CHƯƠNG 2

KIẾN TRÚC PHẦN MỀM
(SOFTWARE ARCHITECTURE – SA)

Chương 2: Software Architecture

NỘI DUNG

1. Kiến trúc phần mềm là gì?
2. Architecture Styles
3. Architecture Patterns

Kiến trúc phần mềm là gì? (Software Architecture)

- **Software architecture** is the set of structures needed to reason about a software system and the discipline of creating such structures and systems. Each structure comprises software elements, relations among them, and properties of both elements and relations.
- **Kiến trúc phần mềm** là 1 tập cấu trúc cơ bản của một hệ thống phần mềm và quy tắc của việc tạo ra những cấu trúc và hệ thống. Mỗi cấu trúc bao gồm sự sắp xếp của các yếu tố phần mềm, mối quan hệ giữa các yếu tố, và tính chất của các yếu tố đó.

Architecture Is a Set of Software Structures

- Cấu trúc (structure) là một tập hợp các thành phần (elements) được kết nối với nhau bằng một mối quan hệ (relation)
- Hệ thống phần mềm bao gồm nhiều cấu trúc và không có cấu trúc đơn lẻ nào có thể được coi là kiến trúc.
- Có 3 loại cấu trúc quan trọng:
  1. Module (Mô-đun)
  2. Component and Connector (Thành phần và bộ kết nối)
  3. Allocation (Phân bổ)

Module Structures

- Module, hay còn gọi là "mô-đun," là các thành phần nhỏ gọn tạo thành một phần của một tổng thể, với mục tiêu xây dựng một hệ thống tổng thể:
  - có tính thống nhất và tích hợp
  - có khả năng thực hiện nhiều nhiệm vụ và chức năng khác nhau.
  - ➢ Mỗi mô-đun có nhiệm vụ cùng chức năng riêng biệt, đóng góp vào sự vận hành và phát triển tổng thể của hệ thống.
- Các mô-đun được giao trách nhiệm tính toán cụ thể và là cơ sở cho các nhiệm vụ công việc cho các nhóm lập trình (programming teams).
- Trong các dự án lớn, các thành phần này (mô-đun) được chia nhỏ để giao cho các nhóm nhỏ (sub-teams).

Component and connector Structures (C&C)

- Các cấu trúc tập trung vào cách các thành phần tương tác (elements interact) với nhau tại thời gian chạy (runtime) để thực hiện các chức năng của hệ thống → các cấu trúc này gọi là cấu trúc thành phần và kết nối (C&C).
- Một thành phần (component) luôn là một thực thể thời gian chạy (runtime entity)
  - Giả sử hệ thống được xây dựng như một tập hợp các dịch vụ (services). Các dịch vụ (services), cơ sở hạ tầng (infrastructure) mà chúng tương tác, các mối quan hệ đồng bộ hóa (synchronization) và tương tác giữa chúng tạo thành một loại cấu trúc, thường được sử dụng để mô tả một hệ thống.
  - Các dịch vụ này được tạo thành từ biên dịch (compiled) các chương trình trong các đơn vị triển khai (implementation units) khác nhau – modules.

Allocation Structures

- Cấu trúc phân bổ (allocation) mô tả quá trình ánh xạ (mapping) từ cấu trúc phần mềm (software structures) đến môi trường của hệ thống (system's environment)
  - Organizational: tổ chức
  - Developmenttal: phát triển
  - Installation: cài đặt
  - Execution: thực thi
- Ví dụ
  - Các mô-đun được giao cho các nhóm (team) để phát triển (develop) và được gán cho các vị trí trong file structure để triển khai, tích hợp và thử nghiệm.
  - Các component được triển khai (deployed) vào phần cứng (hardware) để thực thi.

Architecture is an Abstraction

- Kiến trúc bao gồm các thành phần phần mềm và cách các thành phần liên quan đến nhau.
- Kiến trúc chọn một số chi tiết nhất định và loại bỏ những chi tiết khác.
- Chi tiết riêng tư của các thành phần - chi tiết chỉ liên quan đến việc triển khai nội bộ - không phải là kiến trúc.
- Sự trừu tượng về mặt kiến trúc cho phép xem xét hệ thống theo các yếu tố của thành phần phần mềm và cách các thành phần liên quan đến nhau:
  - cách được sắp xếp
  - cách tương tác
  - cách được cấu thành
  - các đặc tính hỗ trợ cho hệ thống, v.v.
- Sự trừu tượng này rất cần thiết cho một kiến trúc phức tạp.

Mỗi hệ thống nên có 1 Software Architecture

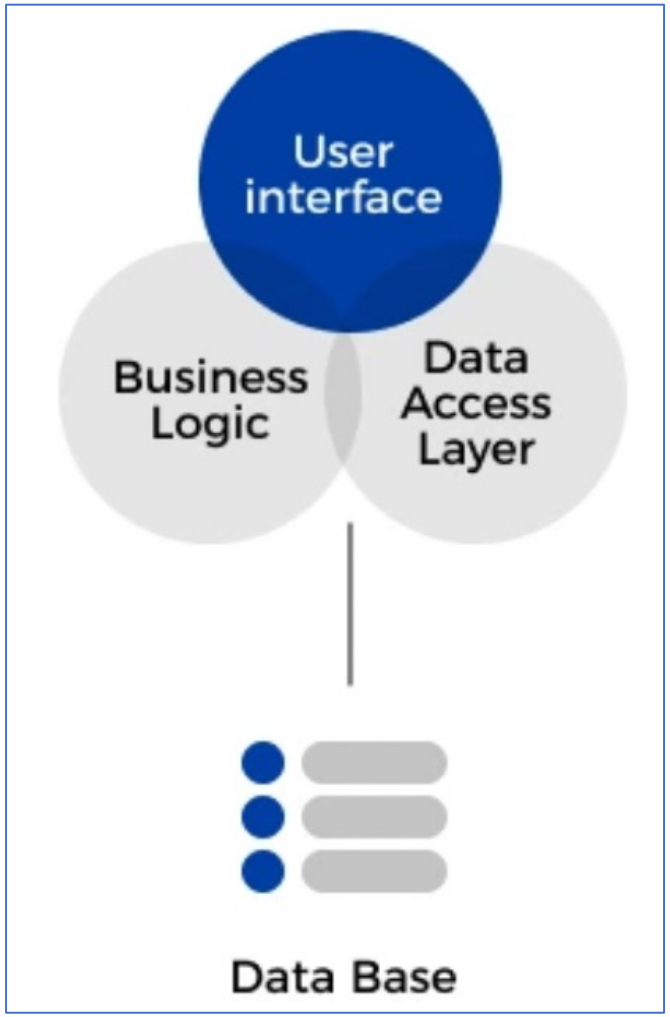
- Mỗi hệ thống nên có tài liệu về Kiến trúc của nó. Vì kiến trúc bao gồm các thành phần và mối quan hệ giữa chúng.
- Nhưng có 1 vài hệ thống không có hoặc không ai biết kiến trúc đó:
  - Có thể tất cả những người thiết kế hệ thống hiện tại không còn làm hệ thống này.
  - Có thể tài liệu đã mất (hoặc không bao giờ được tạo ra)
  - Có thể mã nguồn đã bị mất (hoặc không bao giờ được cung cấp)
- Một kiến trúc có thể tồn tại độc lập với mô tả hoặc thông số kỹ thuật của nó.
- Tài liệu rất quan trọng.

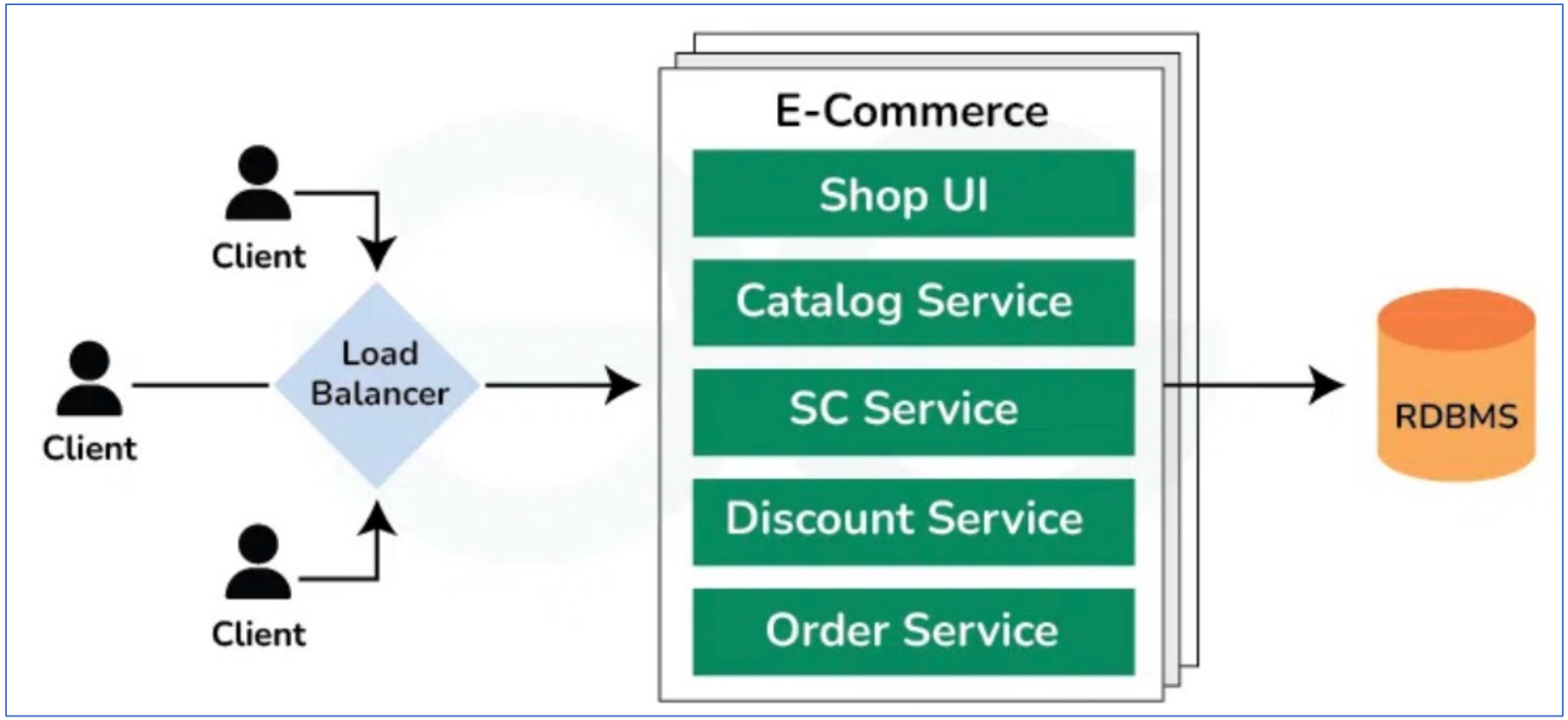
Architecture Styles

- Monolithic Architecture – Kiến trúc nguyên khối
  - Layered (n-tier) Architecture
  - Client-Server Architecture
  - Pipeline Architecture
  - …
- Distributed Architecture – Kiến trúc phân tán
  - Event-driven Architecture
  - Space-based Architecture
  - Service-oriented Architecture
  - Microservice Architecture
  - …

Architecture Styles

Monolithic Architecture – Kiến trúc đơn khối





Architecture Styles

Monolithic Architecture – Kiến trúc đơn khối (tt)

- Kiến trúc đơn (nguyên) khối là mô hình truyền thống của chương trình phần mềm, được xây dựng như một đơn vị thống nhất, độc lập và không phụ thuộc vào các ứng dụng khác.
- Là một khối mã lớn với nhiều mô-đun, được liên kết chặt chẽ với nhau
- Mỗi thành phần (component) của chương trình, bao gồm:
  - Lớp truy cập dữ liệu
  - Logic nghiệp vụ
  - Giao diện người dùng

  ➢ đều được triển khai và tích hợp chặt chẽ với nhau trong thiết kế này.
- Kiến trúc này có thể thuận tiện ngay từ đầu trong vòng đời của dự án để dễ dàng quản lý mã, chi phí nhận thức và triển khai. Điều này cho phép mọi thứ trong đơn khối được phát hành cùng một lúc.

Architecture Styles

Monolithic Architecture – Ưu điểm

- Design: thiết kế đơn giản chỉ 1 code base duy nhất với nhiều function phụ thuộc lẫn nhau
- Development: khi một ứng dụng được xây dựng với một code base, việc phát triển (develop) sẽ dễ dàng hơn.
- Deployment: chỉ thực thi (executable) 1 file hoặc 1 directory nên việc triển khai (deployment) dễ dàng hơn.
- Debugging: tất cả code ở cùng 1 nơi nên việc the dõi yêu cầu (request) và tìm sự cố (issue) dễ dàng hơn
- Testing: Có thể test toàn bộ ứng dụng một cách dễ dàng và tìm kiếm lỗi trong môi trường đồng nhất.
- Performance - Hiệu suất cao trong các ứng dụng ít phức tạp: Các ứng dụng ít phức tạp có thể chạy hiệu quả với kiến trúc monolithic do giảm thiểu giao tiếp mạng và độ trễ.

