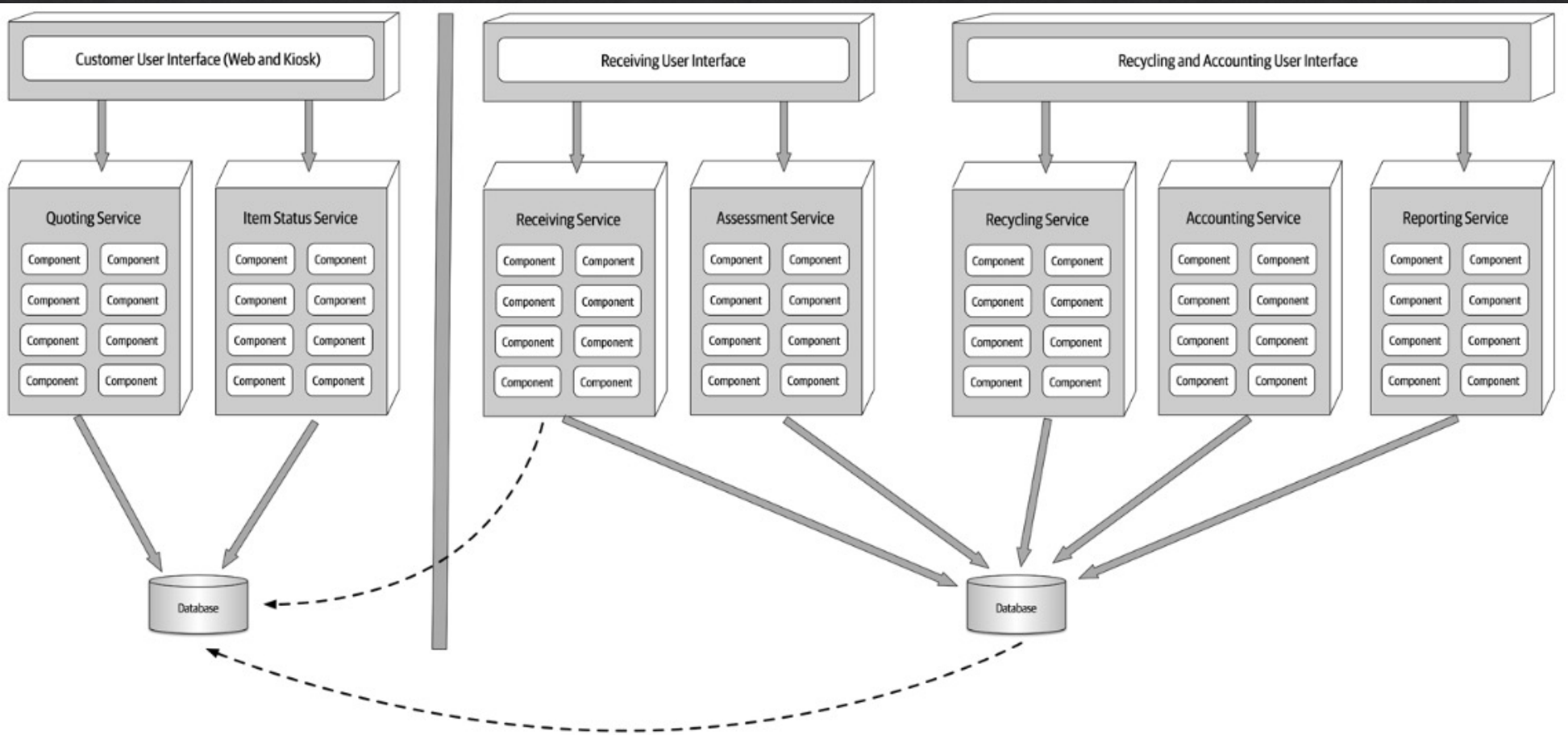


Figure 13-8. Electronics recycling example using service-based architecture

Service-based arch rating

- Domain partitioned << domain-driven design
- Agility
- Testability
- Deployability
- Services are coarse grained: limited scalability >> bigger service instances; only some services need to scale
- Single DB makes consistency easy

Architecture characteristic	Star rating
Partitioning type	Domain
Number of quanta	1 to many
Deployability	★★★★
Elasticity	★★
Evolutionary	★★★
Fault tolerance	★★★★
Modularity	★★★★
Overall cost	★★★★
Performance	★★★
Reliability	★★★★
Scalability	★★★
Simplicity	★★★
Testability	★★★★

Event-driven architecture style

- Distributed asynchronous style
- Highly scalable, high-performance apps
- Adaptable to simple/complex apps
- Can be combined/embedded into other styles

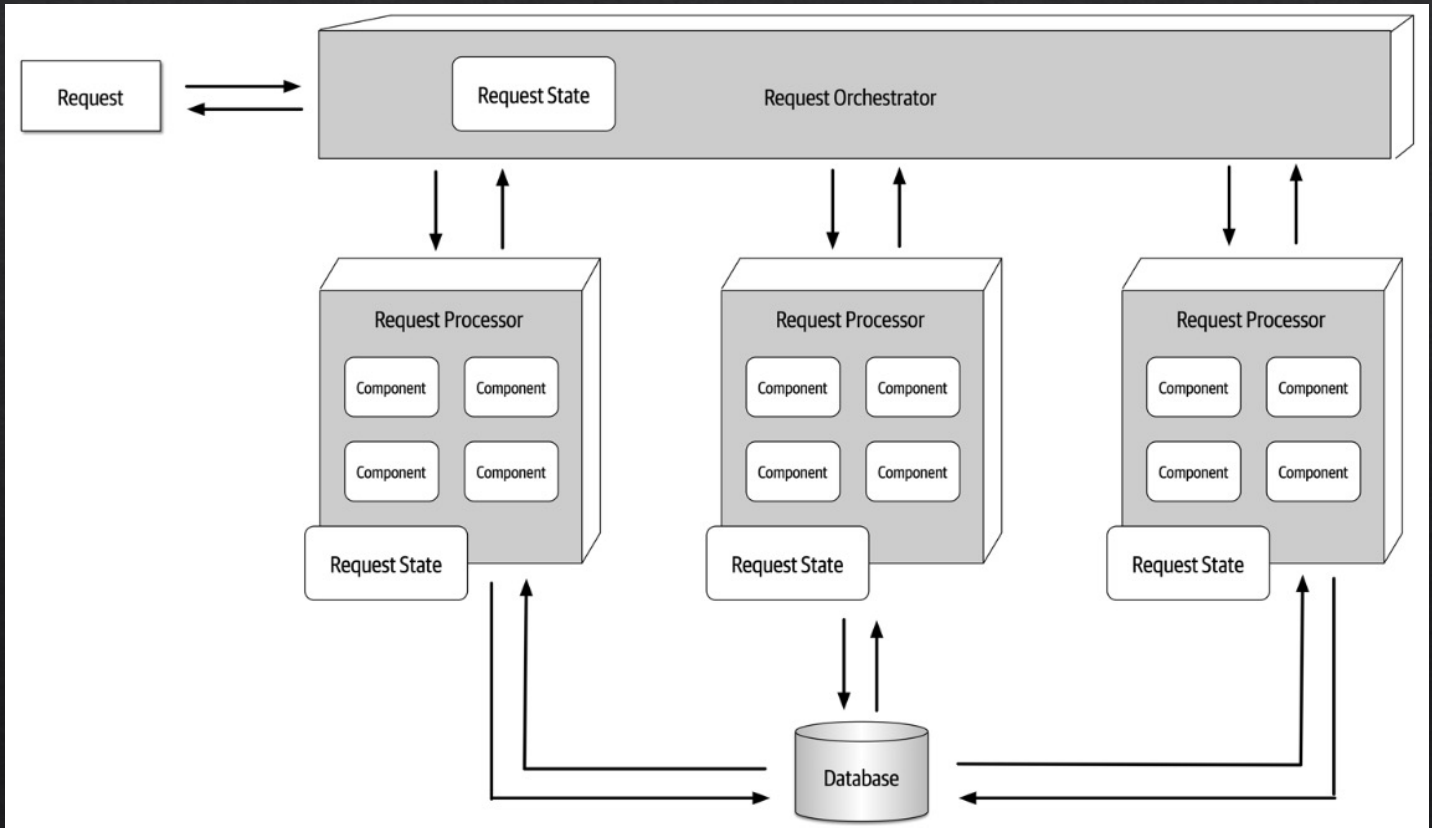


Figure 14-1. Request-based model

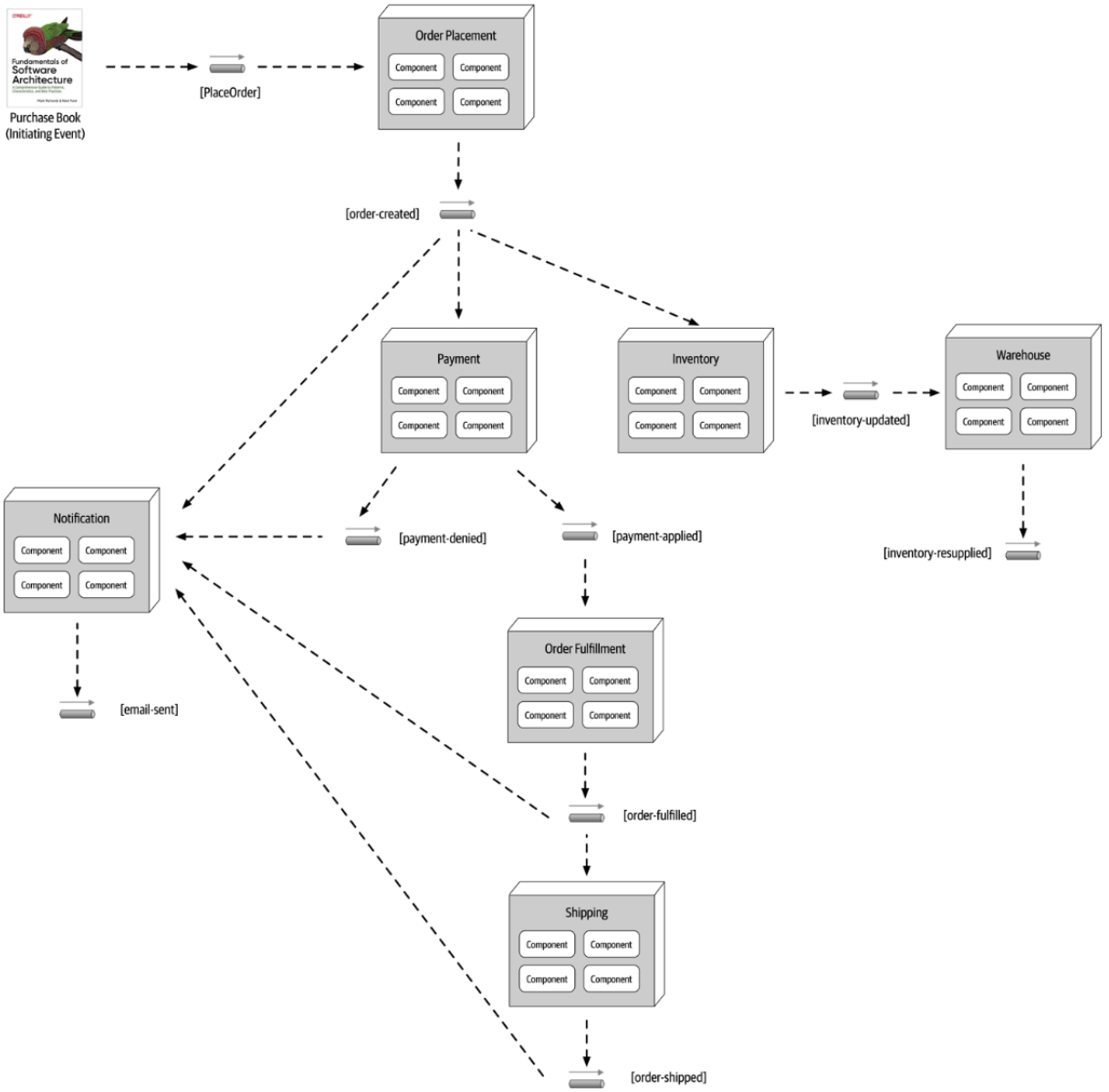
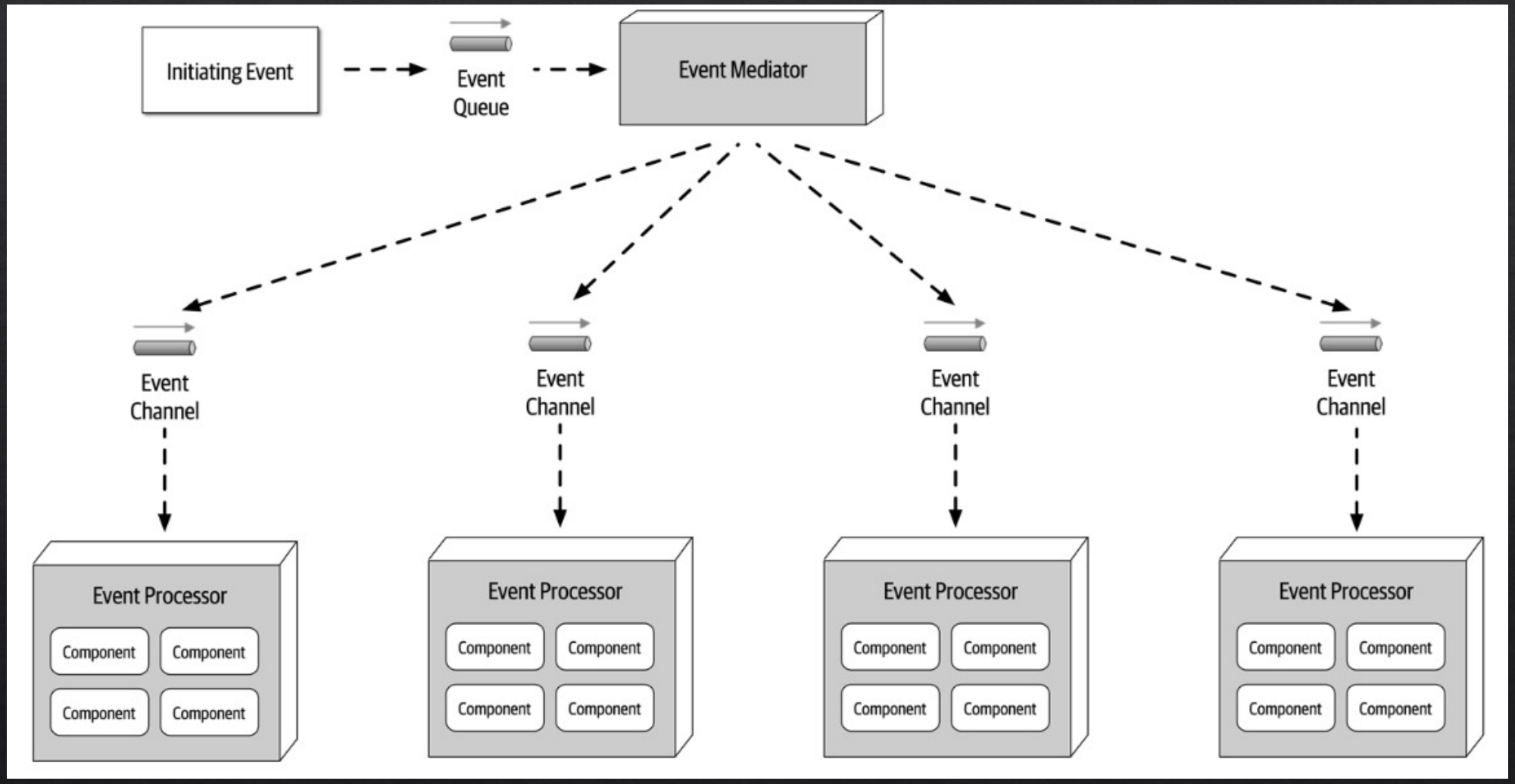