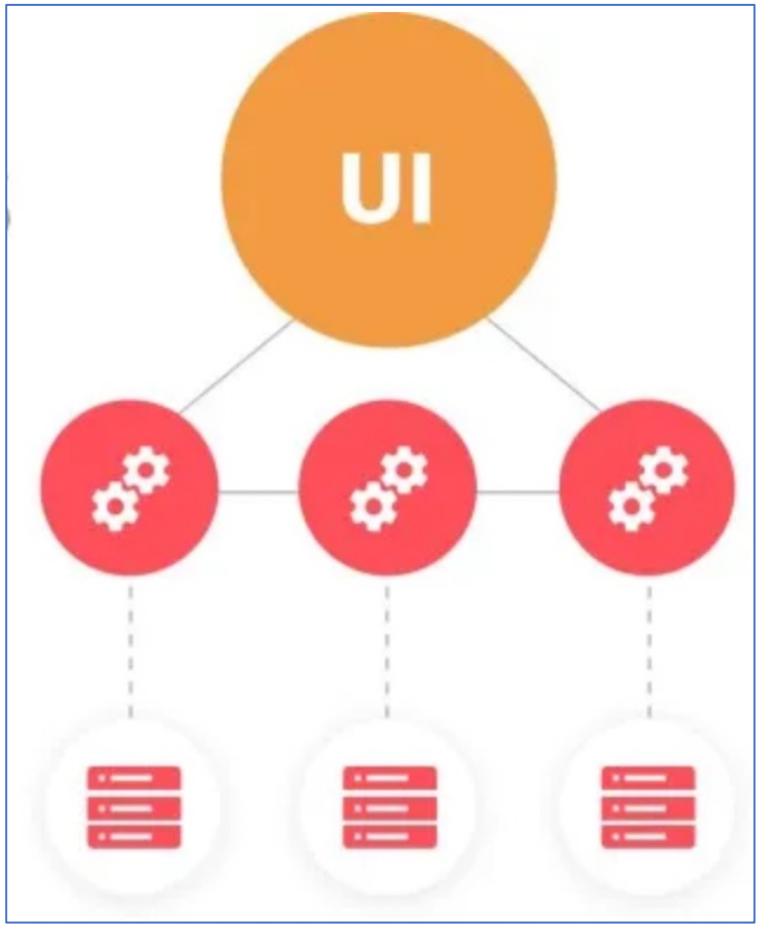
Architecture Styles

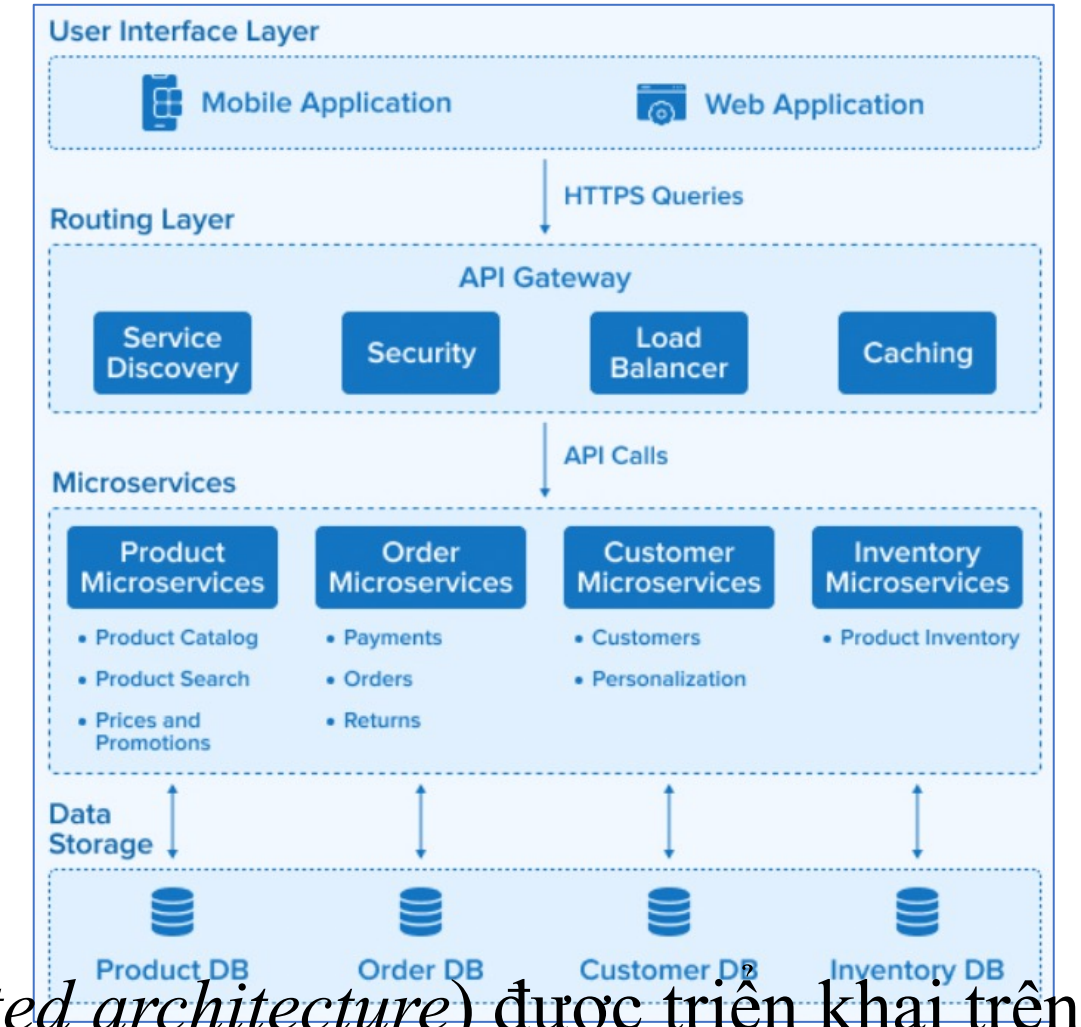
Monolithic Architecture – Khuyết điểm

- Slower development speed: một ứng dụng lớn, nguyên khối khiến quá trình phát triển trở nên phức tạp và chậm hơn.
- Scalability: không thể mở rộng các thành phần riêng lẻ.
- Reliability: nếu có lỗi trong bất kỳ mô-đun nào, lỗi đó có thể ảnh hưởng đến tính khả dụng của toàn bộ ứng dụng.
- Barrier to technology adoption (Rào cản đối với việc áp dụng công nghệ): bất kỳ thay đổi nào trong khuôn khổ hoặc ngôn ngữ đều ảnh hưởng đến toàn bộ ứng dụng, khiến việc thay đổi thường tốn kém và mất thời gian.
- Lack of flexibility (Thiếu tính linh hoạt): bị hạn chế bởi các công nghệ đã được sử dụng trong khối nguyên khối.
- Deployment: một thay đổi nhỏ đối với một ứng dụng nguyên khối đòi hỏi phải triển khai lại toàn bộ hệ thống.

Architecture Styles

Distributed Architecture – Kiến trúc phân tán





- **Kiến trúc phân tán** (*Distributed architecture*) được triển khai trên nhiều đơn vị (multiple deployment units) làm việc cùng nhau để thực hiện một số loại chức năng business.

Architecture Styles

Distributed Architecture – Ưu điểm

- **Khả năng chịu lỗi và tin cậy**: nếu một dịch vụ bị lỗi, các dịch vụ khác có thể tiếp tục thực hiện các yêu cầu dịch vụ.
- **Khả năng thích ứng**: Chức năng của ứng dụng được chia thành các đơn vị phần mềm được triển khai riêng biệt, dễ dàng xác định vị trí và áp dụng thay đổi hơn, phạm vi thử nghiệm được giảm xuống chỉ còn dịch vụ bị ảnh hưởng và rủi ro triển khai giảm đáng kể vì thường chỉ triển khai dịch vụ bị ảnh hưởng.
- **Có khả năng mở rộng:** vì chứa một tập hợp các máy độc lập nên có thể mở rộng theo chiều ngang dễ dàng.
- **Độ trễ thấp:** do có nhiều máy chủ và có khả năng phản hồi đến gần người dùng hơn để giải quyết truy vấn, do đó mất rất ít thời gian để giải quyết truy vấn của người dùng.

Architecture Styles

Distributed Architecture – Khuyết điểm

- **Tăng đáng kể độ phức tạp của kiến trúc**: quản lý nhiều chương trình nhỏ hơn rõ ràng phức tạp hơn so với một ứng dụng monolithic.
- **Yêu cầu phải deploy tự động**.
- **Giảm tổng thể về hiệu suất**: kiến trúc microservices thường dẫn đến mức tiêu tốn tài nguyên cao hơn so với kiến trúc monolithic có cùng quy mô. Các microservice đòi hỏi nhiều bộ nhớ hơn, tốn nhiều chu kỳ CPU hơn và cần băng thông mạng lớn hơn.
- Quản lý các giao dịch phân tán, tính nhất quán cuối cùng, quản lý quy trình làm việc, xử lý lỗi, đồng bộ hóa dữ liệu.

Architecture Styles

Distributed Architecture – Khuyết điểm (tt)

- **Khó khắc phục sự cố và gỡ lỗi**: xác định và giải quyết các vấn đề trong kiến trúc microservices cũng phức tạp tương tự như việc tổng hợp các file theo dõi và log. Khi một yêu cầu thất bại đi qua nhiều microservice được lưu trữ trong các môi trường chạy riêng biệt, việc xác định chính xác vị trí và nguyên nhân của sự cố có thể rất khó khăn.
- **Chi phí cao**:
  - Tăng tổng dung lượng bộ nhớ
  - Tài nguyên bị trùng lặp khi sử dụng nhiều container hoặc máy ảo.
  - Tiêu tốn băng thông khi gọi các dịch vụ web RESTful …

Architecture Styles

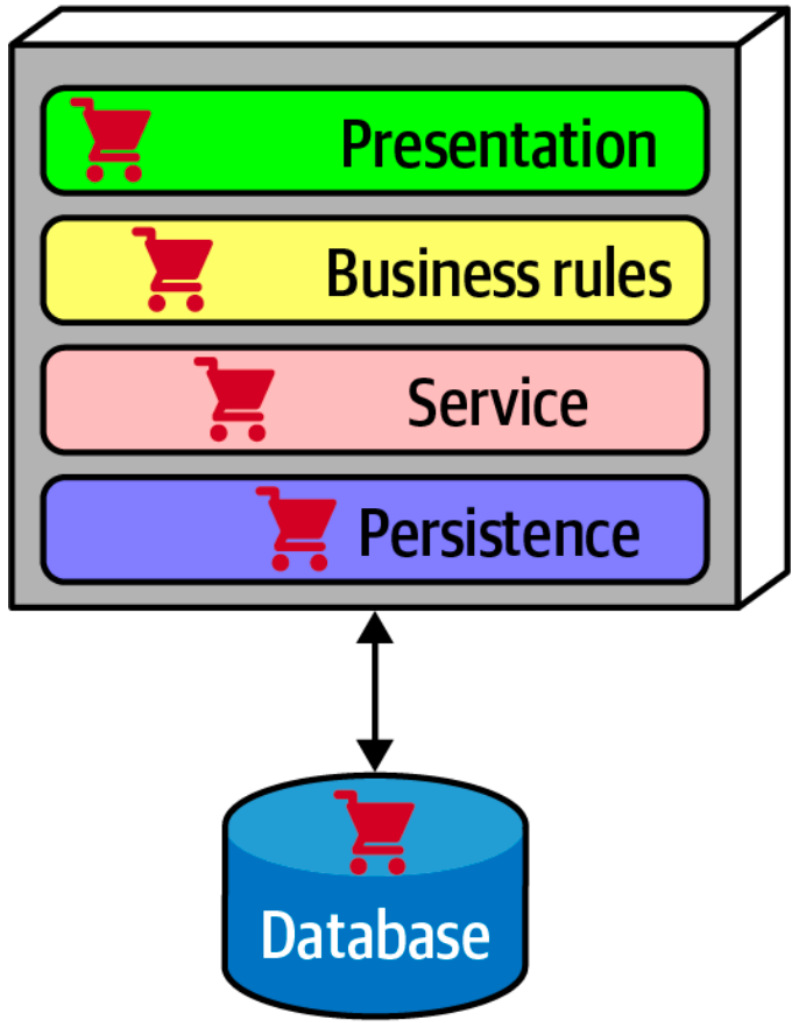
Phân loại khác

Các mẫu kiến trúc phần mềm cũng có thể được phân loại theo cách chia cấu trúc tổng thể của hệ thống. Theo cách chia này, chúng ta có hai loại chính là:

1. Phân vùng kỹ thuật (*Technically Partitioning*)
   - Sắp xếp kiến trúc dựa trên kỹ thuật. Cách tiếp cận này thường biểu hiện dưới dạng kiến trúc Layered, trong đó chức năng của hệ thống được chia thành các lớp kỹ thuật riêng biệt như presentation, business rules, services, và persistence.
   - Ví dụ, mẫu thiết kế Model-View-Controller (MVC) quen thuộc phù hợp với cách tiếp cận này, giúp đơn giản hóa cho developer.
2. Phân vùng theo domain (*Domain Partitioning)*
   - Domain phản ánh chính xác các yêu cầu và quy tắc của nghiệp vụ. Cách tiếp cận này, thường liên quan đến Domain-Driven Design (DDD), nhấn mạnh vào việc mô hình hóa các hệ thống phức tạp dựa trên các domain và tách biệt.

Architecture Styles

Phân loại khác – Technically Partitioning



Trong phân vùng kỹ thuật, các component của hệ thống được tổ chức theo cách sử dụng kỹ thuật (technical usage)

Ưu điểm:

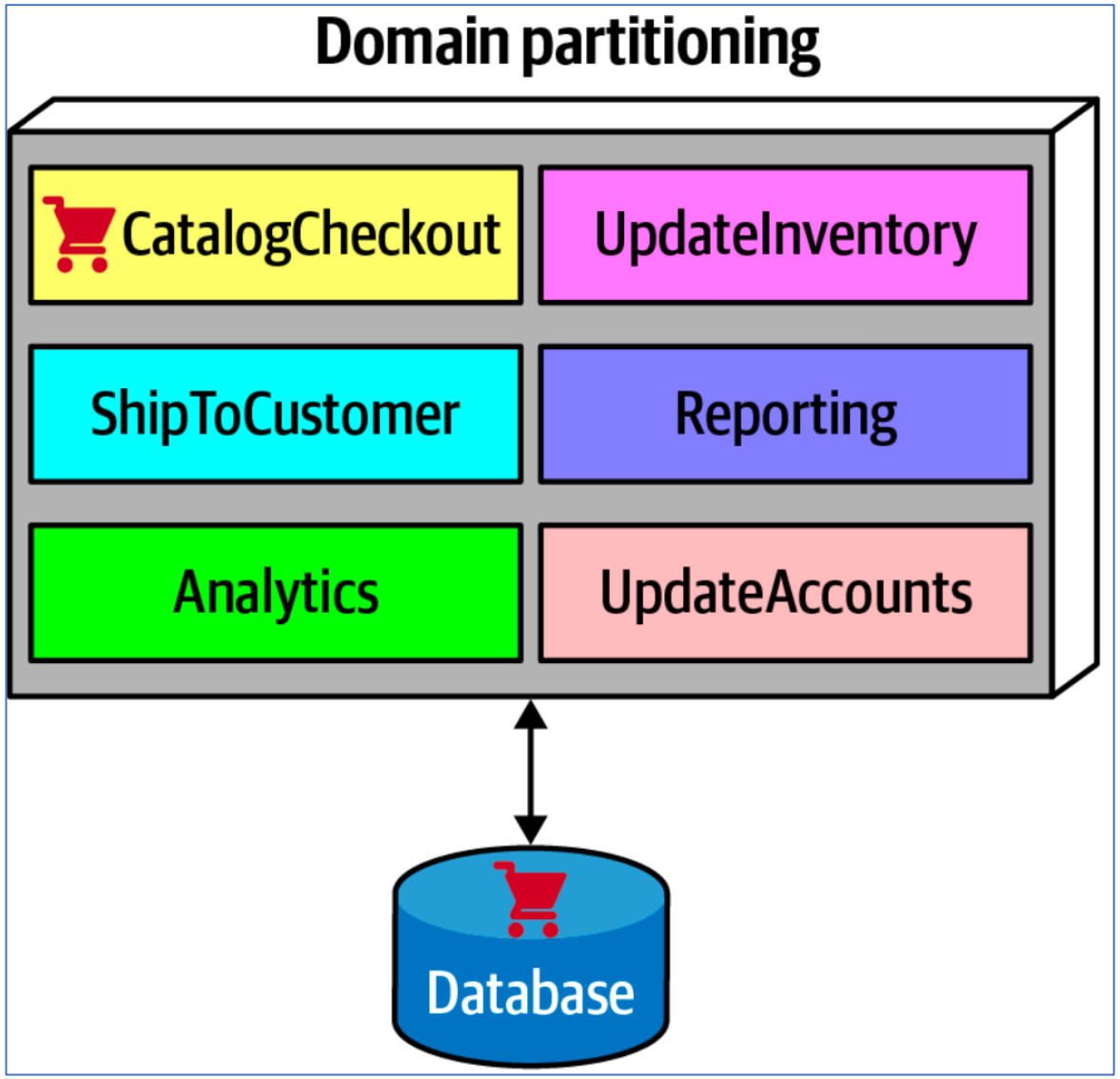
- Phù hợp với các team được tổ chức phân nhóm theo vai trò. VD: nhóm BE, nhóm FE, nhóm designer, ...
- Khi mã nguồn được tổ chức độc lập với nhau. Sự thay đổi của một layer sẽ không ảnh hưởng tới layer khác.
- Testing dễ dàng khi chỉ cần viết test cho từng

Nhược điểm:

- Nếu sửa business thì khả năng cao sẽ phải sửa lại tất cả các tầng do không có sự phân tách business giữa các tầng.

Architecture Styles

Phân loại khác – Domain Partitioning



Các component của hệ thống được tổ chức theo domain.

Ưu điểm:

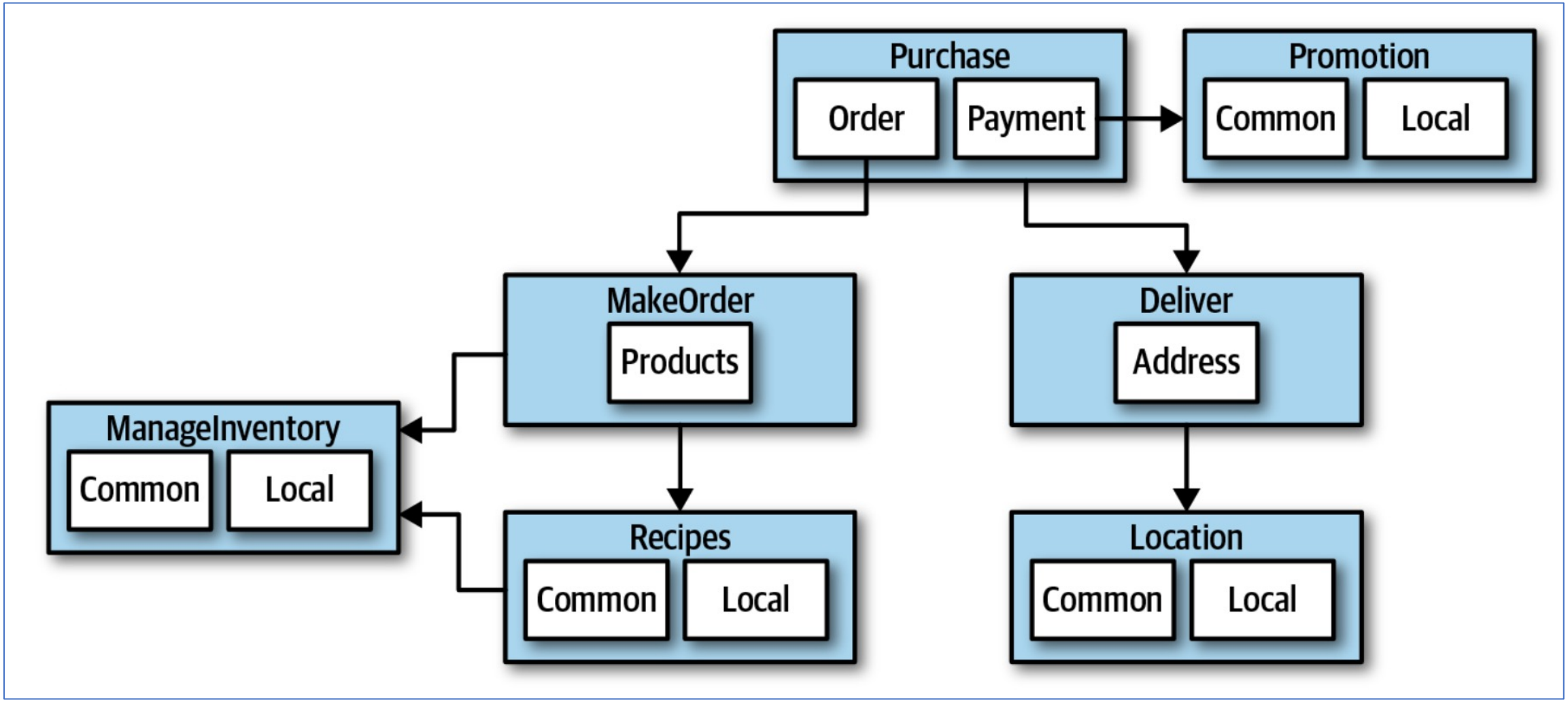
- Nếu business thay đổi thì chỉ ảnh hưởng tới những service chứa domain đó.
- Phù hợp với các nhóm đa chức năng có chuyên môn (cross-functional team). Mỗi nhóm sẽ phụ trách một domain riêng biệt.

Nhược điểm:

- Viết unit test không dễ do logic của các tầng phục vụ cho cùng một domain.

Architecture Styles

Phân loại khác – Domain Partitioning (tt)



Architecture Styles

Phân loại khác – Ví dụ

- Ví dụ: Catalog Service trong Online Shopping. Catalog sẽ quản lý product listings, categories và inventory.

Technical Partitioning:

- Catalog service được chia vào layers như presentation (UI), business logic, và persistence (tương tác với database)
- Presentation layer manages how products are displayed to users.
- Business logic layer handles pricing and availability rules
- Persistence layer: storing và retrieving product data từ DB

Domain Partitioning:

- Catalog service sẽ được tổ chức theo business domains hoặc workflows
- Catalog service bao gồm tất cả các chức năng liên quan managing products, categories, and inventory.
- Components bên trong Catalog service được cấu trúc dựa trên business logic như: product catalog management, category management

Monolithic Architecture vs Microservices Architecture

	Monolithic	Microservice
Architecture (Kiến trúc)	Single-tier	Multi tier
Size (Kích thước)	Large, All components tightly coupled (chặt chẽ)	Small, Loosely coupled(lỏng lẻo) components
Deployment (Triển khai)	Single unit	Individual services can be deployed independently
Scalability (Khả năng mở rộng)	Horizontal (chiều ngang) scaling can be challenging	Easier to scale horizontally

Monolithic Architecture vs Microservices Architecture (tt)

	Monolithic	Microservice
Development (Phát triển)	Simpler intitially	Complex due to managing multiple services
Techology (Kỹ thuật)	Limited technologies choices	Freedom to choose the best technology for each service
Fault Tolerance (Khả năng chịu lỗi)	Entire application may fail if a part fails	Individual services can fail without affecting others
Maintenance (Bảo trì)	Easier to main due to its simplicity	Requires more effort to manage multiple services

Monolithic Architecture vs Microservices Architecture (tt)

	Monolithic	Microservice
Flexibility (Linh hoạt)	Less flexible as all components are tightly coupled	More flexible as components can be developed, deployed, and
Communication (Giao tiếp)	Communication between components is faster	Communication may be slower due to network calls

Example

Giả sử viết ứng dụng Web có 2 chức năng:

1. ***Register, Login*** (Đăng ký, Đăng nhập)
2. ***Order*** (Đặt hàng)

Example (tt)

Mono:

- Nếu chọn ***Java*** để phát triển thì ***Spring MVC*** là thích hợp và ***tất cả các thành viên phải hiểu biết về framework này!***
- Nếu chọn ***CSharp*** để phát triển thì ***.NET*** là thích hợp và ***tất cả các thành viên phải hiểu biết về framework này!***

Microservice:

Giả sử chúng ta chia thành 2 service

1. Register, Login
2. Order

Thì 2 service này có thể sử dụng 2 ***ngôn ngữ khác nhau*** để phát triển. Và có thể sử dụng 2 DB hoàn toàn khác nhau để lưu trữ.