Figure 14-4. Example of the broker topology


Initiating Event
Event Queue
Event Mediator
Event Channel
Event Channel
Event Channel
Event Channel
Event Processor
Component
Component
Component
Component
Event Processor
Component
Component
Component
Component
Event Processor
Component
Component
Component
Component
Event Processor
Component
Component
Component
Component

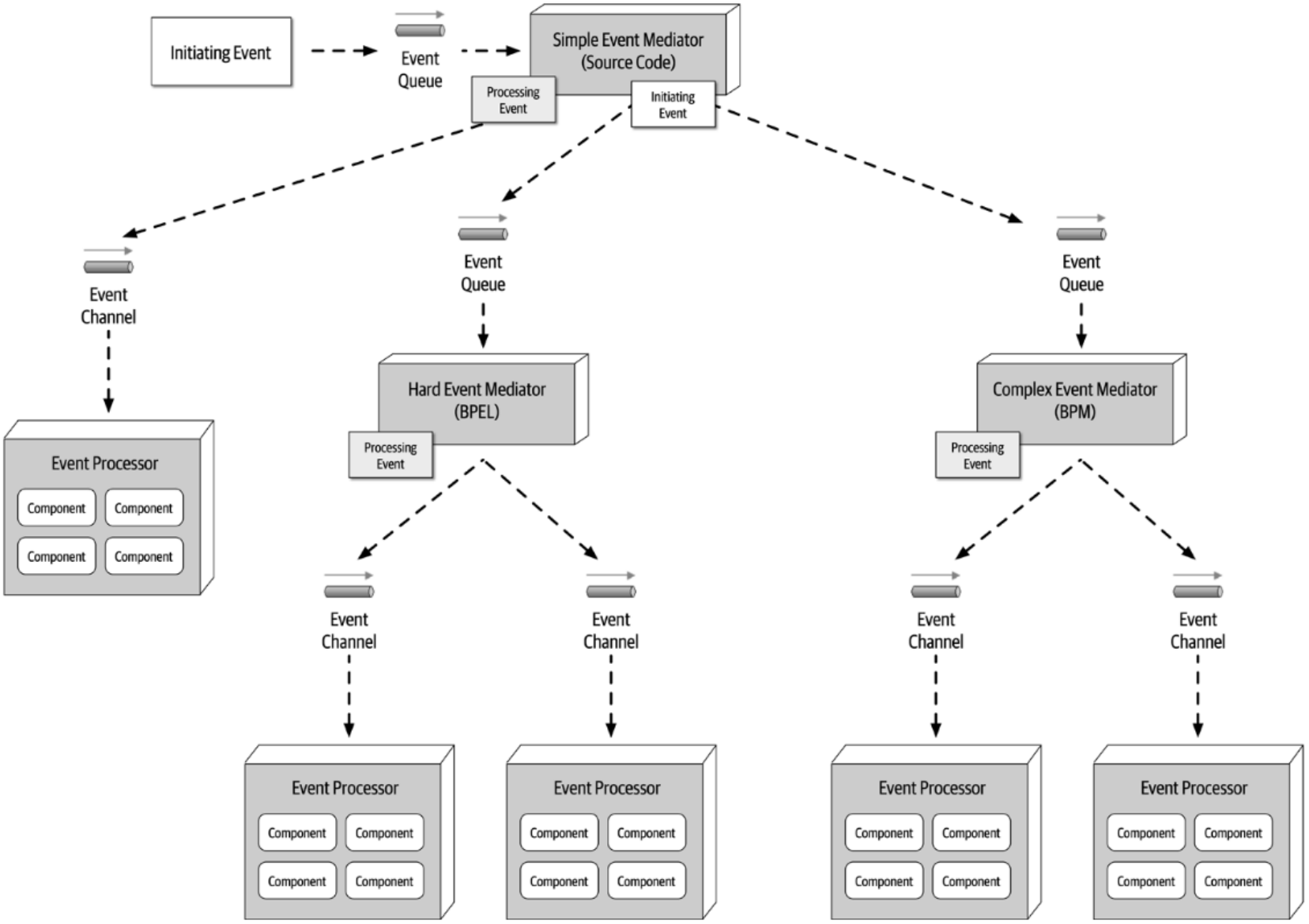


Figure 14-6. Delegating the event to the appropriate type of event mediator

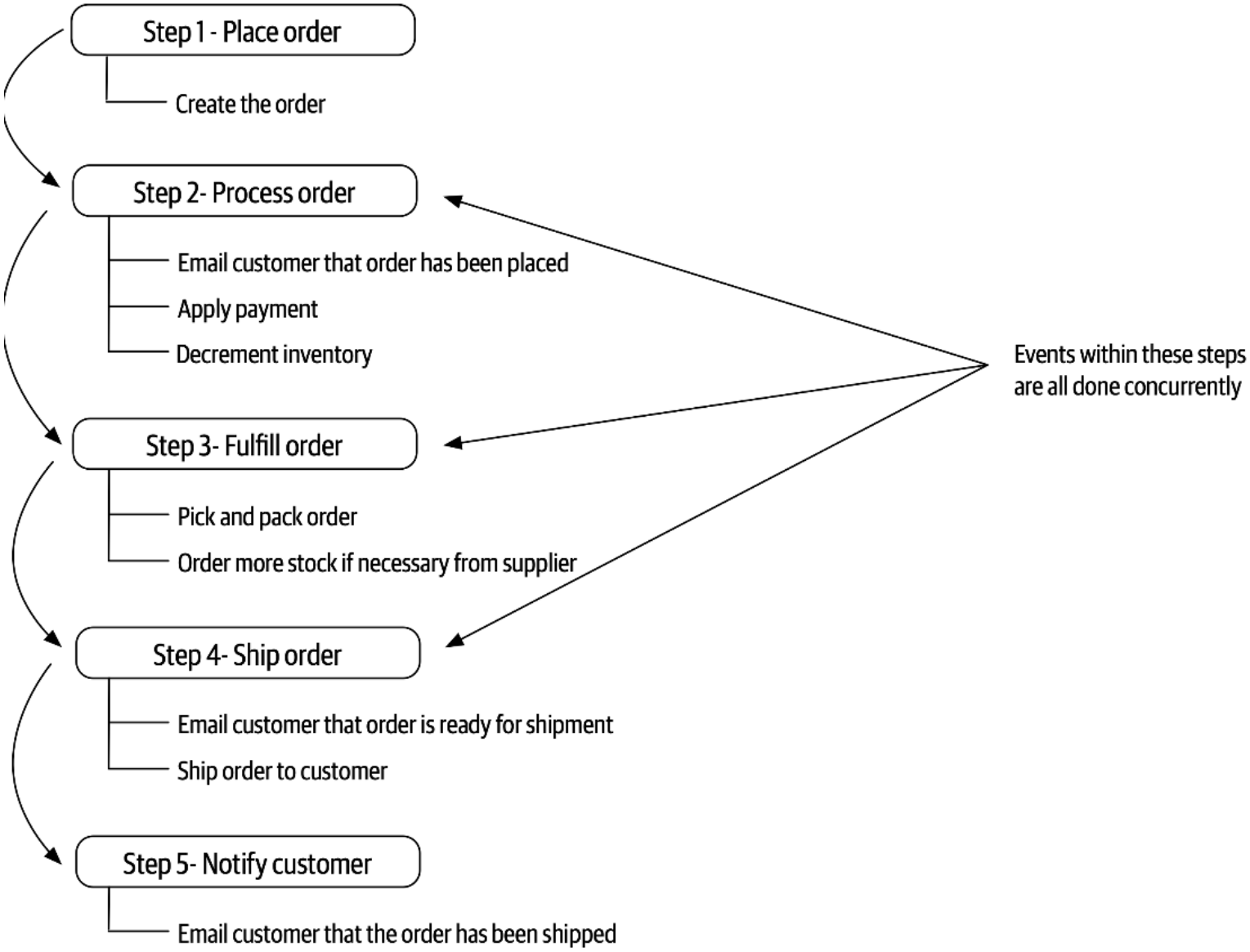


Figure 14-7. Mediator steps for placing an order

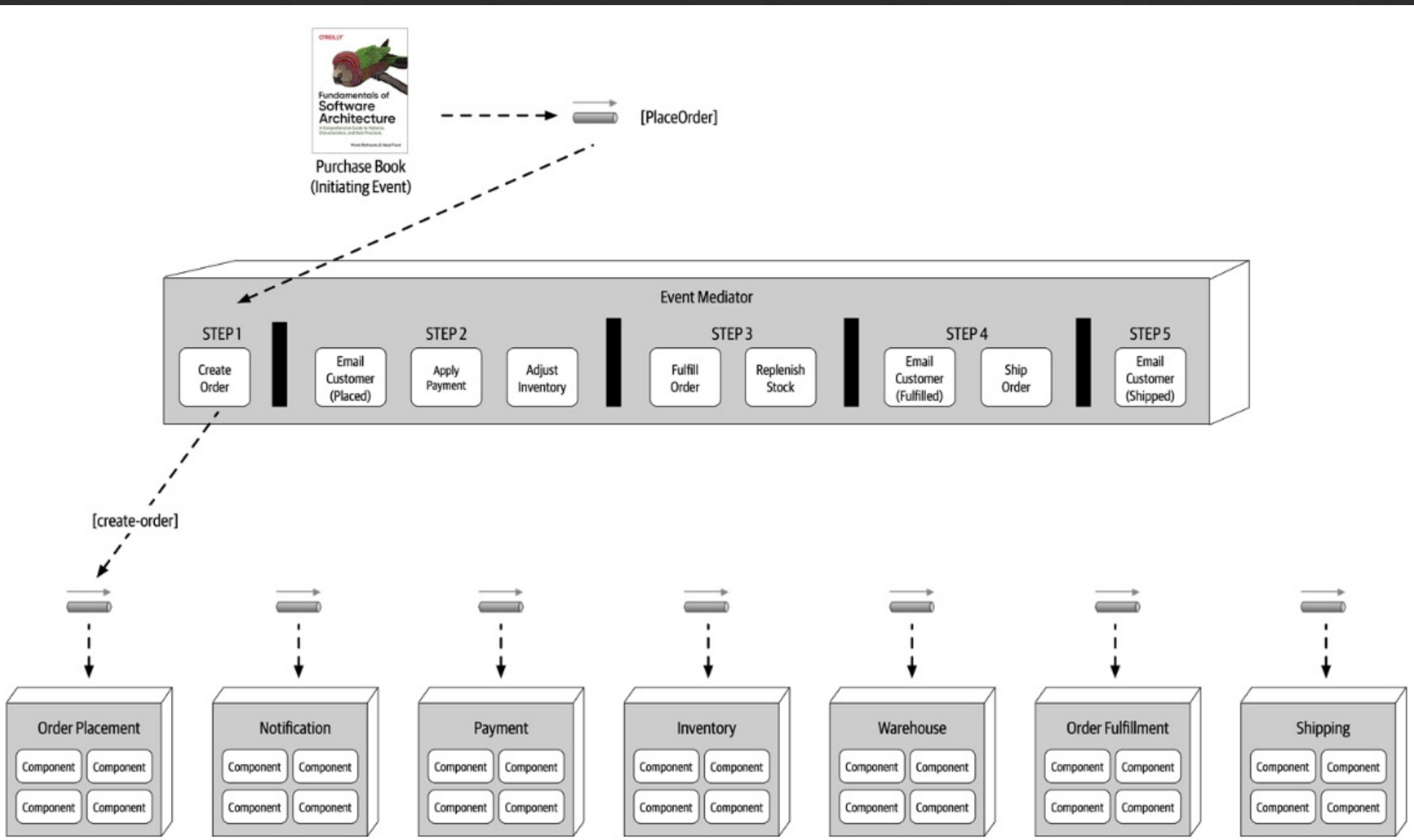


Figure 14-8. Step 1 of the mediator example

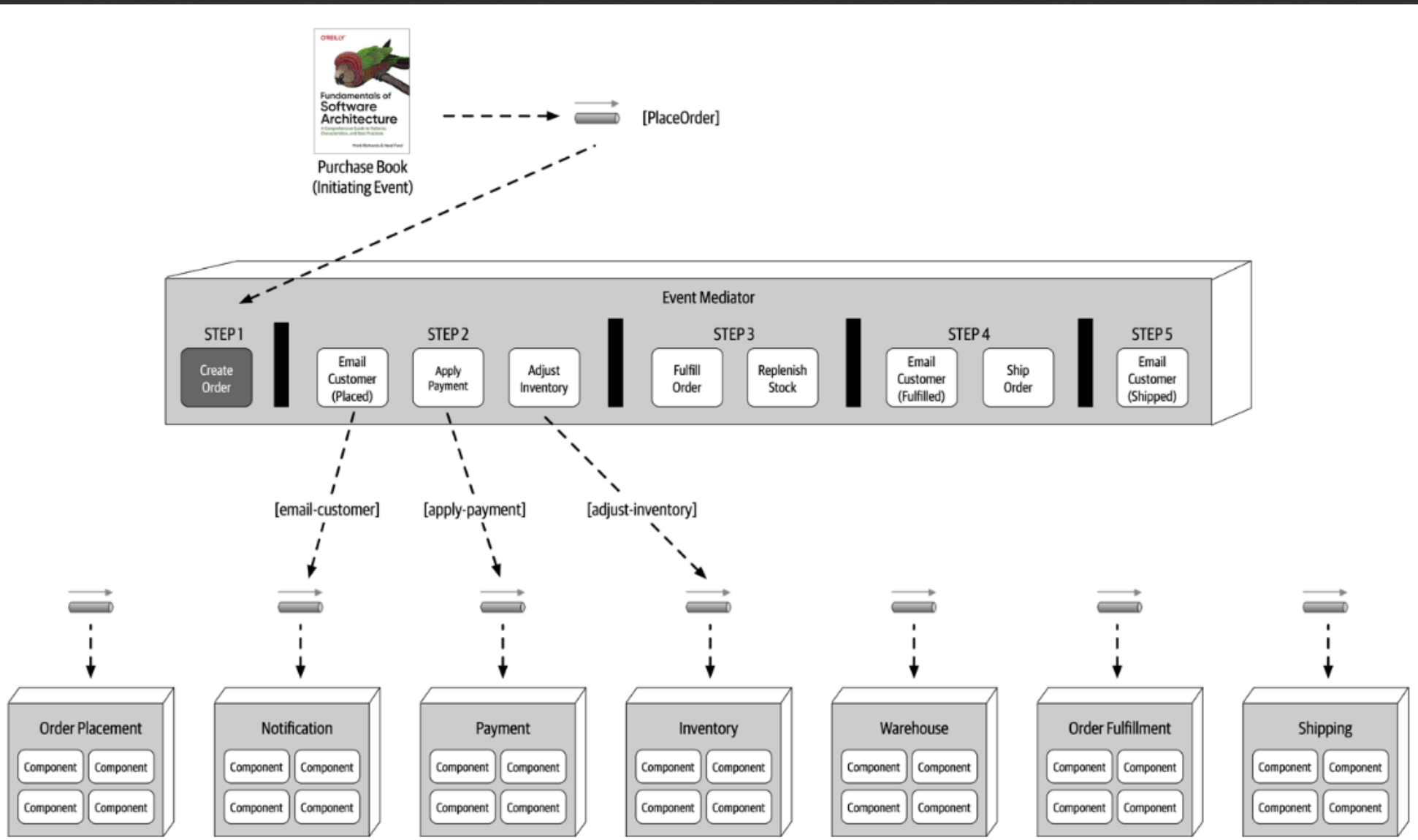


Figure 14-9. Step 2 of the mediator example

Event-driven

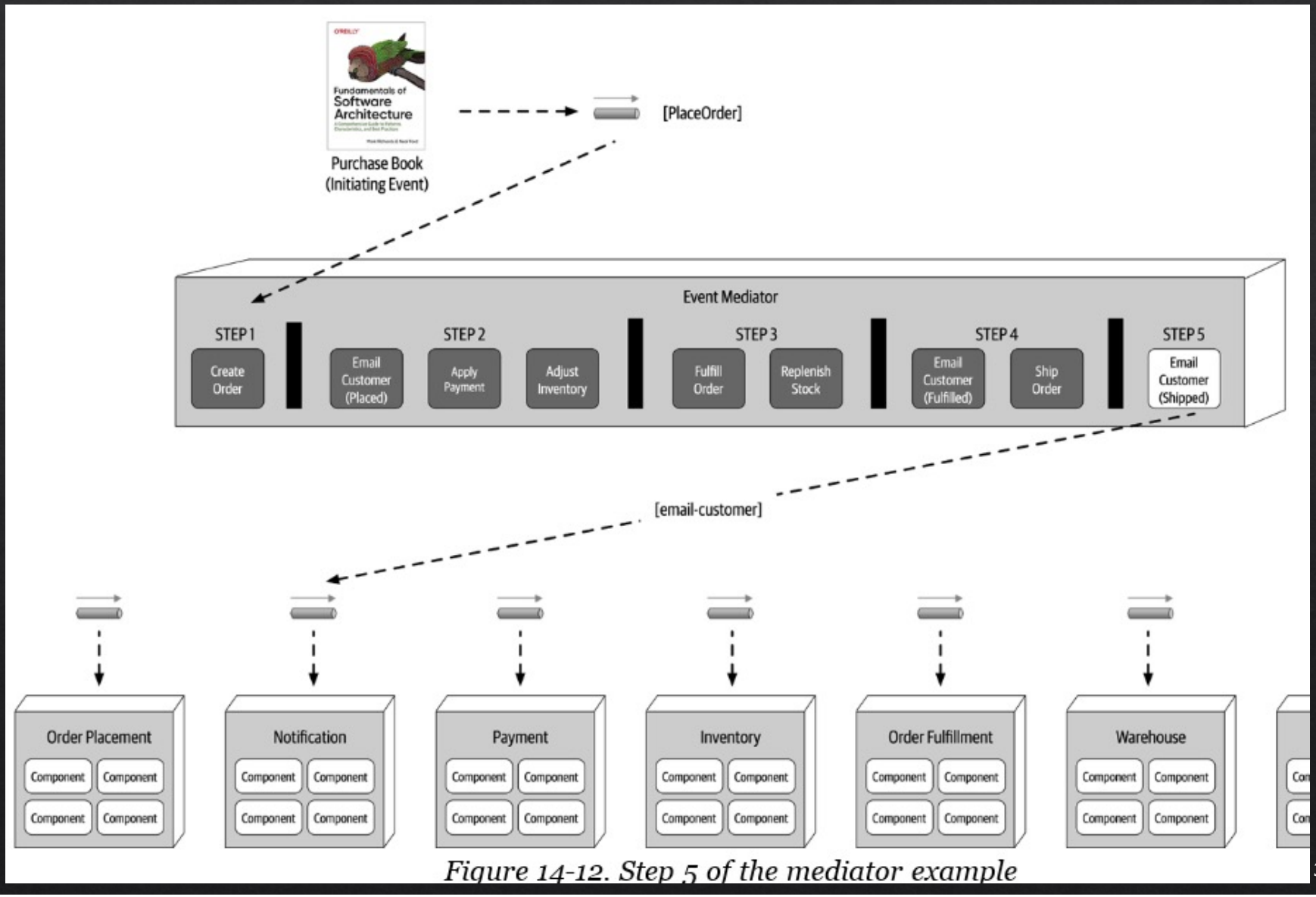


Figure 14-12. Step 5 of the mediator example

Event-driven

- Technically partitioned
- Performance, scalability, fault tolerance
- NOT simple, hard to test
- Highly evolutionary, can easily add event processors

Architecture characteristic	Star rating
Partitioning type	Technical
Number of quanta	1 to many
Deployability	★★★
Elasticity	★★★
Evolutionary	★★★★★
Fault tolerance	★★★★★
Modularity	★★★★
Overall cost	★★★
Performance	★★★★★
Reliability	★★★
Scalability	★★★★★
Simplicity	★
Testability	★★

07 Architecture Styles

INT210 Architecture, Integration and Deployment

Space-based architecture style

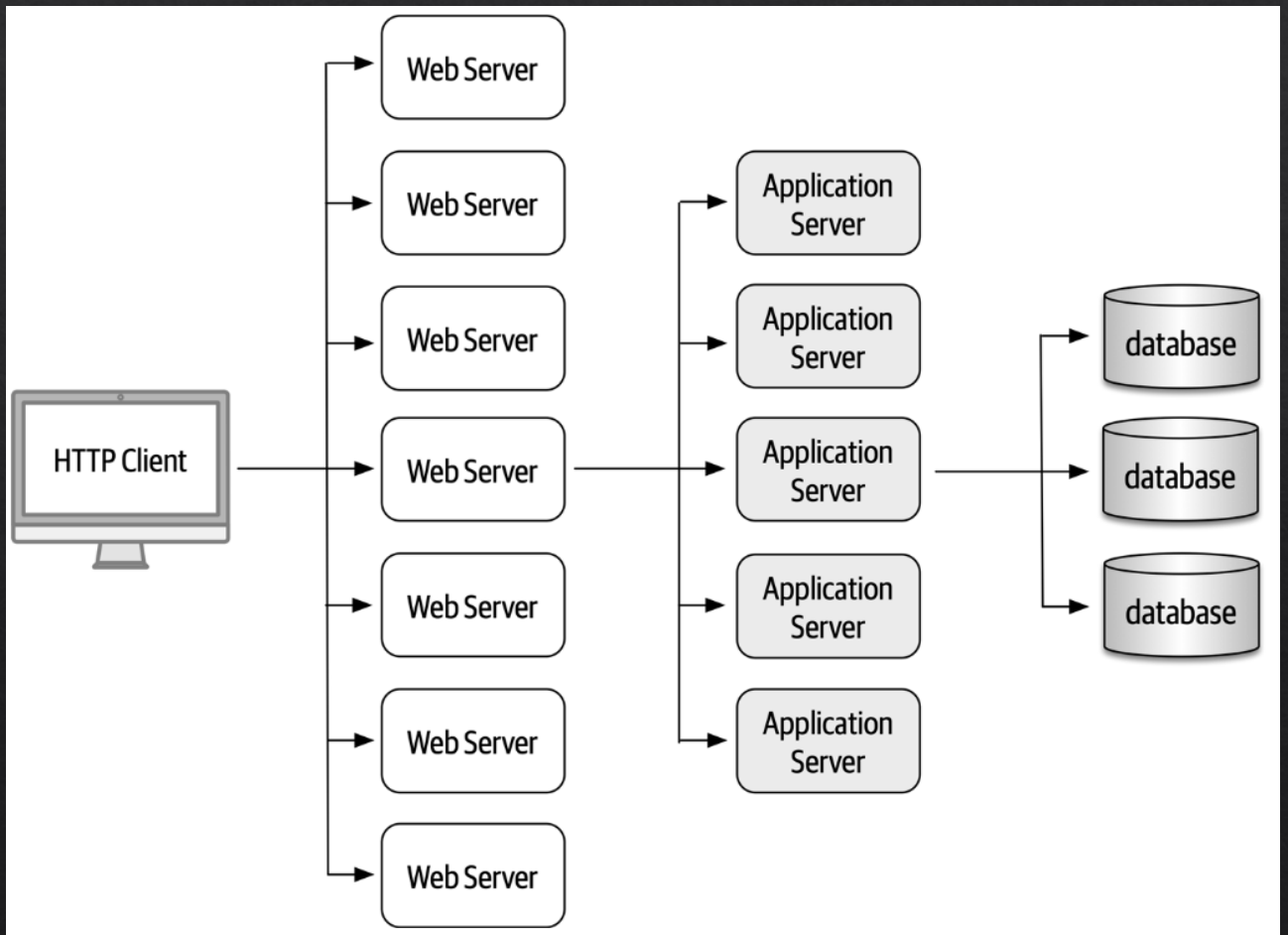


Figure 15-1. Scalability limits within a traditional web-based topology

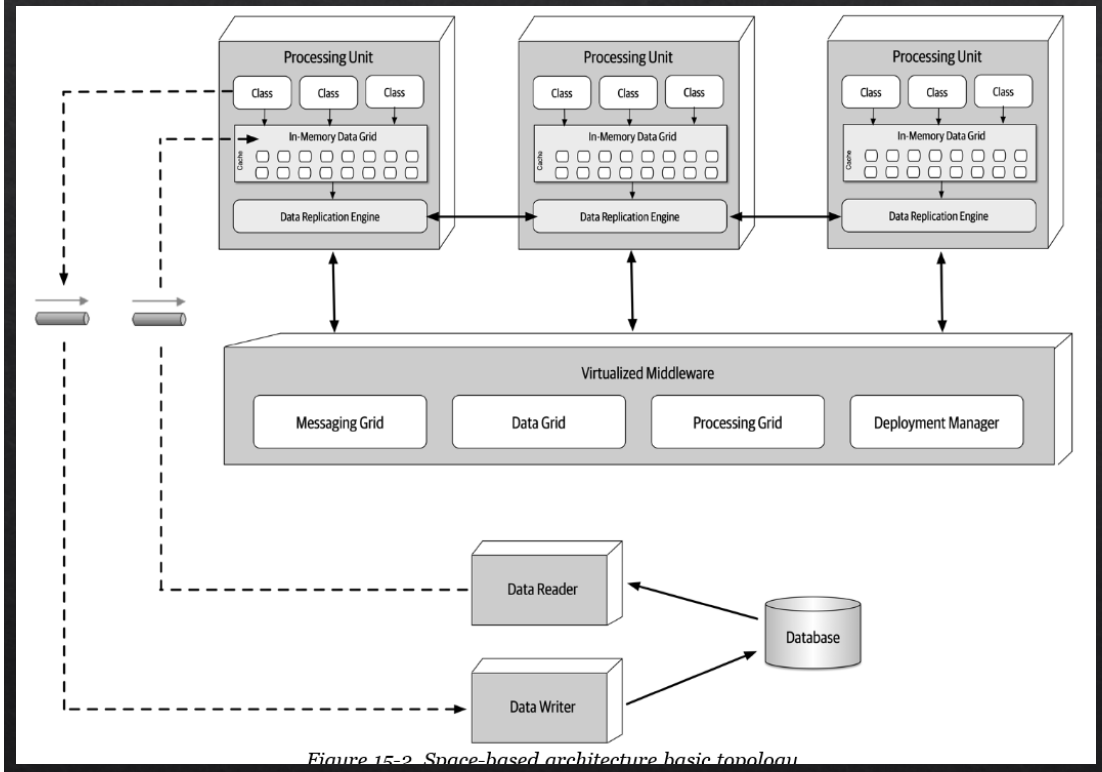


Figure 15-3. Space-based architecture basic topology

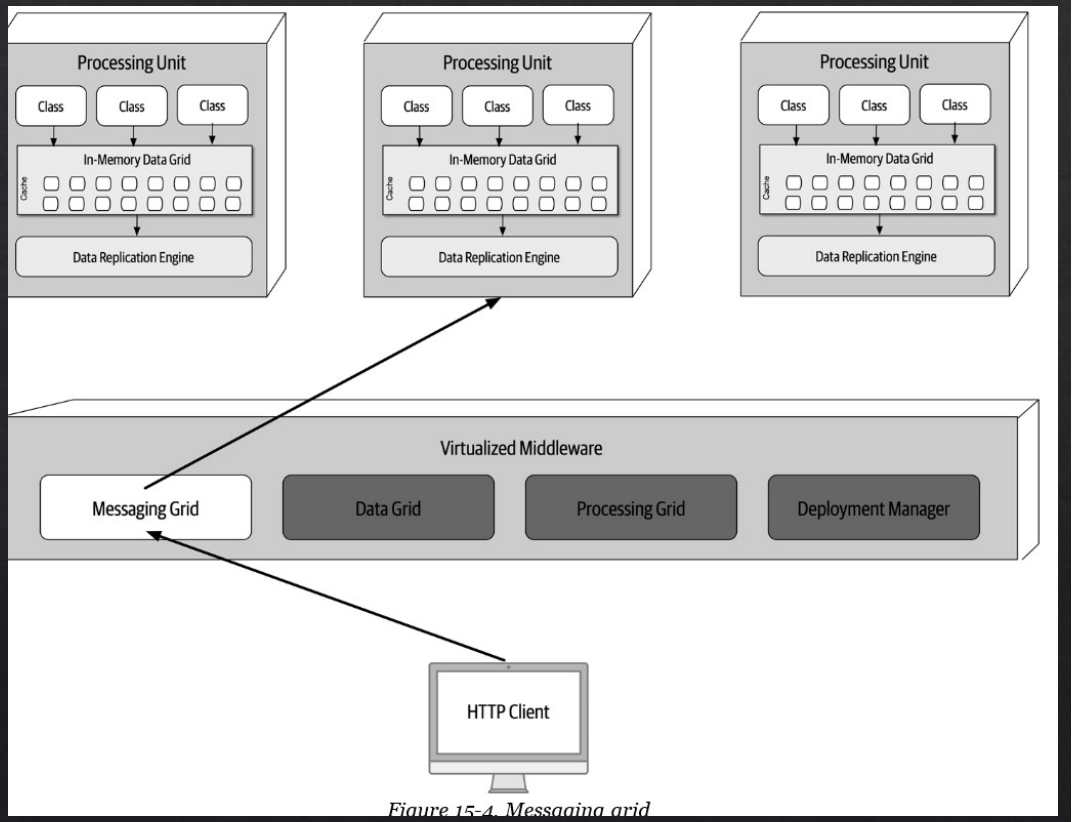


Figure 15-4. Messaging grid

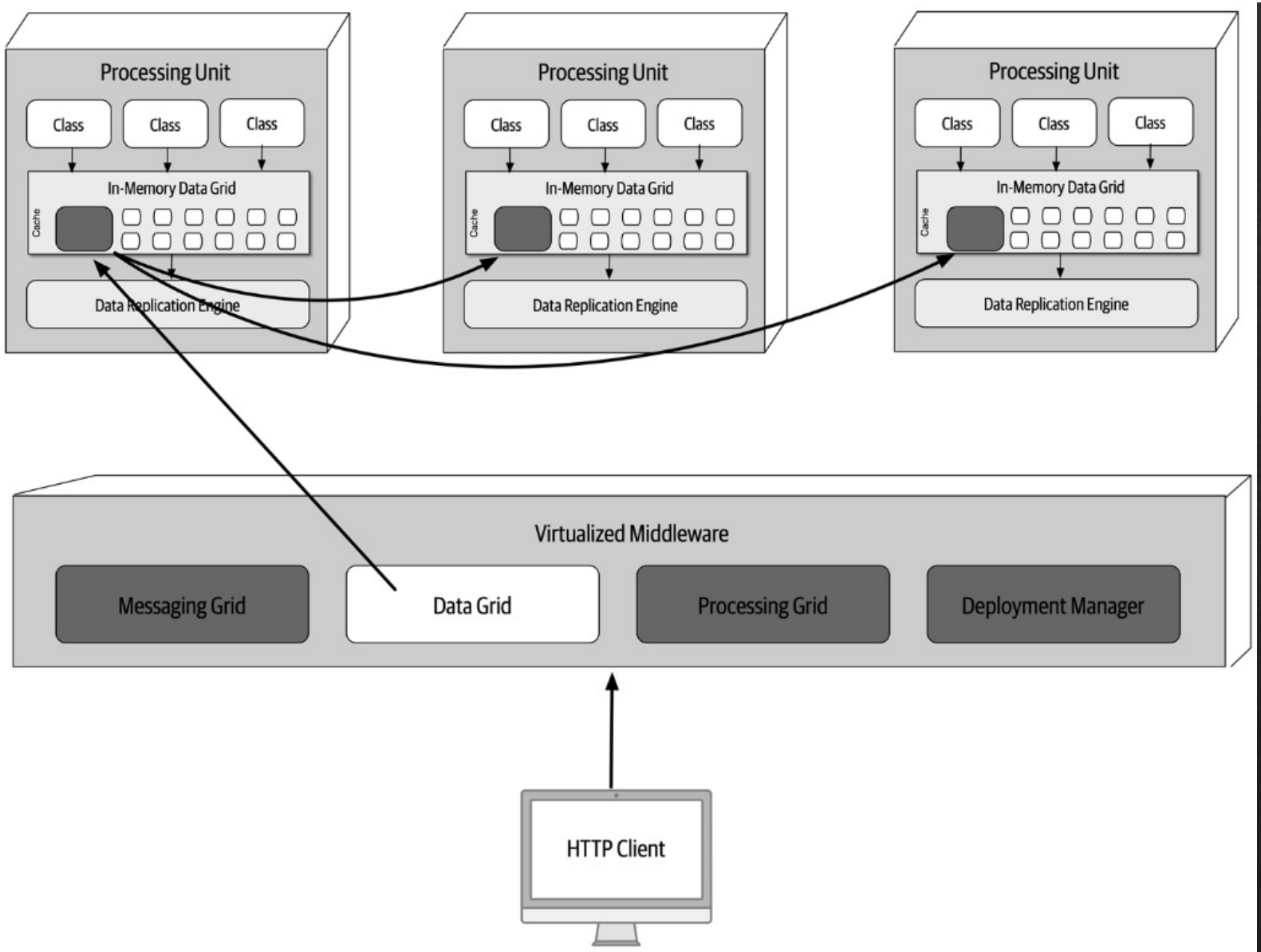


Figure 15-5. Data grid

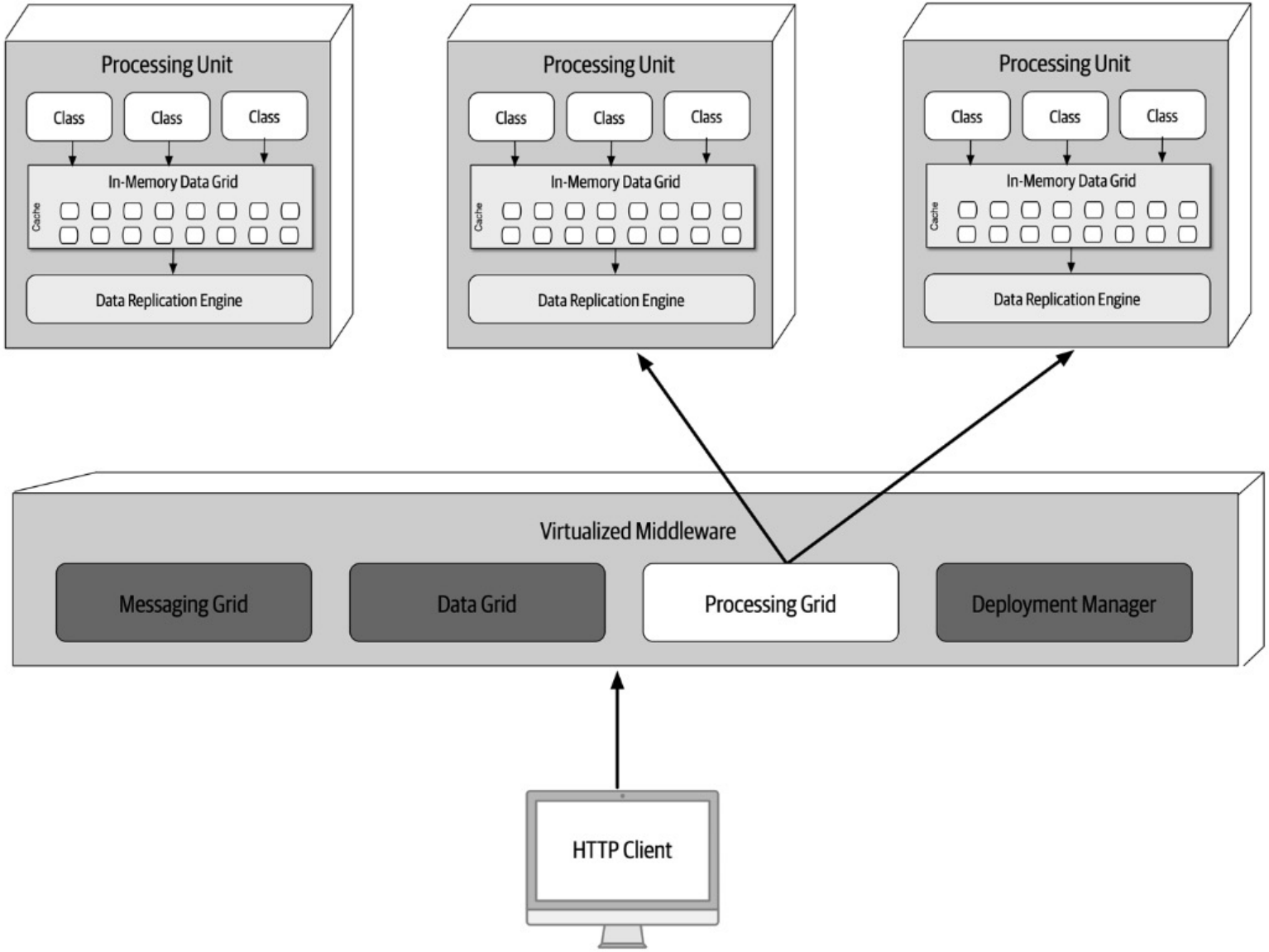


Figure 15-6. Processing grid

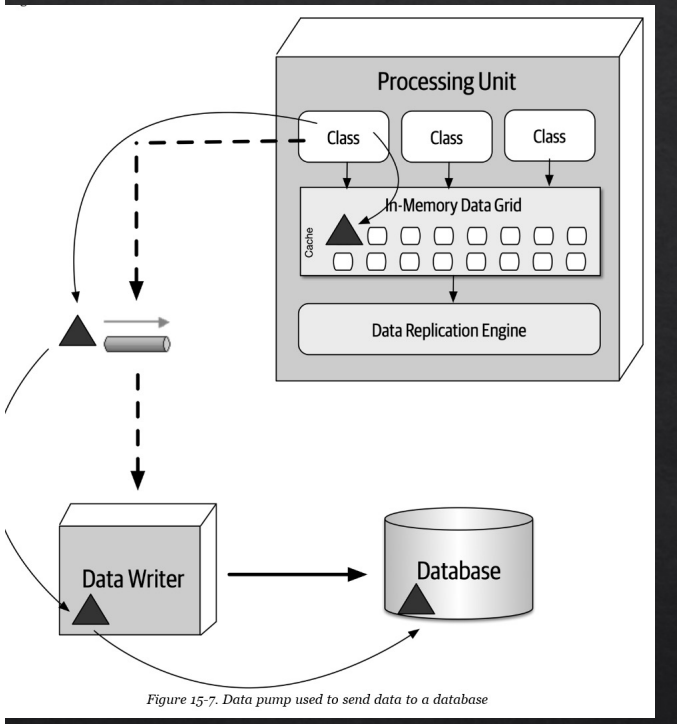


Figure 15-7. Data pump used to send data to a database

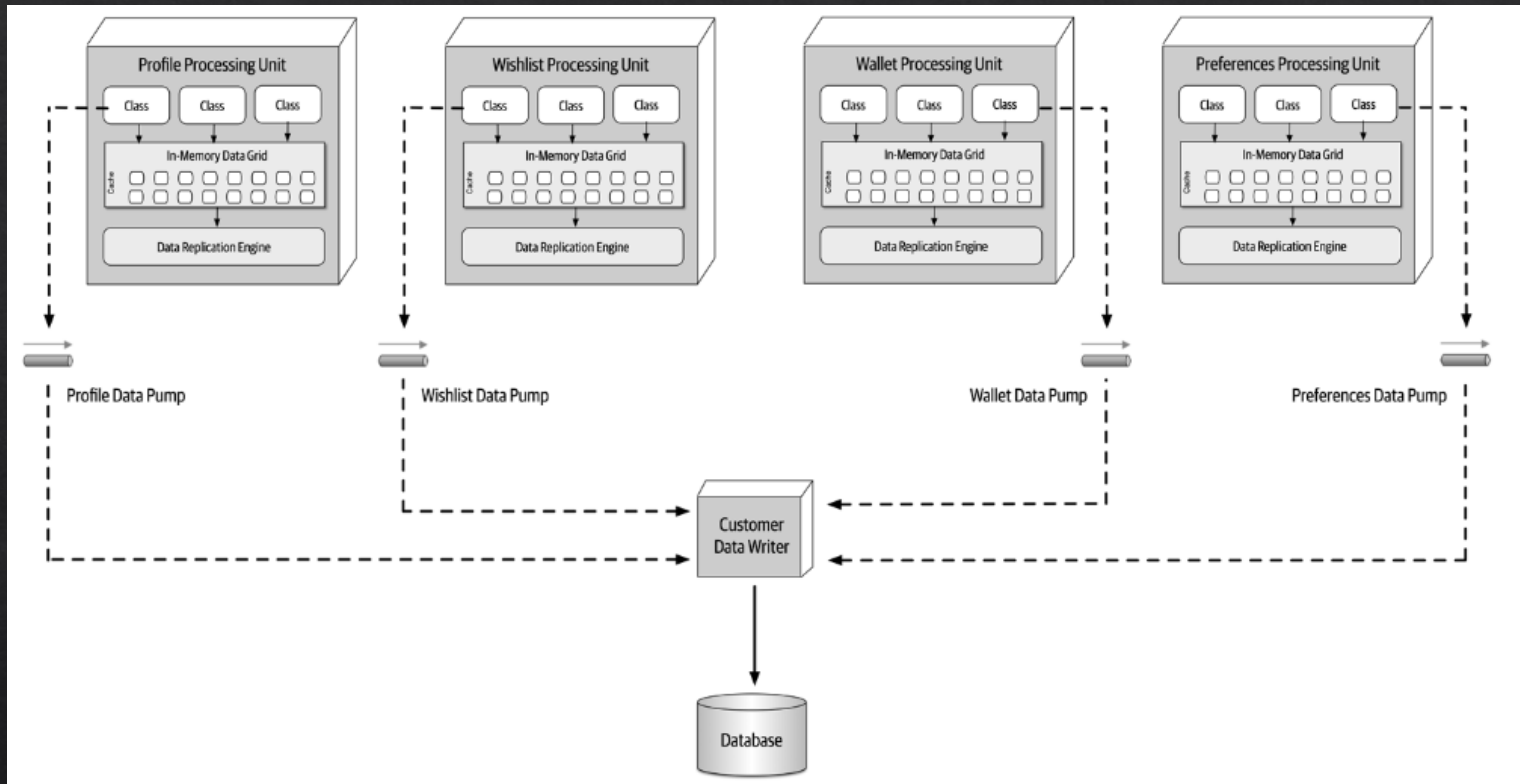


Figure 15-8. Domain-based data writer

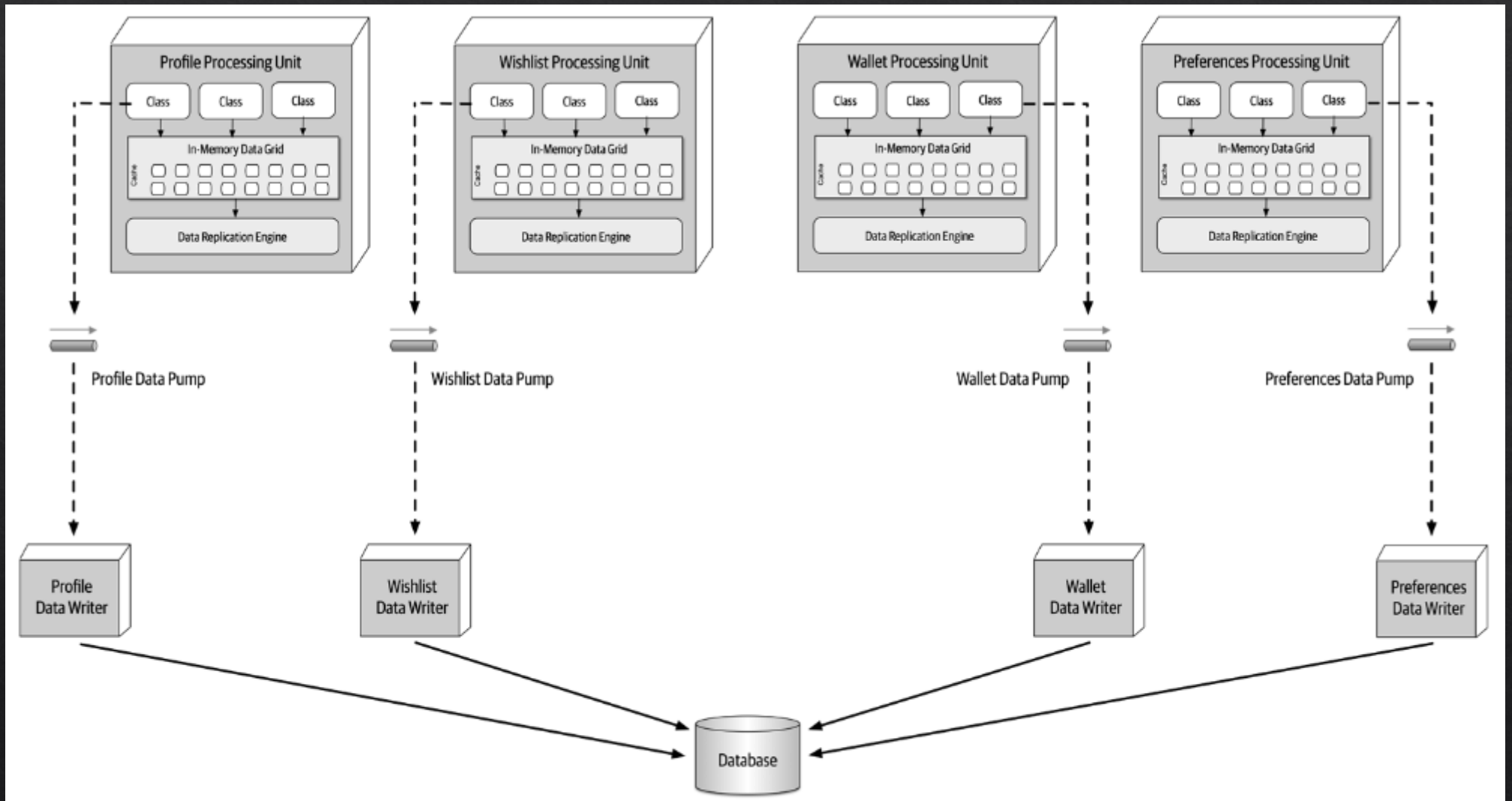