Architecture Patterns

Layered Architecture

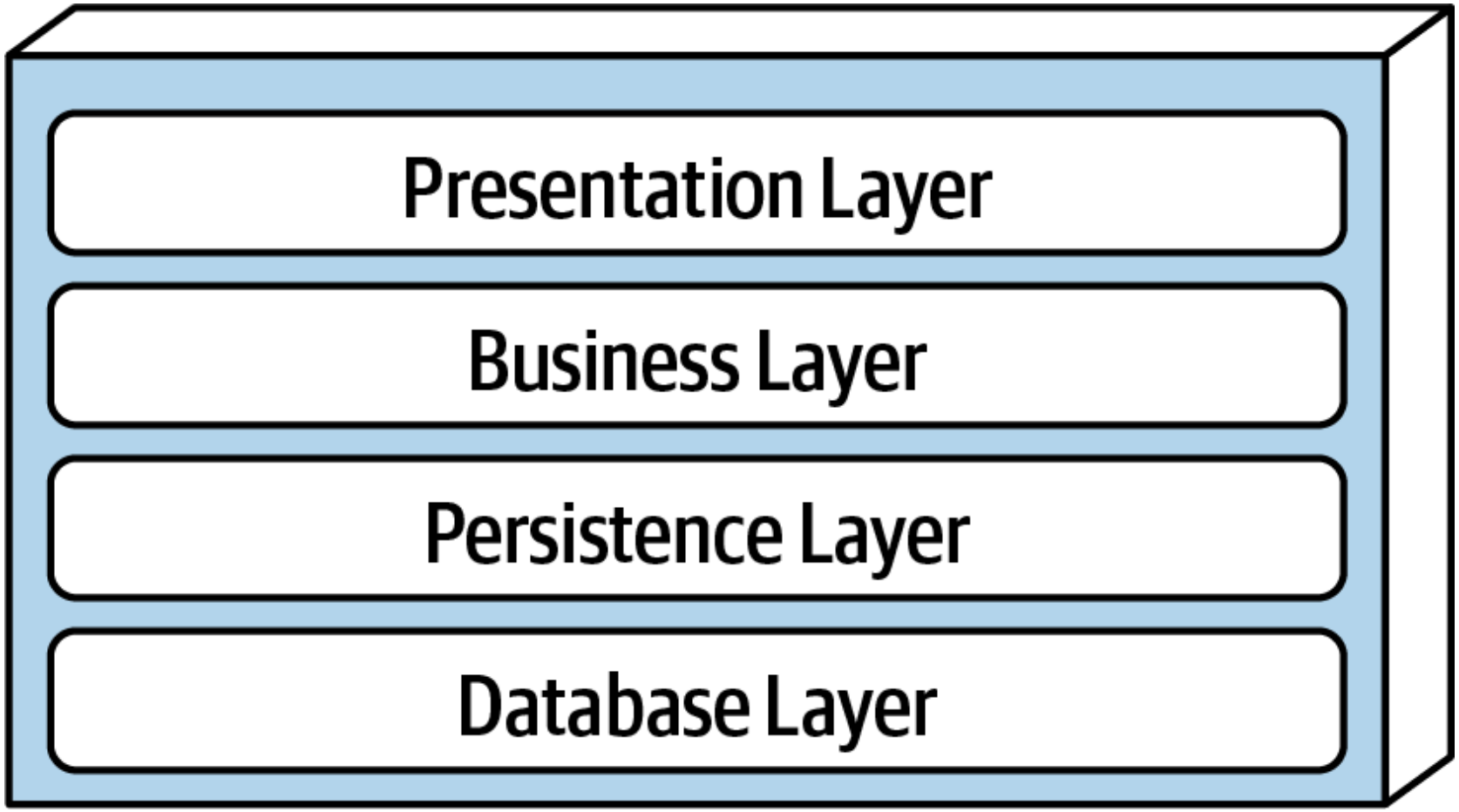


Fig. Standard logical layers within the layered architecture style

Architecture Patterns

Layered Architecture

1. **Lớp Presentation**: gồm các thành phần như giao diện người dùng, trình duyệt, ứng dụng di động, hoặc các thành phần tương tác với người dùng khác. Có chức năng hiển thị thông tin và kết quả từ lớp Logic cho người dùng và chuyển dữ liệu người dùng nhập vào lớp Logic.
2. **Lớp Business**: Đôi khi còn được gọi là lớp Service. Có nhiệm vụ xử lý các logic nghiệp vụ, nhận yêu cầu từ lớp Presentation, xử lý nó và gửi yêu cầu tương ứng đến lớp Persistence.
3. **Lớp Persistence**: Đảm nhiệm việc gửi các yêu cầu đến lớp Database để thực hiện các thao tác liên quan đến dữ liệu.
4. **Lớp Database**: Bao gồm cơ sở dữ liệu và các thành phần liên quan như hệ quản trị cơ sở dữ liệu. Có nhiệm vụ quản lý cơ sở dữ liệu, thực hiện thao tác đọc và ghi dữ liệu và triển khai các truy vấn và lưu trữ dữ liệu theo cách được định nghĩa từ lớp Persistence.

Architecture Patterns

Layered Architecture

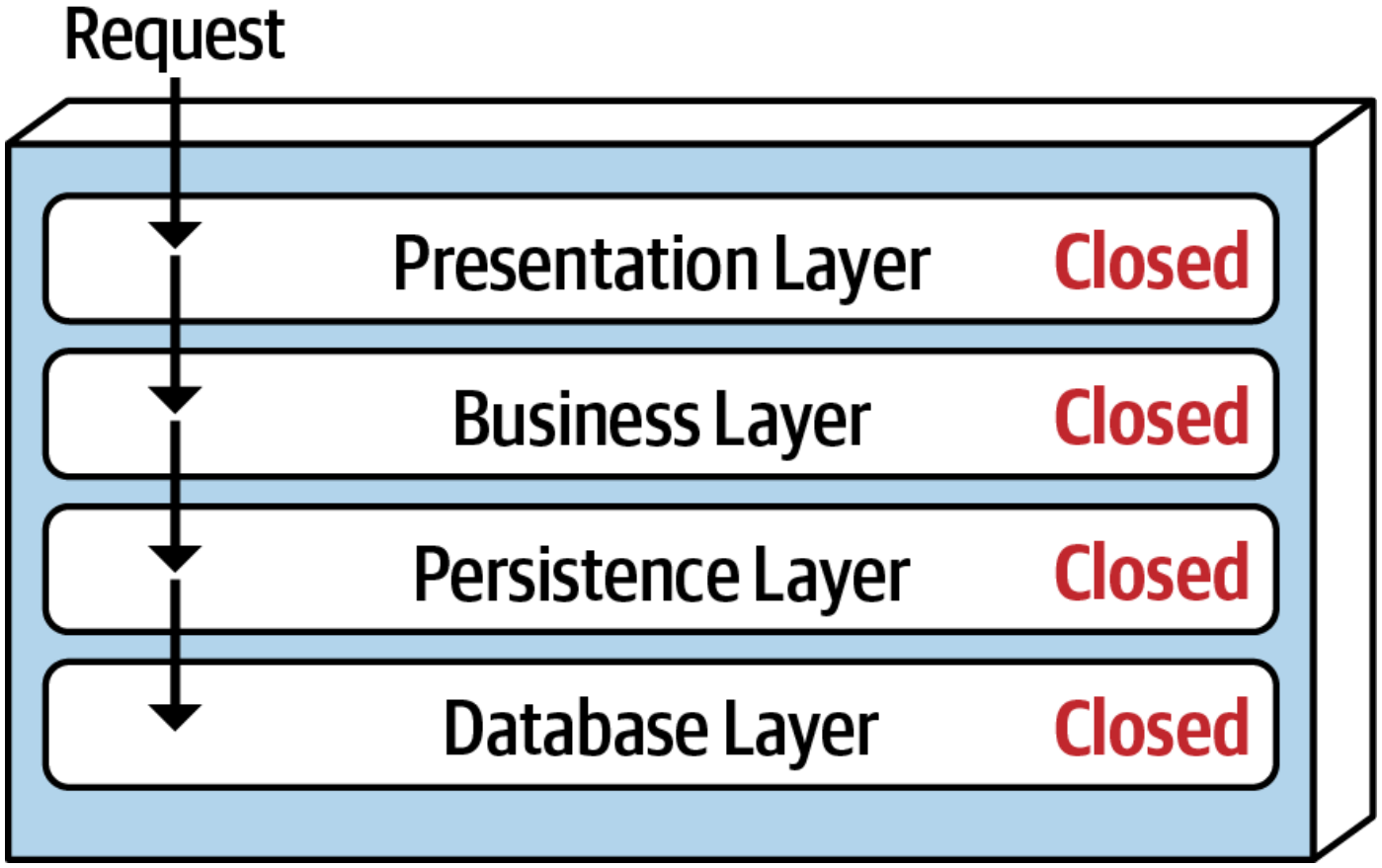


Fig. Closed layers within the layered architecture

Layers of Isolation (cô lập):

- Mỗi lớp trong kiểu kiến trúc nhiều layer có thể đóng (closed) hoặc mở (open).
- Một lớp đóng có nghĩa là khi một yêu cầu di chuyển từ trên xuống dưới từ lớp này sang lớp khác, yêu cầu không thể bỏ qua bất kỳ lớp nào mà phải đi qua lớp ngay bên dưới nó để đến lớp tiếp theo