Figure 15-9. Dedicated data writers for each data pump

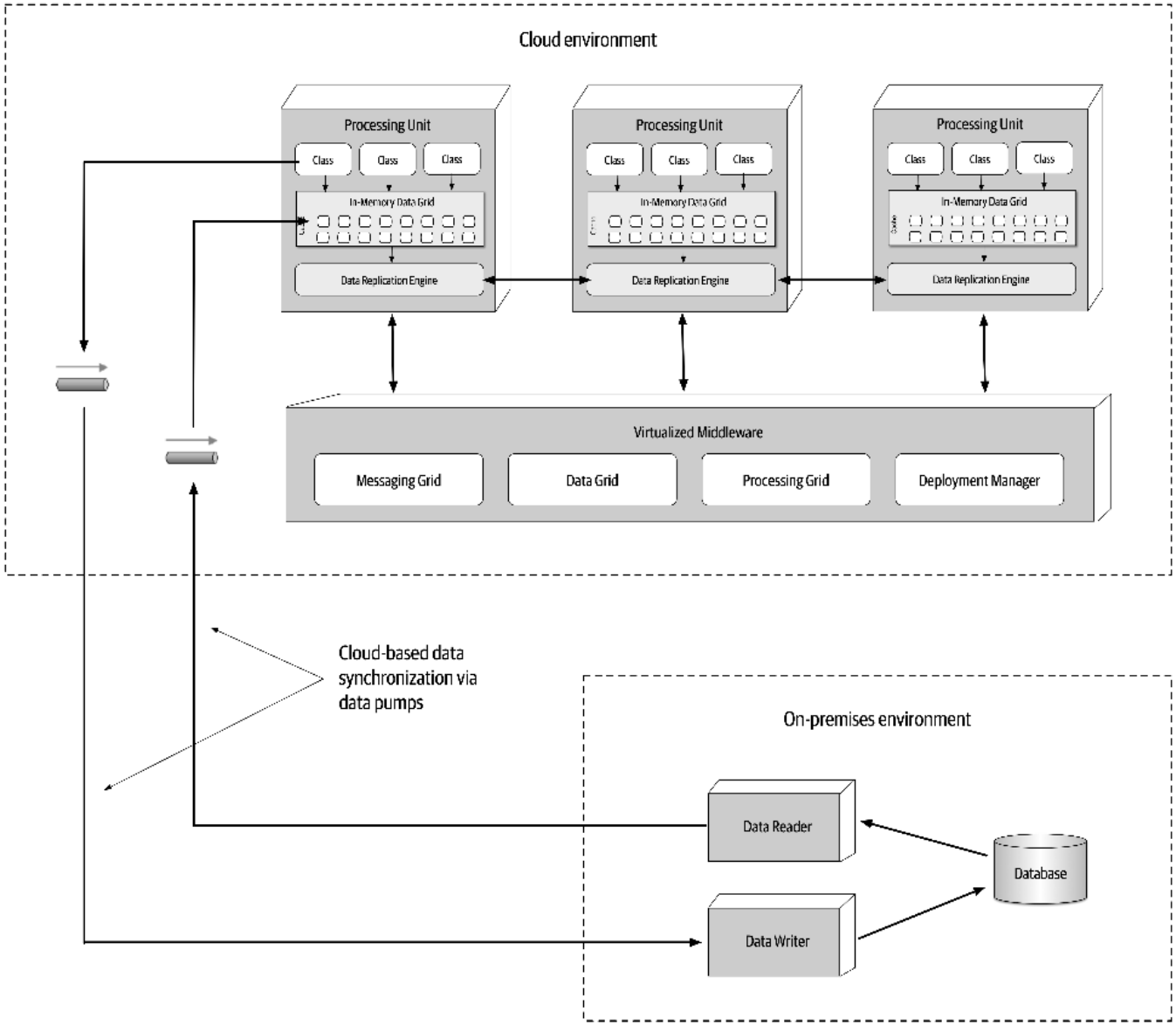


Figure 15-11. Hybrid cloud-based and on-prem topology

Rating

- Example systems: concert ticketing, online auction
- Elasticity, scalability, performance
- Reliability
- Bad: complex, hard to test

Architecture characteristic	Star rating
Partitioning type	Domain and technical
Number of quanta	1 to many
Deployability	★★★
Elasticity	★★★★★
Evolutionary	★★★
Fault tolerance	★★★
Modularity	★★★
Overall cost	★★
Performance	★★★★★
Reliability	★★★★
Scalability	★★★★★
Simplicity	★
Testability	★

Figure 15-15. Space-based architecture characteristics ratings

Orchestration-driven Service-Oriented Architecture (SOA)

- Reuse!!!!
- Late 1990's
- OS, DBMS were expensive

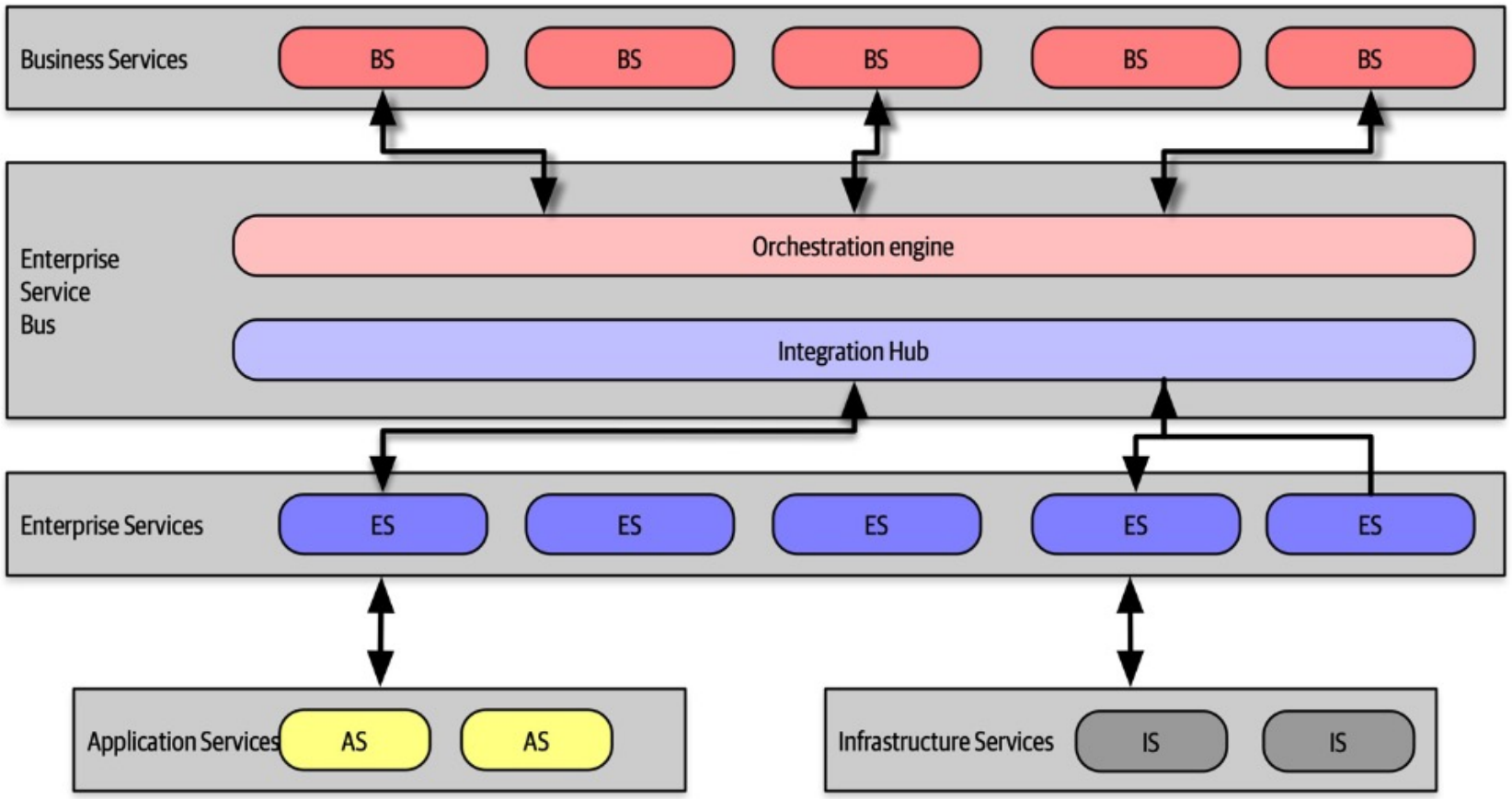


Figure 16-1. Topology of orchestration-driven service-oriented architecture

SOA components

- Business services: ‘PlaceOrder’, ‘ExecuteOrder’; coarse-grained services
  - “are we in the business of …?”, no code, input, output, schema
- Enterprise services: shared implementation, ‘CreateCustomer’ ‘CalculateQuote’; building blocks that makes up BS; fine-grained services
- Application services: one-off service to a domain; e.g. geolocation
- Infrastructure services: operational functions; monitoring, logging, authentication …
- Orchestration engine: coordinates transactions, message transformation, defines relationship between BS and ES
- Difficult to choose boundaries