Architecture Patterns

Layered Architecture – Add layer

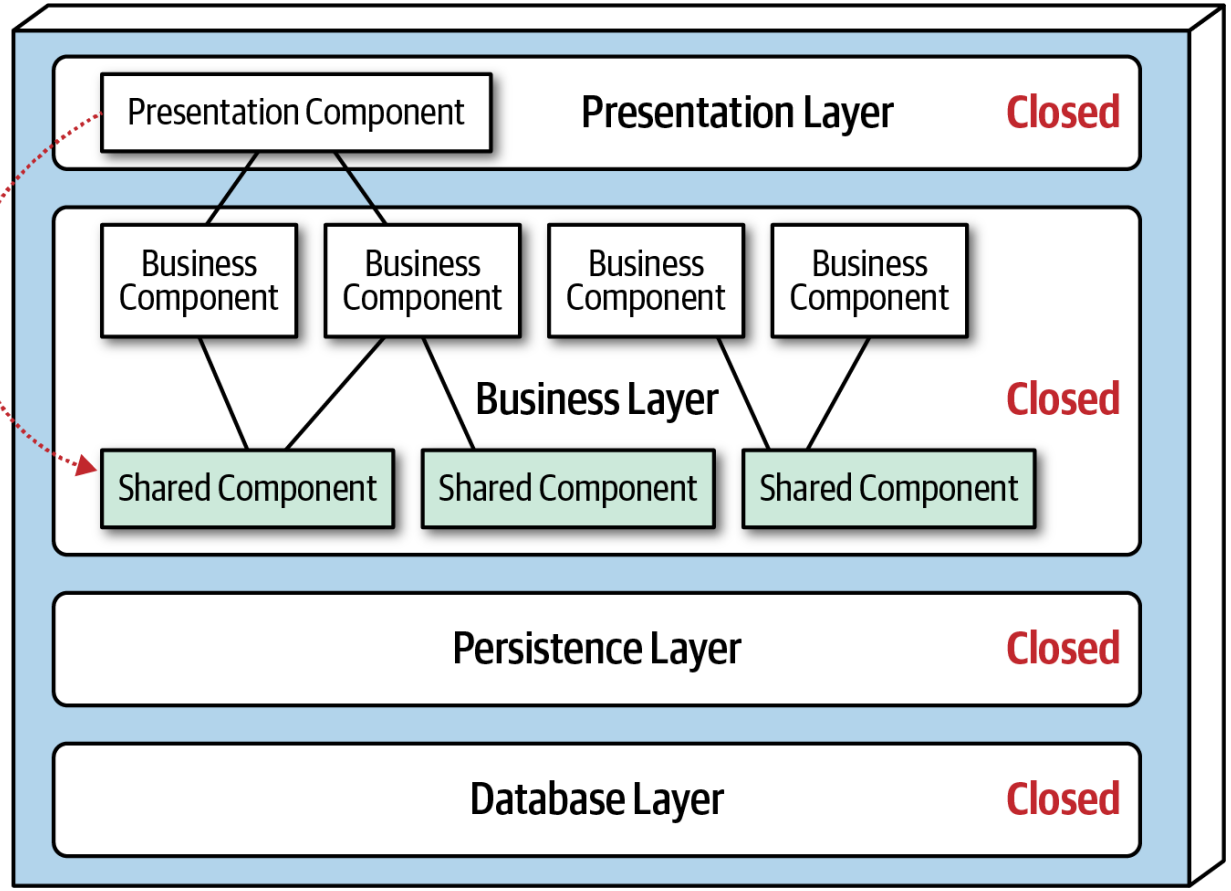


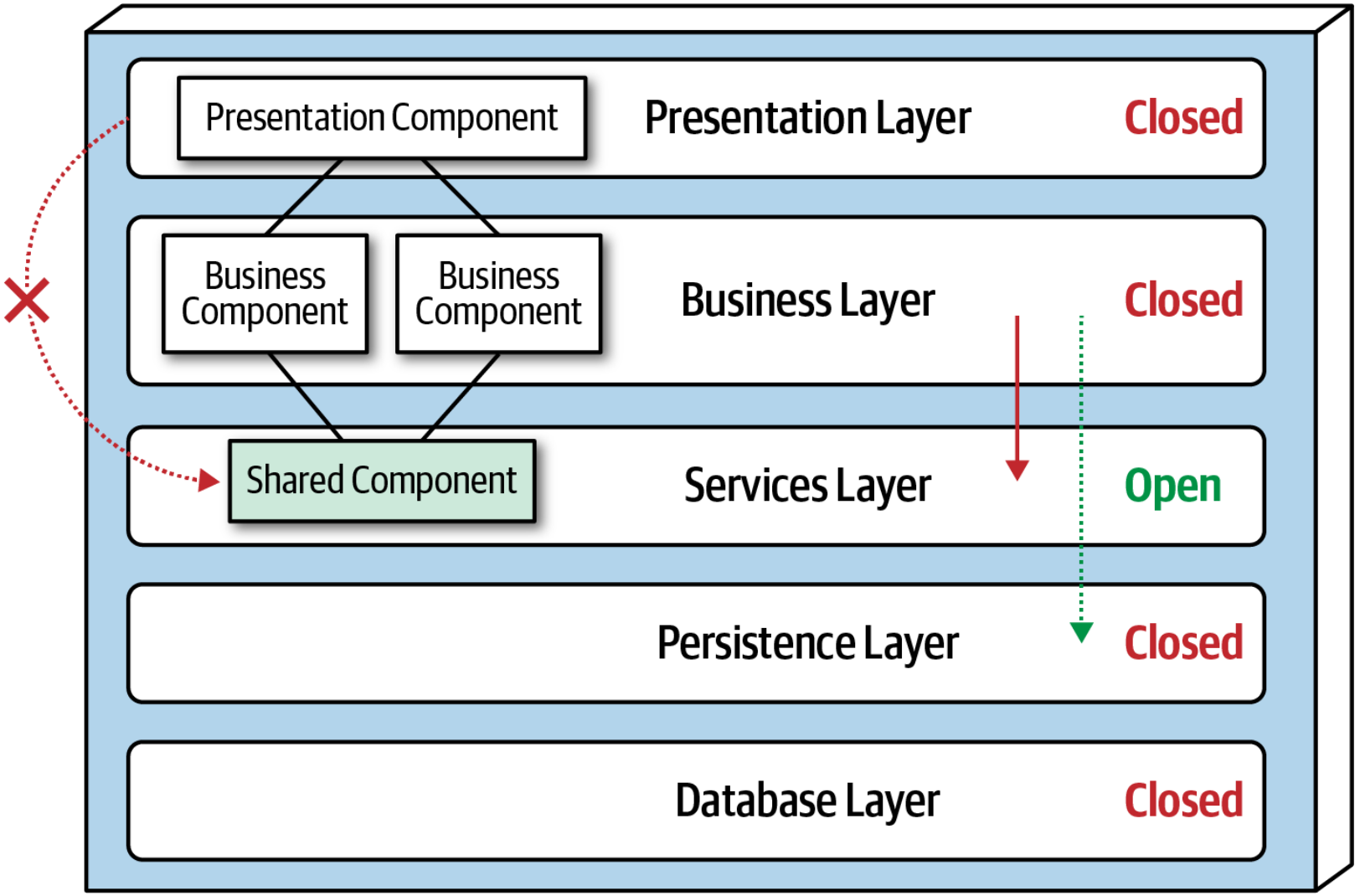
Fig. Shared objects within the business layer

- Trong khi các lớp đóng tạo điều kiện cho các lớp cô lập và giúp Layer cô lập thay đổi nội bộ kiến trúc, thì có những thay đổi Layer từ Closed sang Open là hợp lý.
- VD: như date and string utility classes, auditing classes, logging classes …

=> Nhưng kịch bản này khó quản lý và kiểm soát vì về mặt kiến trúc, Presentation có quyền truy cập vào Business và do đó có quyền truy cập vào các đối tượng được chia sẻ trong lớp đó.

Architecture Patterns

Layered Architecture – Add layer (tt)



- Thêm 1 layer Services Layer và layer này phải là ***Open***.
- Nếu không là ***Open*** thì ***Business Layer bắt buộc phải qua Service Layer thì mới truy cập được vào Persistence layer***

Fig. Adding a new services layer to the architecture

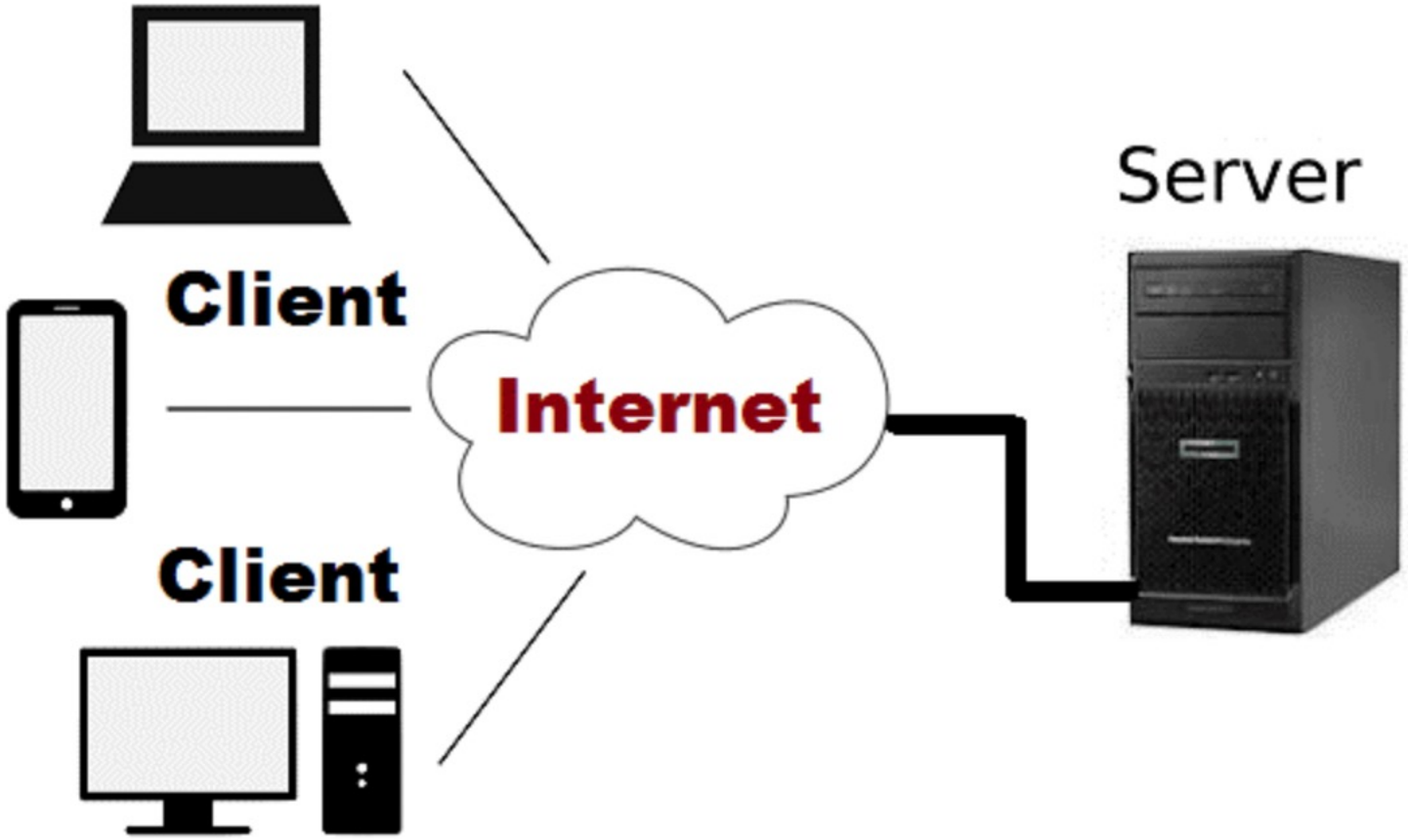
Architecture Patterns

Layered Architecture (tt)

- ***Ưu điểm:*** tách biệt logic giữa các tầng
- ***Khuyết điểm:*** kiến trúc phân lớp vẫn có một yếu điểm dễ thấy là logic phải được truyền theo thức tự các tầng, bất kể tầng đó có xử lý dữ liệu hay không. Điều này gây ra sự lãng phí tài nguyên.
- **Trường hợp nên dùng kiến trúc phân lớp:** Nếu dự án có những hạn chế nhất định về thời gian và ngân sách, kiến trúc phân lớp là một sự lựa chọn thích hợp vì các lý do chính sau:
  - Thuộc loại kiến trúc monolithic nên không phức tạp
  - Dựa trên mức độ phổ biến của kiến trúc phân lớp, hầu hết developer điều quen thuộc, việc vận hành, phát triển dự án trở nên dễ dàng hơn.
  - Kiến trúc phân lớp thuộc nhóm kiến trúc phân vùng kỹ thuật (Technical Partitioning) nên việc sử dụng kiến trúc này phù hợp với nhiều tổ chức khác nhau do các nhóm được chia thành nhóm chuyên trách kỹ thuật như nhóm FE, nhóm BE, nhóm database, ...

Architecture Patterns

Client – Server Architecture



Client-Server Architecture Examples:

1. Web Servers
2. E-Mail Servers
3. File Servers
4. Domain Name Server (DNS)

Architecture Patterns

Client – Server Architecture (tt)

- Kiến trúc Client-Server là nền tảng của thiết kế hệ thống hiện đại, trong đó cơ sở hạ tầng mạng được cấu trúc để bao gồm nhiều máy khách (Client) và một máy chủ trung tâm (Server).
- Client là thiết bị hoặc chương trình đưa ra yêu cầu (request) về dịch vụ (service) hoặc tài nguyên (resource).
- Server là máy chủ hoặc phần mềm đáp ứng các yêu cầu này. Giao tiếp giữa máy khách và máy chủ tuân theo giao thức request-response, chẳng hạn như HTTP/HTTPS cho dịch vụ web hoặc SQL cho truy vấn cơ sở dữ liệu.
  - Kiến trúc này cho phép quản lý dữ liệu và phân bổ tài nguyên hiệu quả bằng cách tập trung các chức năng quan trọng trên máy chủ, có khả năng xử lý phức tạp và lưu trữ dữ liệu quy mô lớn.
  - Máy khách quản lý các tương tác của người dùng và gửi các yêu cầu cụ thể đến máy chủ, máy chủ sẽ xử lý các yêu cầu này và gửi lại phản hồi.

Architecture Patterns

Client – Server Architecture (tt)

- Kiến trúc Client-Server có khả năng mở rộng cao vì nó có thể hỗ trợ nhiều máy khách hơn bằng cách mở rộng khả năng của máy chủ hoặc thêm các máy chủ khác.
- Thiết kế này phổ biến trong nhiều ứng dụng, bao gồm dịch vụ web, quản lý cơ sở dữ liệu và hệ thống email, cung cấp một khuôn khổ mạnh mẽ để phát triển và quản lý các hệ thống phân tán phức tạp một cách hiệu quả.