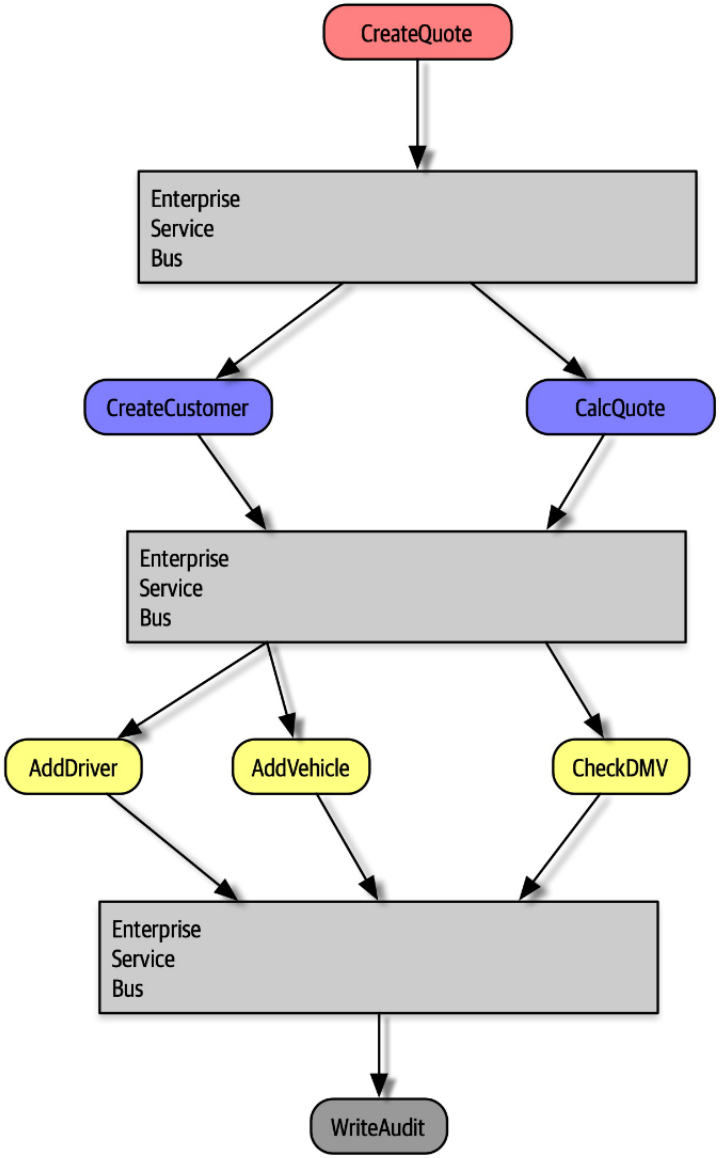


Figure 16-2. Message flow with service-oriented architecture

Reuse

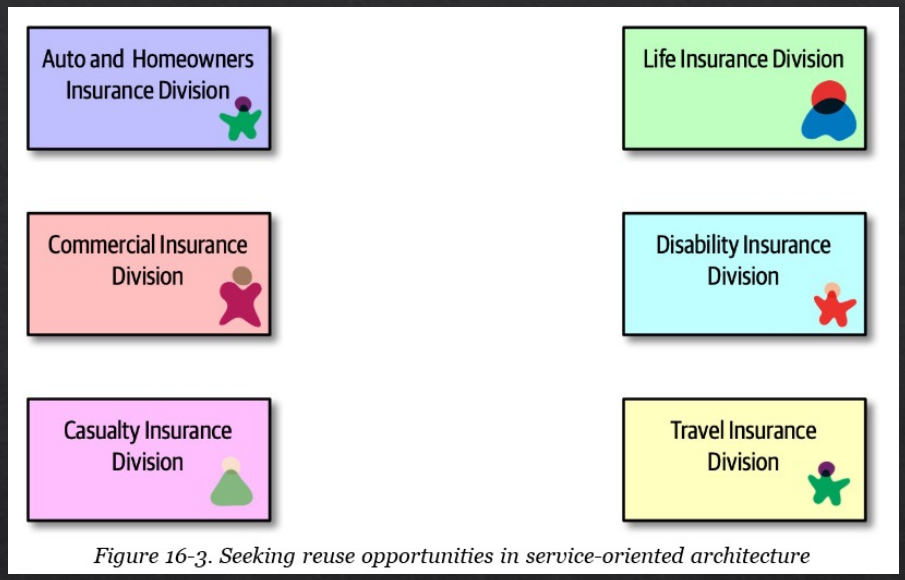


Figure 16-3. Seeking reuse opportunities in service-oriented architecture

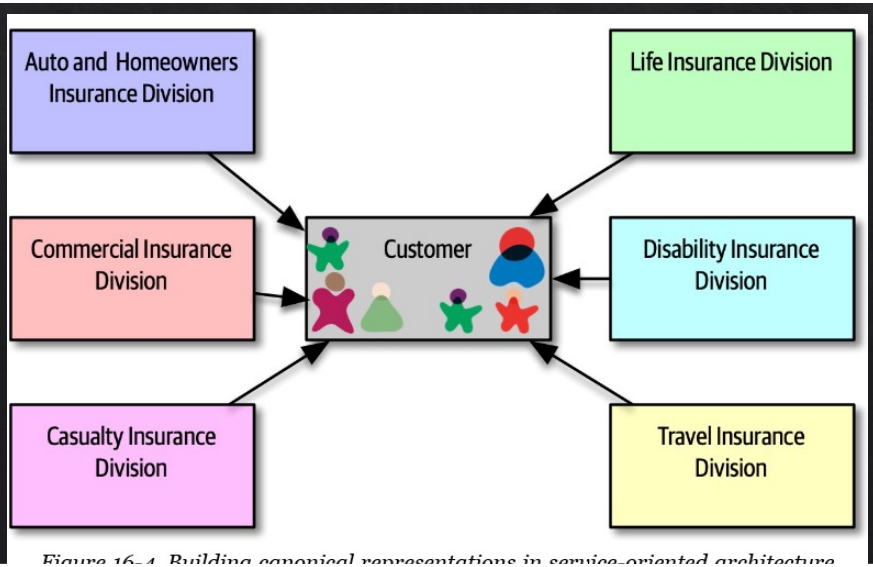


Figure 16-4. Building canonical representations in service-oriented architecture

SOA ratings

- Good: scalability
- Bad: complex, hard to test, cost

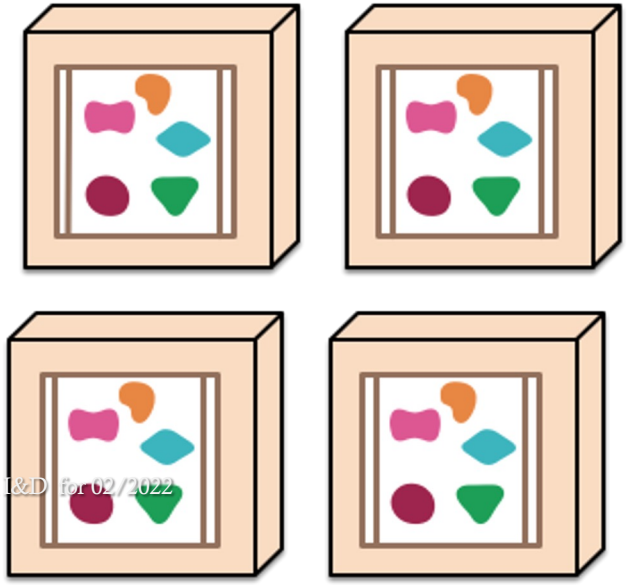
Architecture characteristic	Star rating
Partitioning type	Technical
Number of quanta	1
Deployability	★
Elasticity	★★★
Evolutionary	★
Fault tolerance	★★★
Modularity	★★★
Overall cost	★
Performance	★★
Reliability	★★
Scalability	★★★★
Simplicity	★
Testability	★

Figure 16-5. Ratings for service-oriented architecture

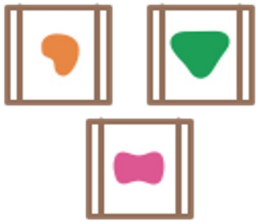
Microservices architecture

A monolithic application puts all its functionality into a single process...

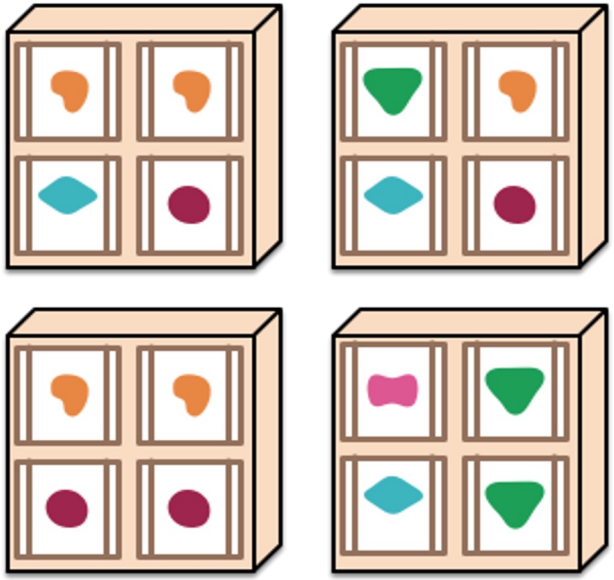
... and scales by replicating the monolith on multiple servers



A microservices architecture puts each element of functionality into a separate service...



... and scales by distributing these services across servers, replicating as needed.



Microservices architecture

- Distributed services
- Granularity: too small >> many communication links
  - Purpose: domain
  - Want low Transactions across services
  - Choreography: may need to combine services
- Data isolation

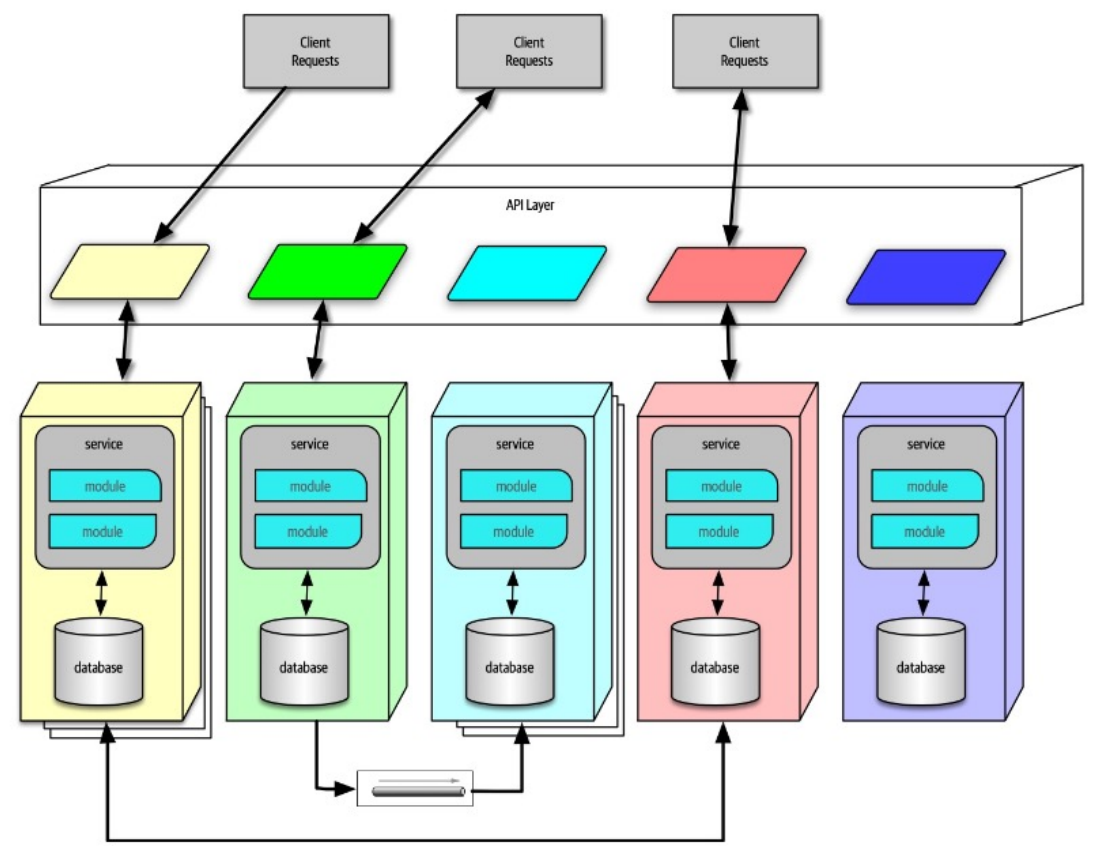


Figure 17-1. The topology of the microservices architecture style

Reuse: sidecar pattern

- Same functions in multiple services; “side-car” must be dev/maintained together