Architecture Patterns

Client – Server Architecture (tt)

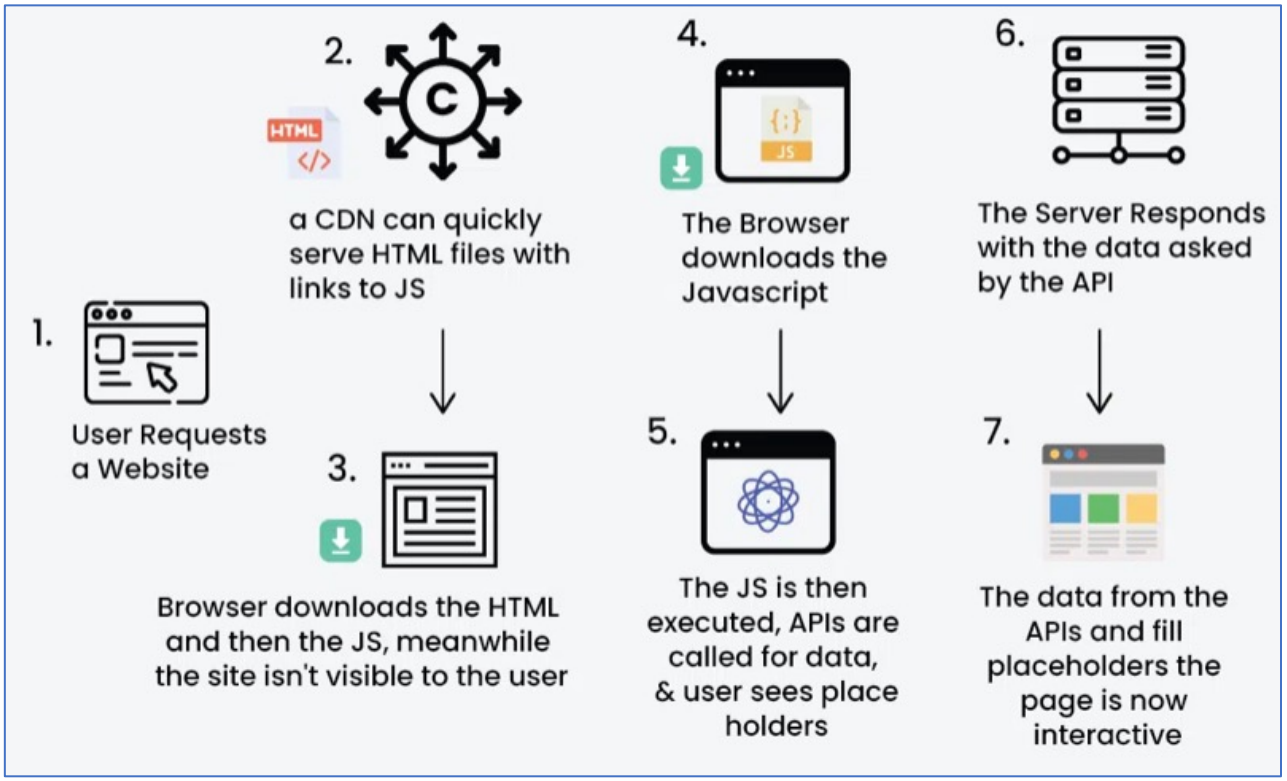
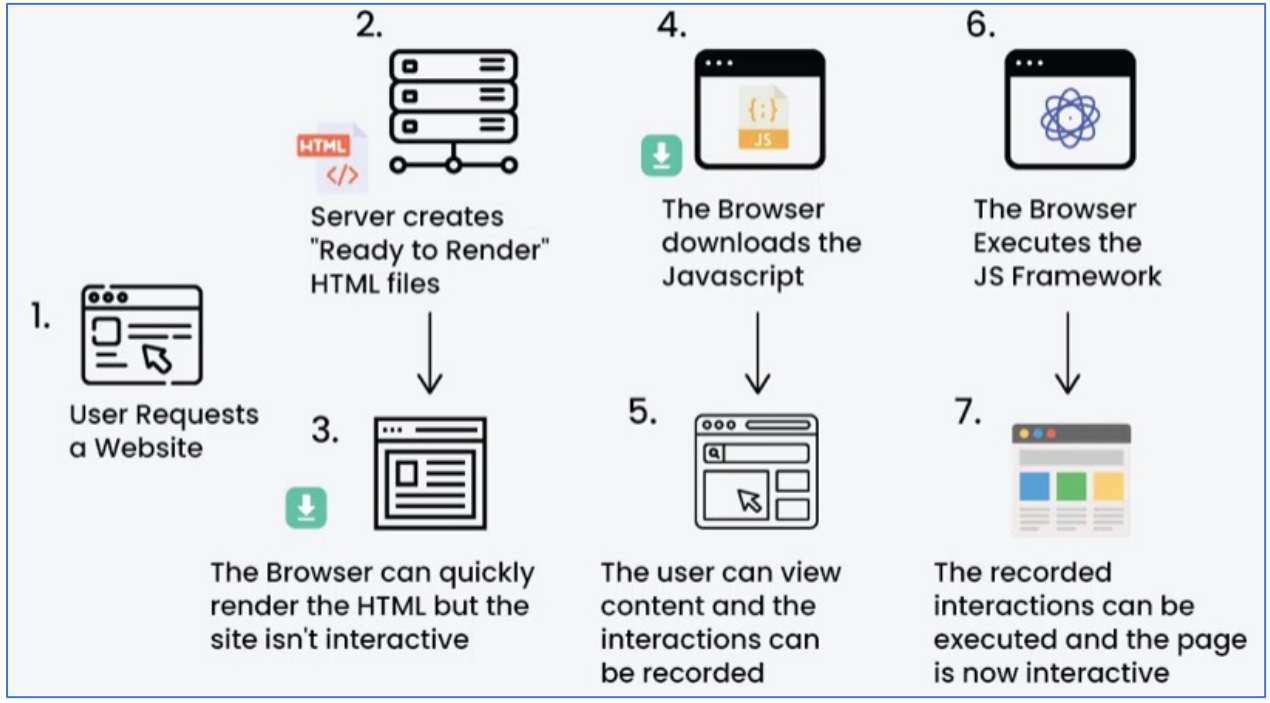


Fig. Client Side

Fig. Server Side



Architecture Patterns

Client – Server Architecture – Ưu điểm

- Quản lý tập trung trên một Server, kiến trúc này đơn giản hóa việc bảo trì, cập nhật và quản lý bảo mật. Người quản trị có thể giám sát và quản lý dữ liệu, áp dụng các bản cập nhật và thực thi chính sách bảo mật hiệu quả từ một vị trí duy nhất.
- Khả năng mở rộng: khi số lượng Client tăng lên, có thể thêm Server hoặc mở rộng dung lượng Servẻ hiện có mà không làm thay đổi đáng kể kiến trúc hệ thống tổng thể.
- Tối ưu hóa tài nguyên: mô hình này cho phép phân bổ tài nguyên được tối ưu hóa. Server được thiết kế để xử lý chuyên sâu và lưu trữ dữ liệu lớn, trong khi Client được tối ưu hóa cho các tương tác và yêu cầu của người dùng. Sự tách biệt này đảm bảo sử dụng hiệu quả tài nguyên hệ thống.
- Độ tin cậy và tính khả dụng: với cơ sở hạ tầng máy chủ mạnh, hệ thống Client-Server có thể đảm bảo độ tin cậy và tính khả dụng cao.

Architecture Patterns

Client – Server Architecture – Khuyết điểm

- Kiểm soát tập trung có thể dẫn đến khả năng xảy ra lỗi cao hơn. Khi nhiều Client gửi yêu cầu đồng thời đến Server, Server có thể quá tải và làm chậm hiệu suất. Điều này cũng có thể dẫn đến lỗi Server, khiến toàn bộ hệ thống ngừng hoạt động.
- Server mạnh hơn Client, nghĩa là Server đắt hơn. Server cũng yêu cầu người có kiến thức về mạng và cơ sở hạ tầng để quản lý hệ thống.
- Kiến trúc Client-Server dễ bị tấn công Từ chối dịch vụ (DoS) vì số lượng Server thường nhỏ hơn số lượng Client.
- Các gói dữ liệu có thể bị thay đổi trong quá trình truyền, dẫn đến khả năng mất thông tin hữu ích.

Architecture Patterns

Pipeline Architecture

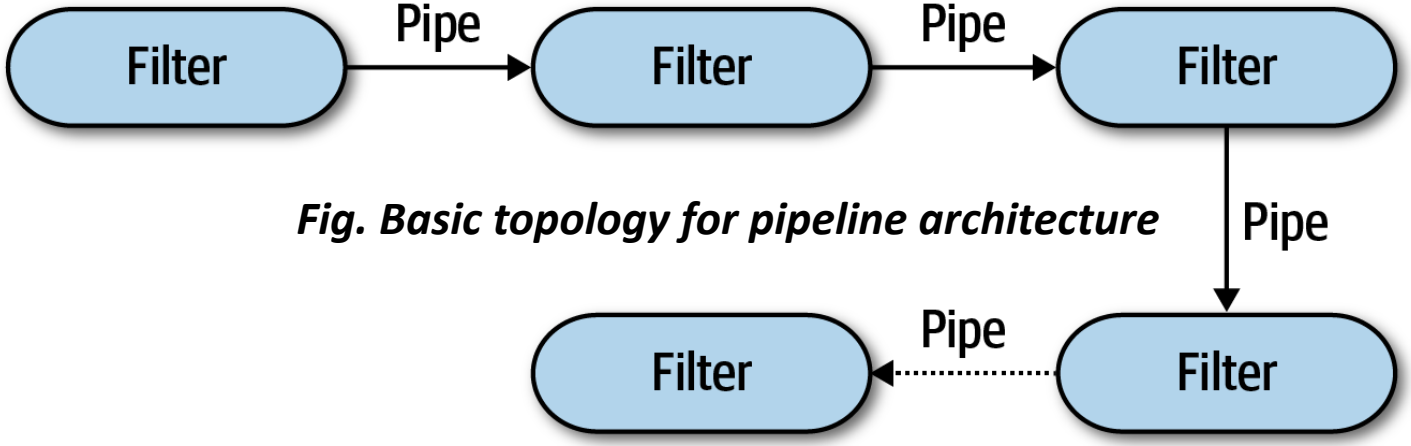


Fig. Basic topology for pipeline architecture

- Pipes: các pipe trong kiến trúc này tạo thành kênh truyền thông giữa các bộ lọc (Filter). Mỗi pipe thường là một chiều (unidirectional) và point-to-point (thay vì broadcast), chấp nhận đầu vào (input) từ một nguồn và luôn hướng đầu ra đến một nguồn khác. Thường thiết kế lượng dữ liệu nhỏ để cho phép hiệu suất cao.
- Filters: bộ lọc là độc lập, độc lập với các bộ lọc khác và thường không có trạng thái (stage). Các bộ lọc chỉ nên thực hiện một tác vụ. Các tác vụ tổng hợp nên được xử lý bằng một chuỗi các bộ lọc thay vì một bộ lọc duy nhất.

Architecture Patterns

Pipeline Architecture – Ví dụ

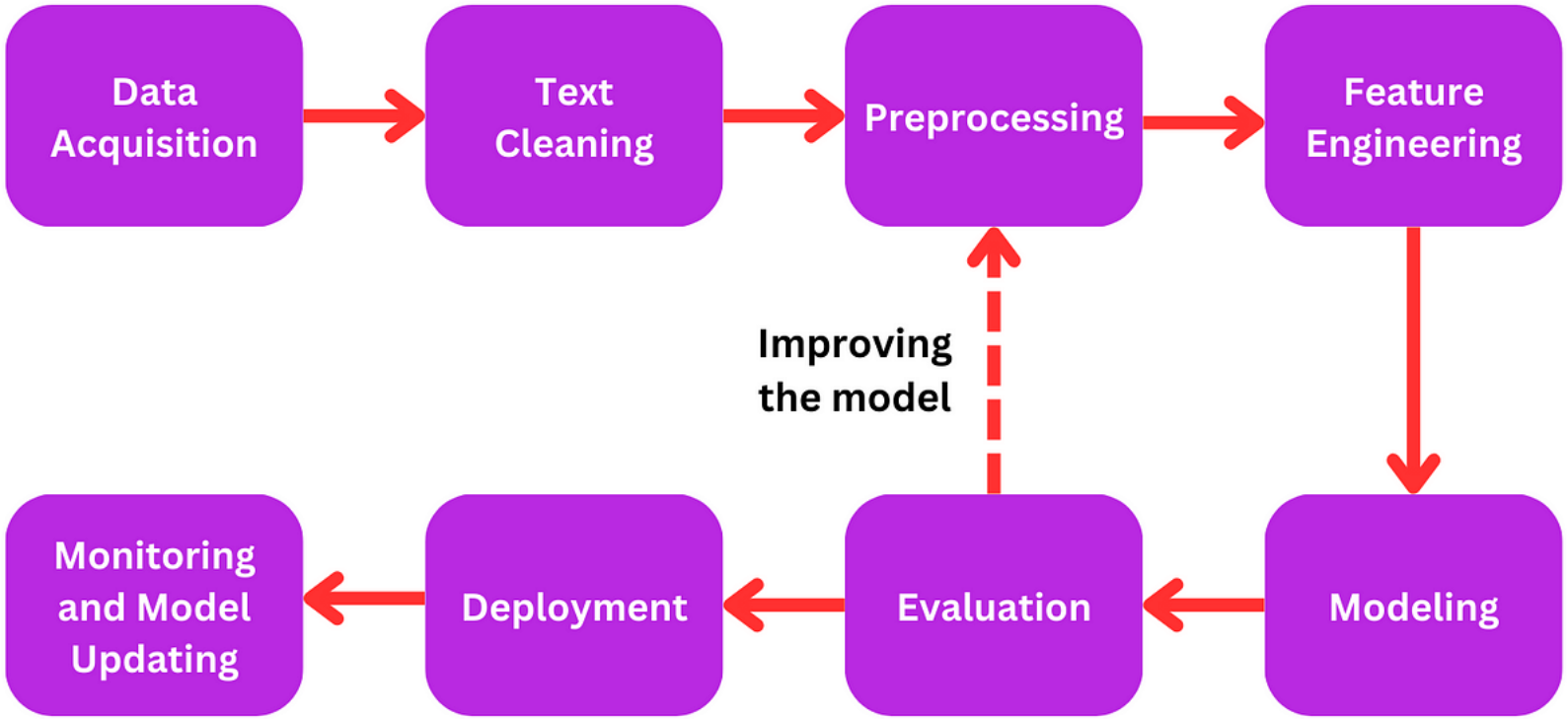


Fig. Natural Language Processing (NLP) Pipelines

Architecture Patterns

Pipeline Architecture

- ***Ưu điểm:*** dễ dàng thay đổi/bảo trì/dùng lại từng filter của hệ thống, phù hợp với nhiều hoạt động nghiệp vụ, dễ dàng nâng cấp bằng cách thêm filter mới.
- ***Khuyết điểm:*** 2 filter kề nhau cần tuân thủ định dạng dữ liệu chung
- **Tình huống nên dùng**: trong các ứng dụng xử lý dữ liệu mà dữ liệu nhập cần được xử lý bởi nhiều công đoạn khác nhau và có tính độc lập cao trước khi tạo ra kết quả cuối cùng.