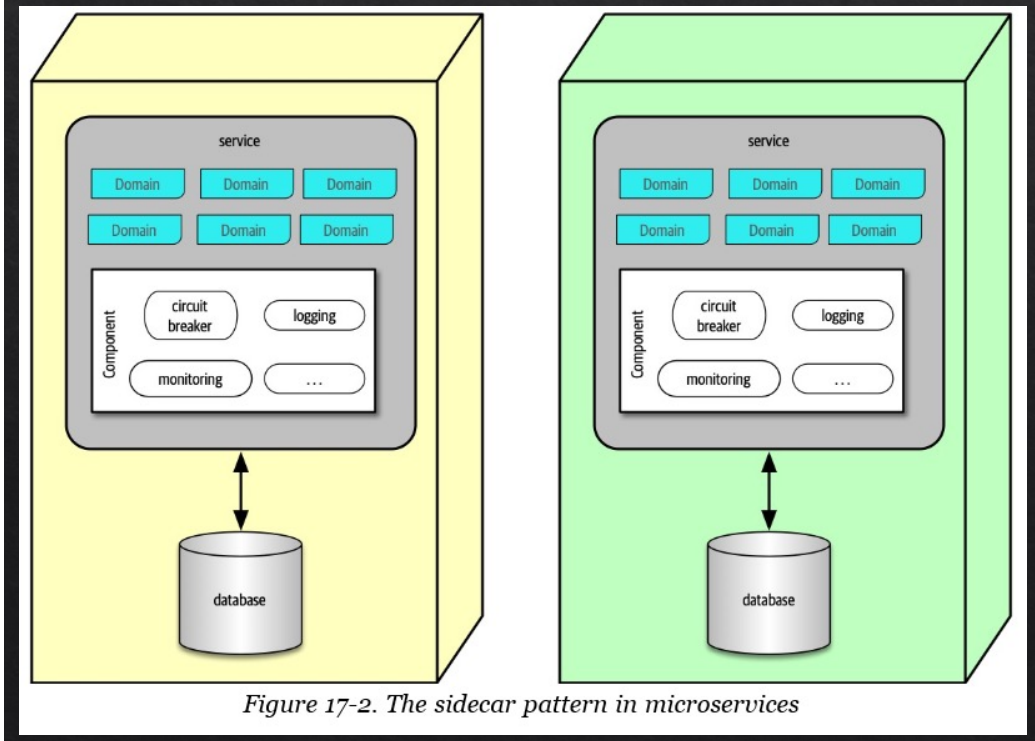


Figure 17-2. The sidecar pattern in microservices

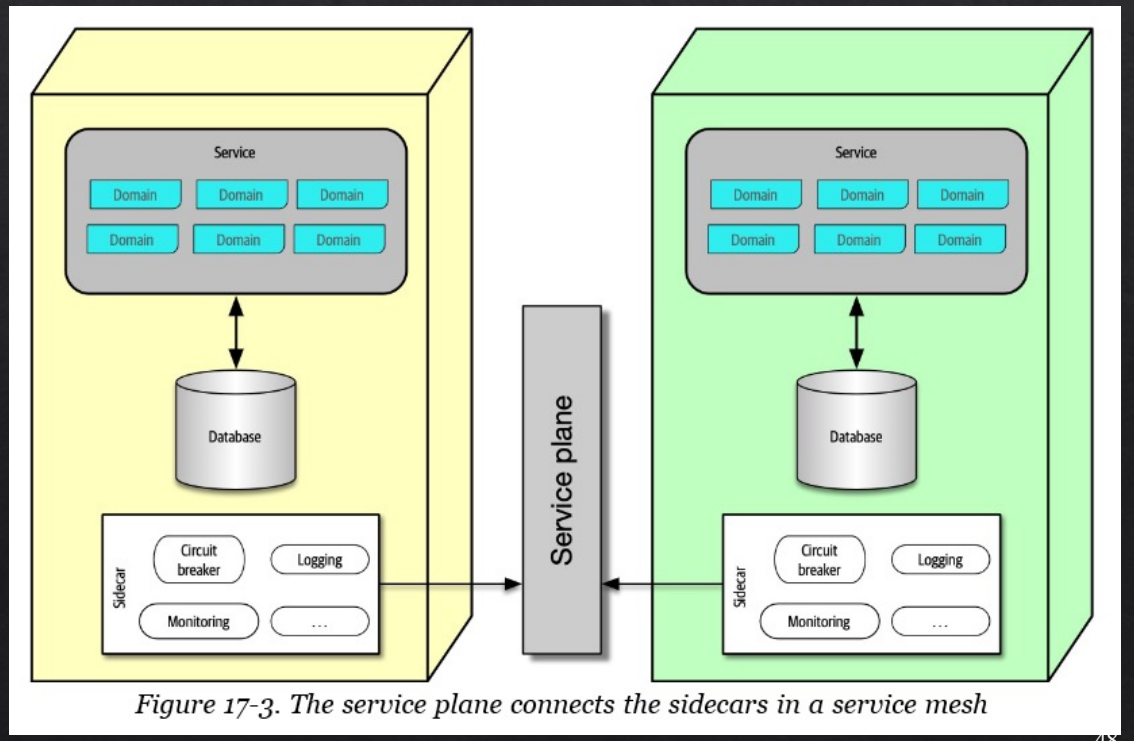


Figure 17-3. The service plane connects the sidecars in a service mesh

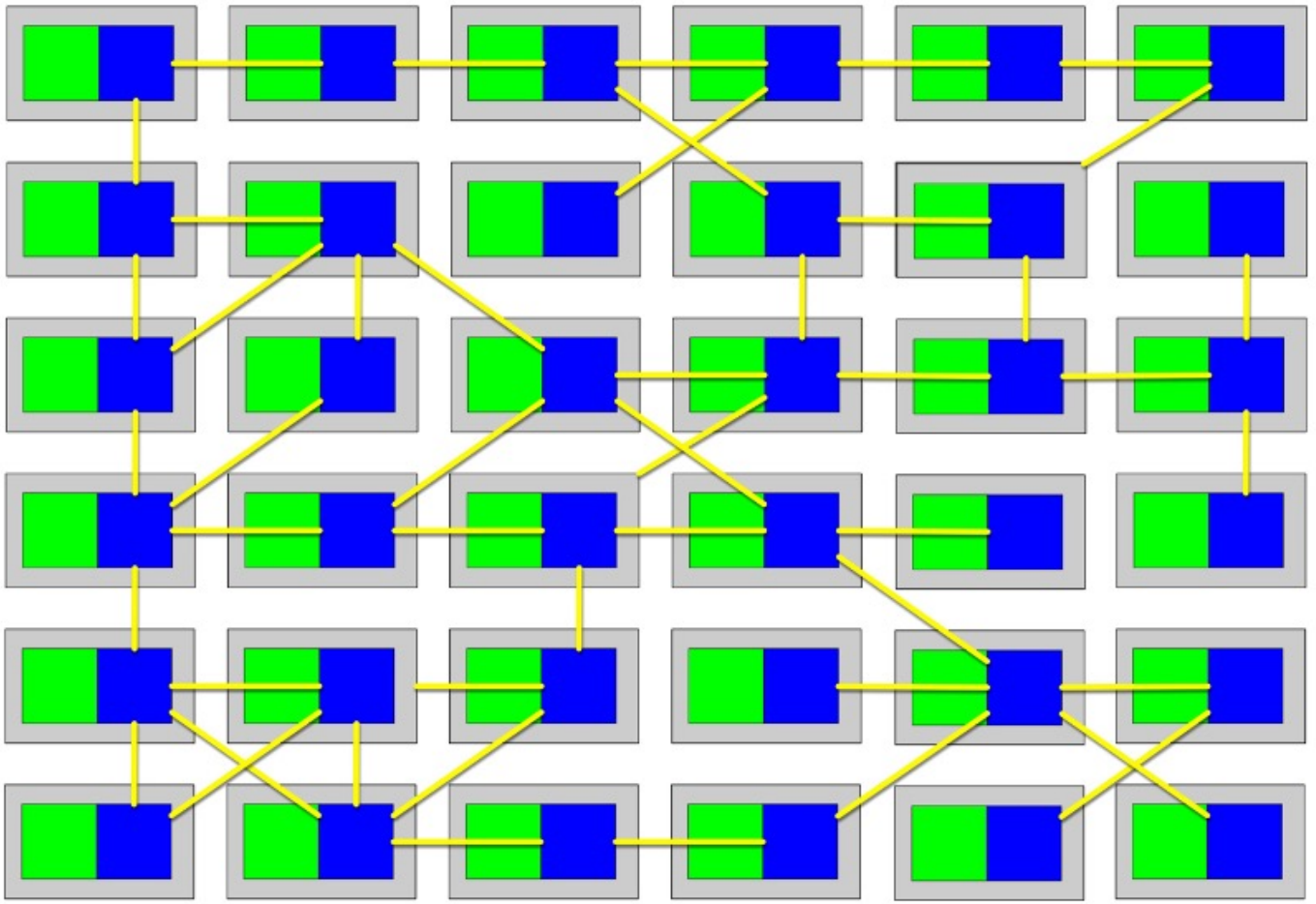


Figure 17-4. The service mesh forms a holistic view of the operational aspect of micro-services