Architecture Patterns

Event-Driven Architecture - EDA

- Là một mô hình kiến trúc phần mềm trong đó các thành phần hoặc dịch vụ của hệ thống giao tiếp với nhau chủ yếu thông qua sự kiện (event): **event producer (sản xuất)** và **event consumer (tiêu thụ)**.
- Các "sự kiện" (event) này có thể là các hành động người dùng, cập nhật dữ liệu, hoặc các thông báo từ các hệ thống khác. EDA giúp tạo ra các hệ thống linh hoạt, mở rộng và dễ tích hợp.
- Phân loại kiến trúc hướng sự kiện:
  - **Broker topology:** luồng tin nhắn (RabbitMQ, ActiveMQ, …) sẽ phân phối đều tới các event processor qua message broker mà không cần tới một trung tâm điều phối event.
  - **Mediator topology**: có một thành phần trung gian, thường được gọi là "**mediator**", được sử dụng để điều phối và quản lý luồng sự kiện giữa các service hoặc component khác nhau trong hệ thống.

Architecture Patterns

Event-Driven Architecture – Broker topology

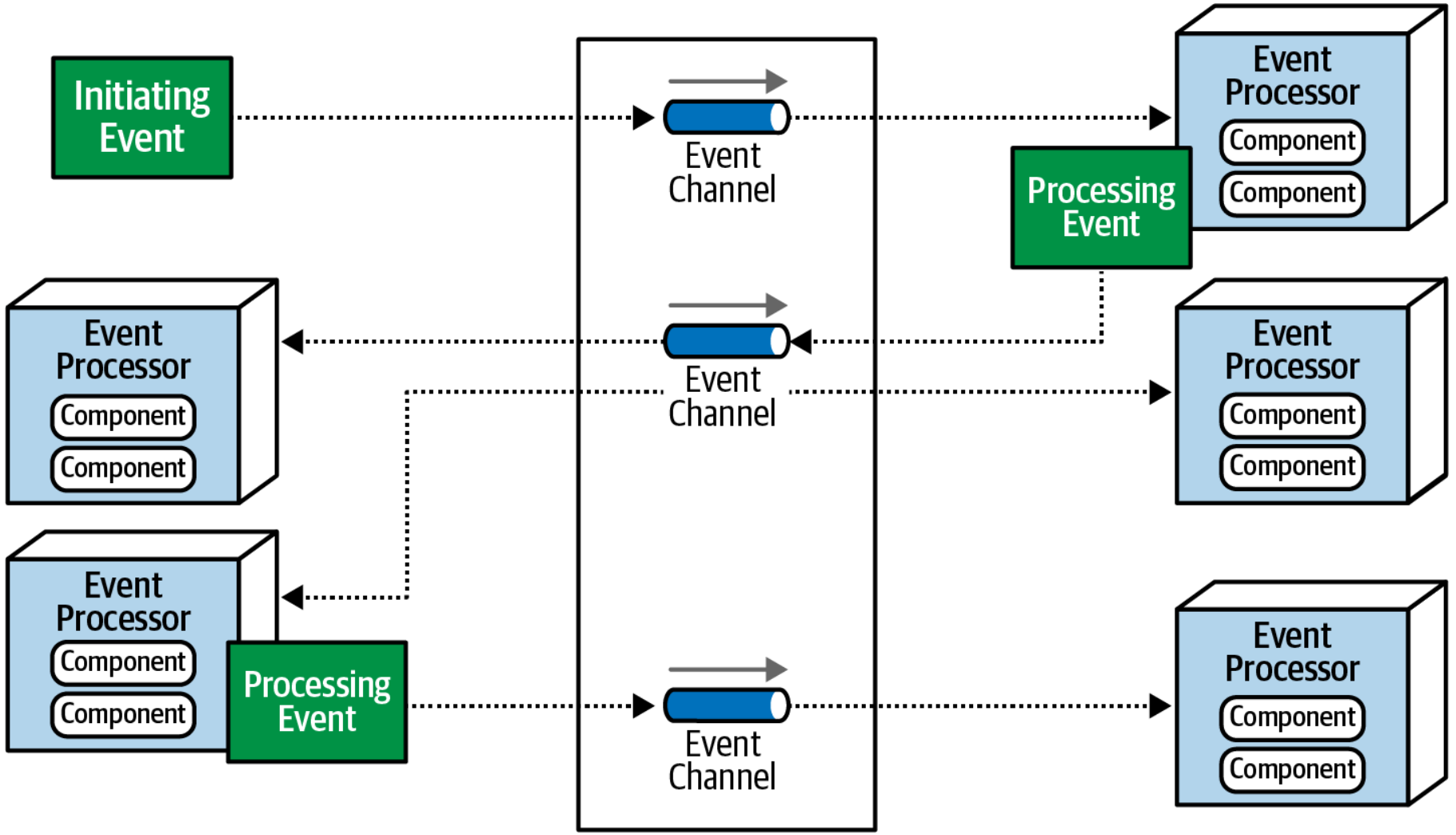


Fig. Broker topology

Architecture Patterns

Event-Driven Architecture – Mediator topology

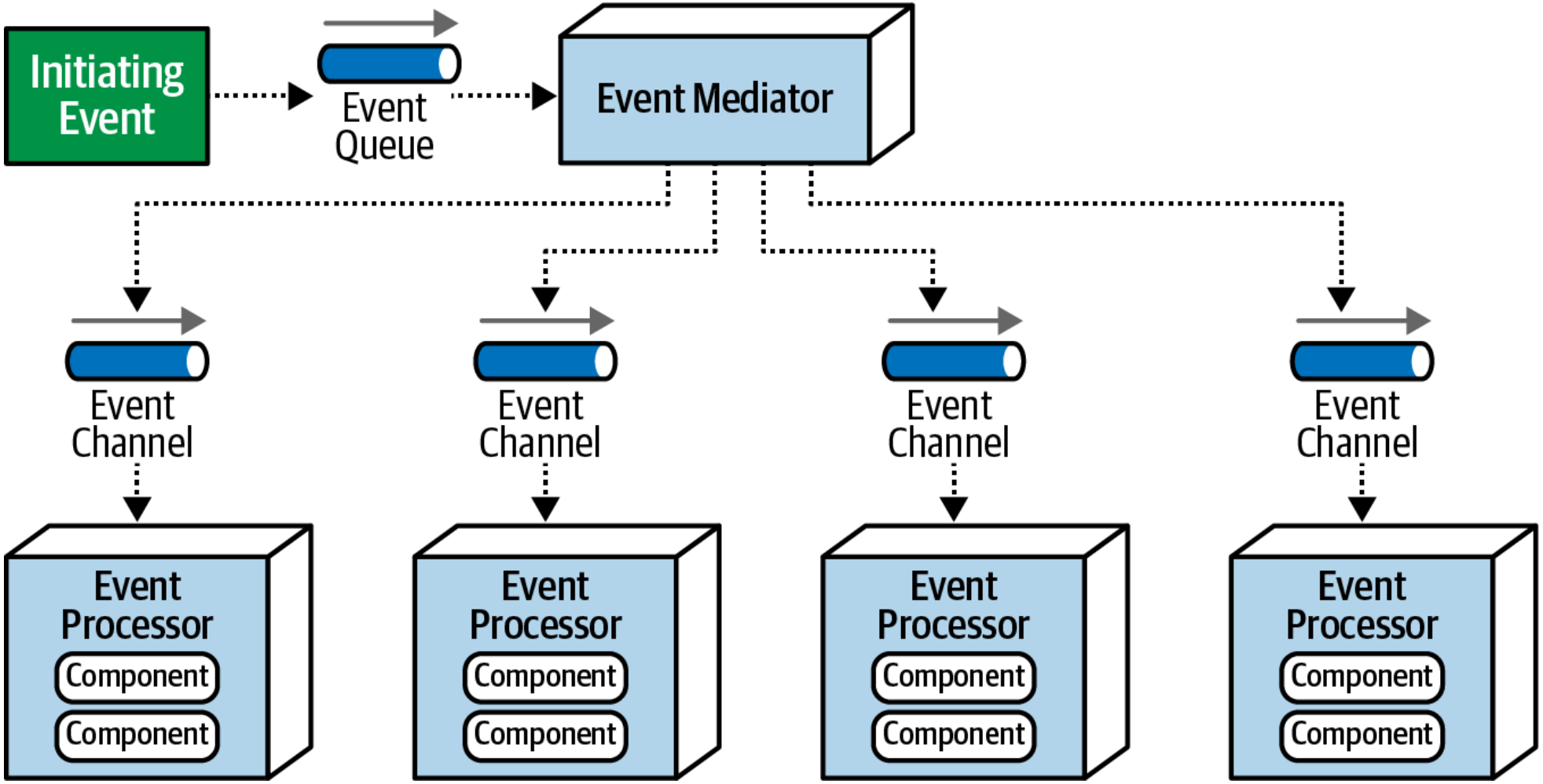
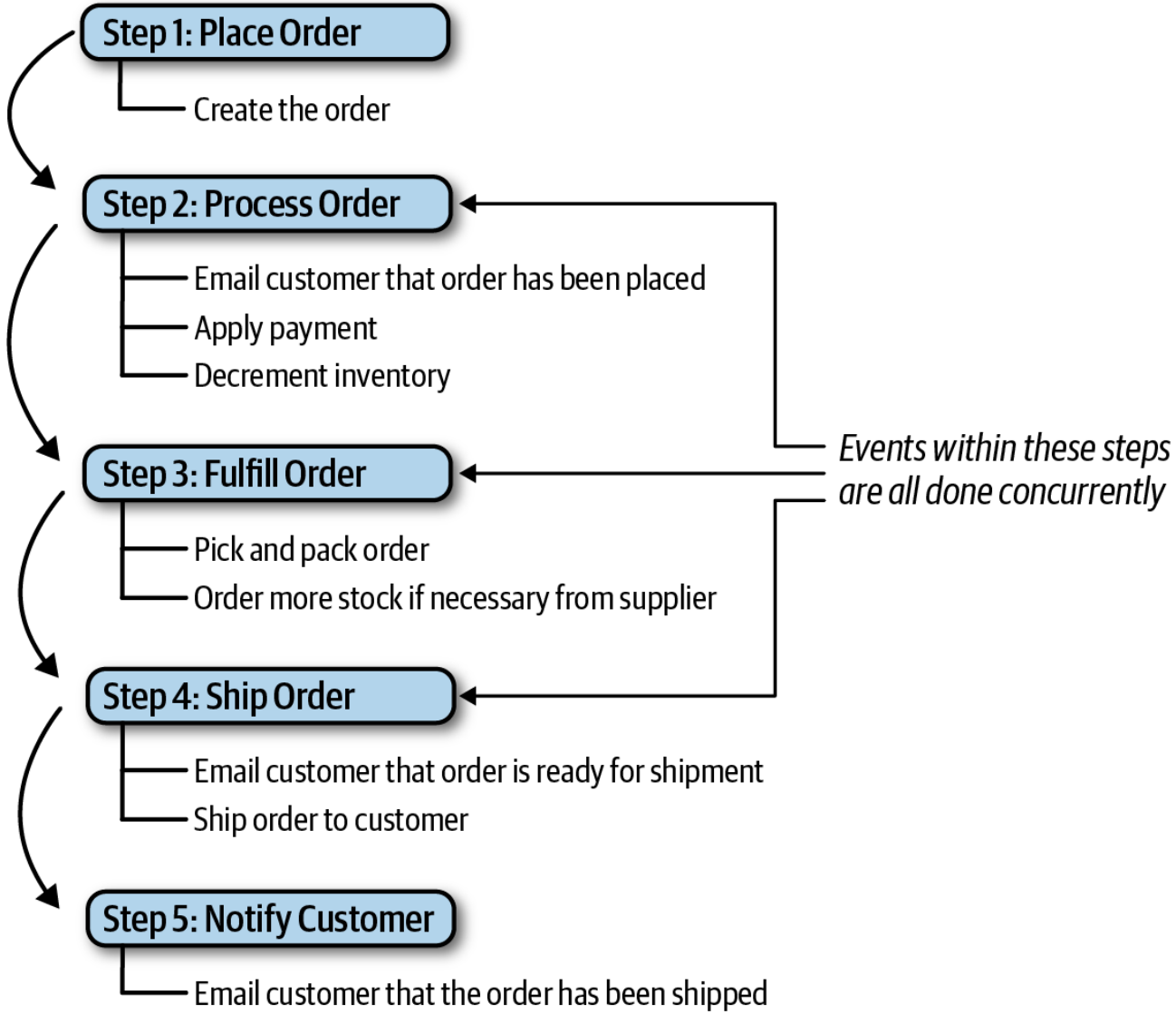