Frontend, micro frontend

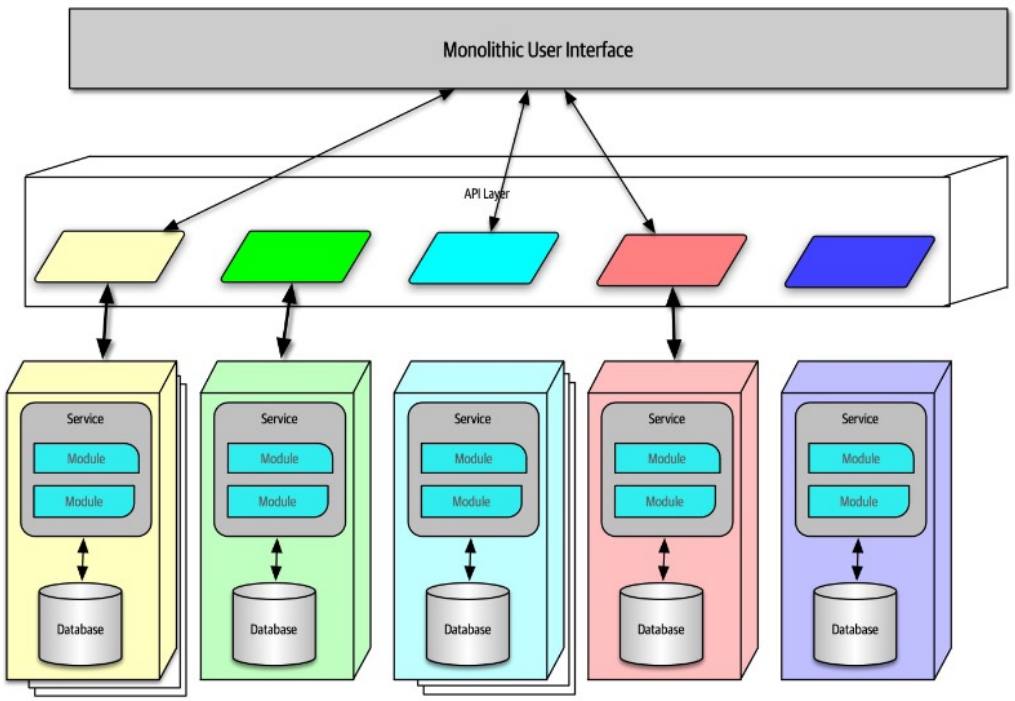


Figure 17-5. Microservices architecture with a monolithic user interface

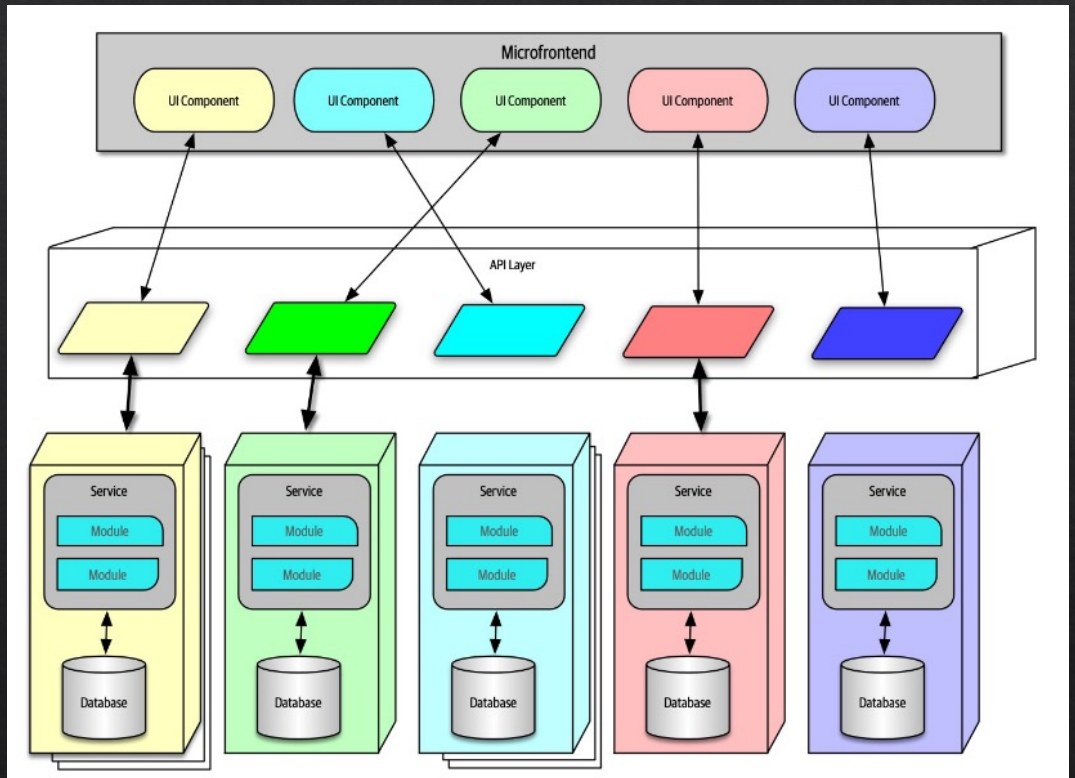


Figure 17-6. Microfrontend pattern in microservices

Choreography vs orchestration

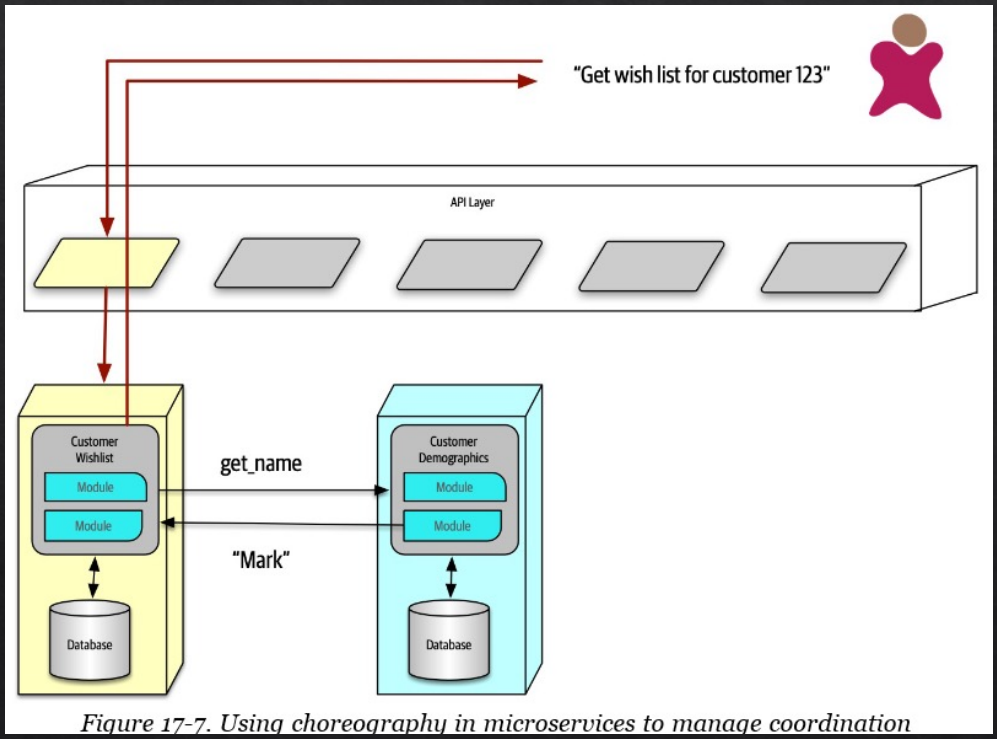


Figure 17-7. Using choreography in microservices to manage coordination

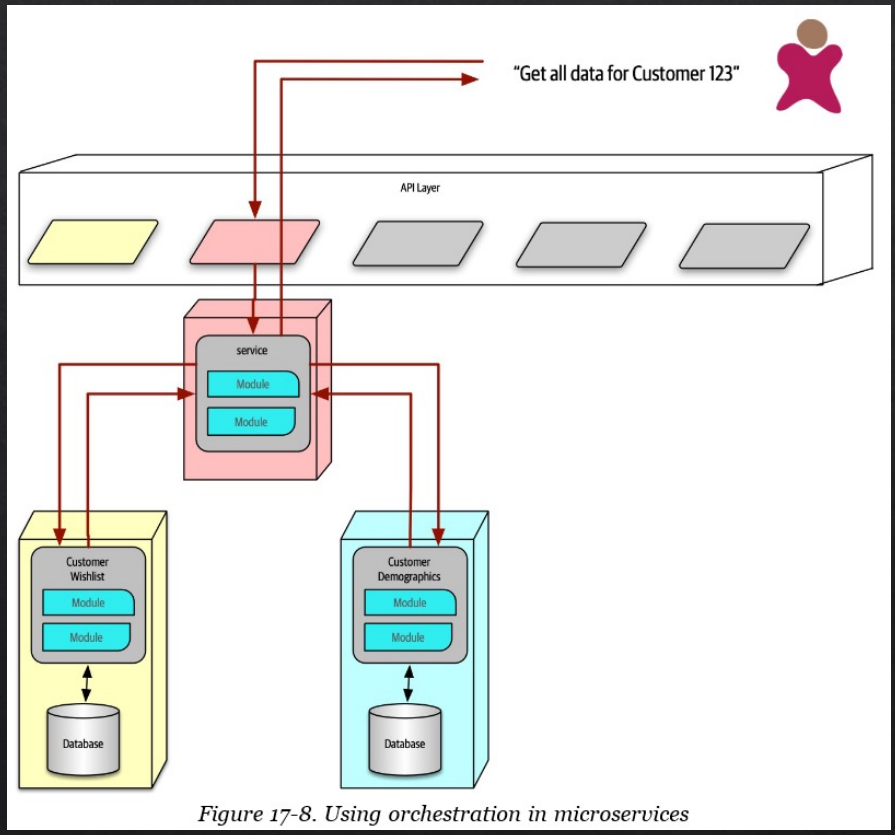


Figure 17-8. Using orchestration in microservices

Choreography vs orchestration

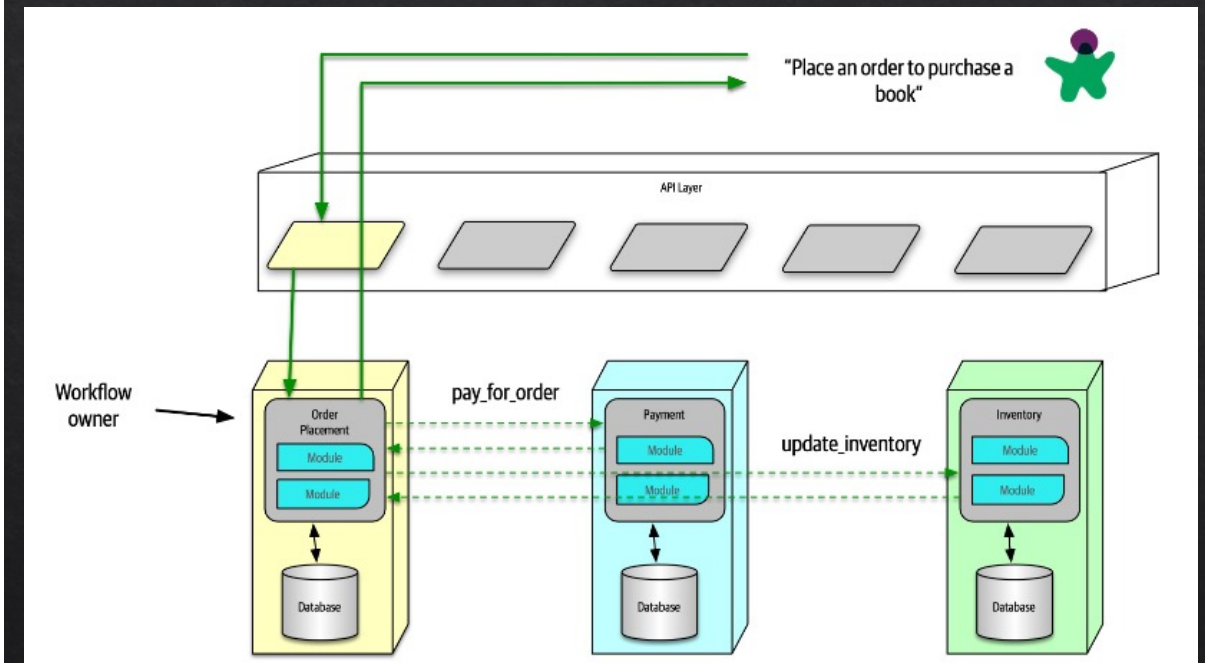


Figure 17-9. Using choreography for a complex business process

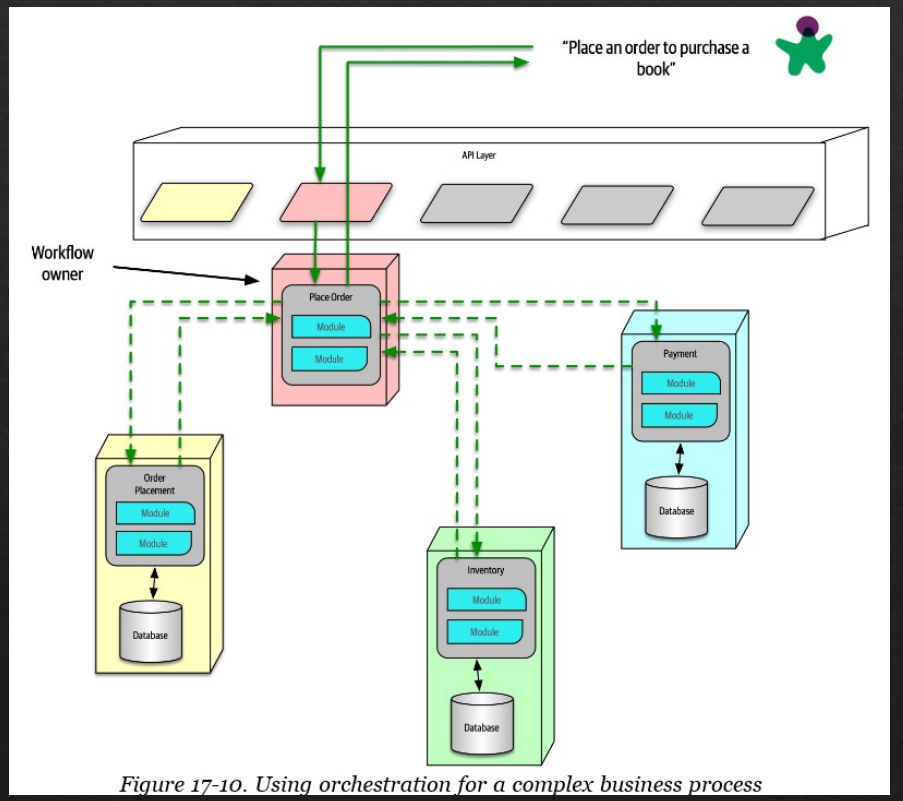


Figure 17-10. Using orchestration for a complex business process

TIP

Don’t do transactions in microservices—fix granularity instead!

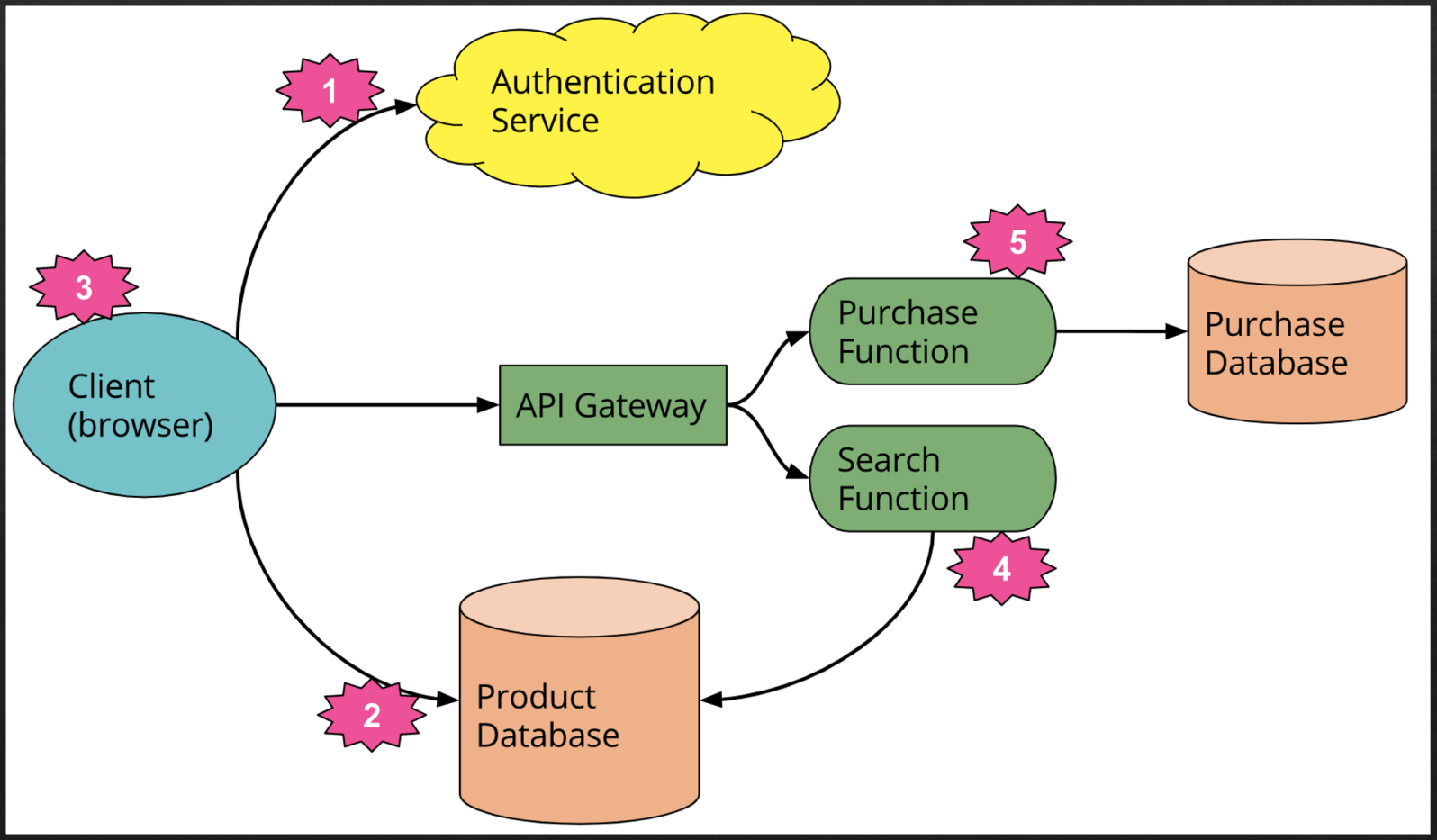
Rating

- Elasticity, evolutionary
- Bad: complex, cost

Architecture characteristic	Star rating
Partitioning type	Domain
Number of quanta	1 to many
Deployability	★★★★
Elasticity	★★★★★
Evolutionary	★★★★★
Fault tolerance	★★★★
Modularity	★★★★★
Overall cost	★
Performance	★★
Reliability	★★★★
Scalability	★★★★★
Simplicity	★
Testability	★★★★

Figure 17-13. Ratings for microservices

Serverless Architecture



https://martinfowler.com/articles/serverless.html

https://thenewstack.io/serverless-architecture-five-design-patterns/?fbclid=IwAR0lM91S1mPPSM2y_Di06maTiVoWkBh395BPyCaLFij2Ih92W94HgP6TnoA