Fig. Mediator topology

Architecture Patterns

Event-Driven Architecture – Mediator topology



Architecture Patterns

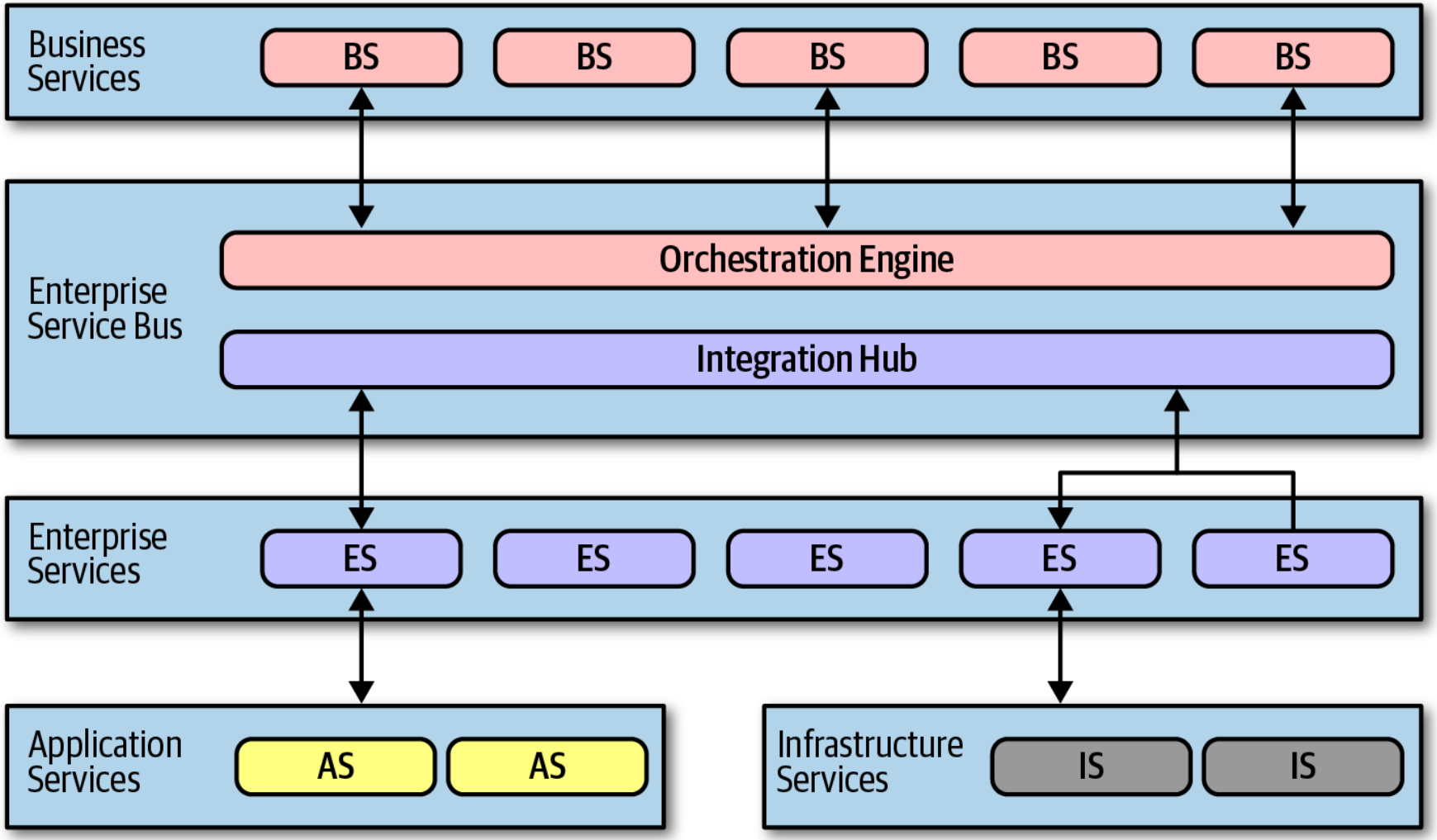
Event-Driven Architecture – Mediator topology

	Broker topology	Mediator topology
Intiating event	Khởi tạo event	
Event queue	Được sử dụng để lưu trữ các sự kiện được tạo ra và phân phối các sự kiện đó đến một service phản hồi chúng. Event channel có thể ở dạng topic hoặc queue	Lưu trữ các event khởi tạo, đảm bảo chúng không bị mất mát trong quá trình xử lý. Queue giúp xử lý các sự kiện tuần tự, đảm bảo tính nhất quán trước khi event được chuyển
Event broker/mediator	Chứa các event channel tham gia vào một luồng sự kiện	Nơi điều phối và quản lý luồng sự kiện, là trung tâm của mô hình Meditator
Event processor	Service chịu trách nhiệm xử lý sự kiện	
Processing event	Là event được tạo khi trạng thái của một số service thay đổi và gửi thông báo cho phần còn lại của hệ thống về sự thay đổi trạng thái đó. Event này được gửi theo cơ chế *fire-and-forget broadcasting*, tức là gửi và sau đó không cần xác nhận việc gửi có thành công hay không.	

Architecture Patterns

Service-Oriented Architecture - SOA

- **Chapter 16. Service-Oriented Architecture – Page 199**



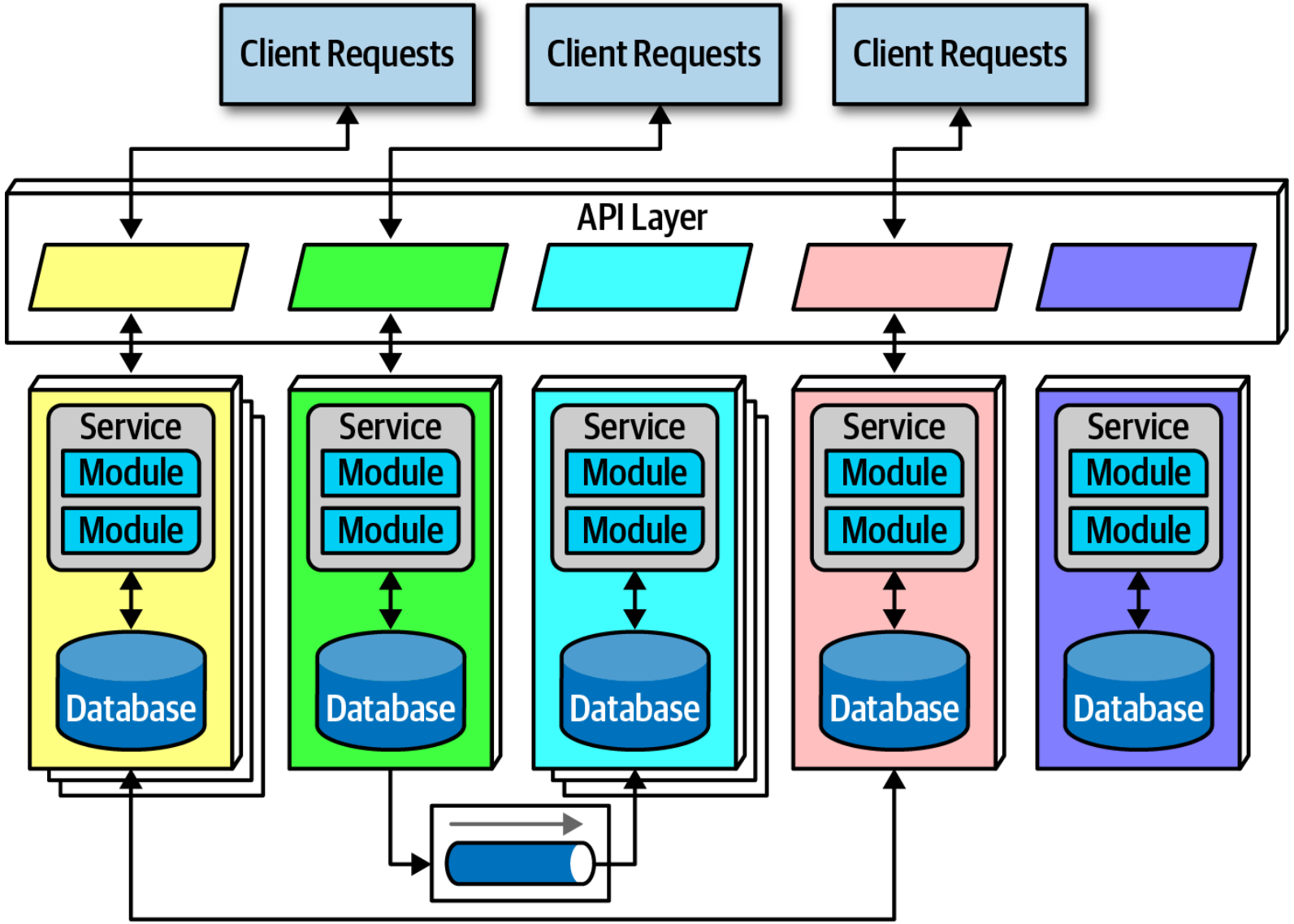
Architecture Patterns

Service-Oriented Architecture (tt)

- Kiến trúc hướng dịch vụ (SOA) là một phương pháp phát triển phần mềm sử dụng các thành phần của phần mềm được gọi là dịch vụ để tạo ra các ứng dụng dành cho doanh nghiệp. Mỗi dịch vụ cung cấp một tính năng doanh nghiệp, đồng thời các dịch vụ cũng có thể giao tiếp với nhau giữa nhiều nền tảng và ngôn ngữ. Nhà phát triển tận dụng SOA để tái sử dụng các dịch vụ trong nhiều hệ thống khác nhau hoặc kết hợp một số dịch vụ độc lập để thực hiện các tác vụ phức tạp.
- Ví dụ: nhiều quy trình kinh doanh trong tổ chức yêu cầu chức năng xác thực người dùng. Thay vì phải viết lại đoạn mã xác thực cho tất cả quy trình kinh doanh, chúng ta có thể tạo và tái sử dụng duy nhất một dịch vụ xác thực cho mọi ứng dụng. Tương tự, hầu hết mọi hệ thống trong tổ chức chăm sóc sức khỏe, chẳng hạn như hệ thống quản lý bệnh nhân và hệ thống hồ sơ y tế điện tử cần đăng ký bệnh nhân. Những hệ thống này có thể gọi một dịch vụ chung để thực hiện tác vụ đăng ký bệnh nhân.

Architecture Patterns

Microservices Architecture



Architecture Patterns

Microservices Architecture (tt)

- Microservice vốn là một thuật ngữ được đặt ra vào đầu những năm 2010, đề cập đến phong cách phát triển phần mềm trong đó các ứng dụng bao gồm các dịch vụ nhỏ và độc lập, thường gọi là *"microservice"*.
- Mỗi microservice được thiết kế để thực hiện một chức năng duy nhất hoặc một nhóm nhỏ các chức năng liên quan, được quản lý độc lập với nhau.
- ***The term "microservice" is a label, not a description.***

Architecture Patterns

Microservices Architecture (tt)

Các thành phần của **kiến trúc microservice** cơ bản bao gồm:

- Service: chứa logic nghiệp vụ của microservice, chỉ có service mới được truy cập vào database
- Database
- API gateway: cung cấp một điểm truy cập thống nhất cho các microservice, giúp quản lý, điều phối và bảo mật giao tiếp giữa các service với nhau trong kiến trúc microservices, đồng thời cung cấp một nền tảng để triển khai các chính sách quản lý API và tối ưu giao tiếp.

Trong thực tế, người ta thường sử dụng **message broker** và **API layer**. Client giao tiếp với microservice thông qua API layer. Còn các microservice tương tác với nhau thông qua API dựa trên giao thức HTTP/REST, ... hoặc thông qua một message broker.

Architecture Patterns

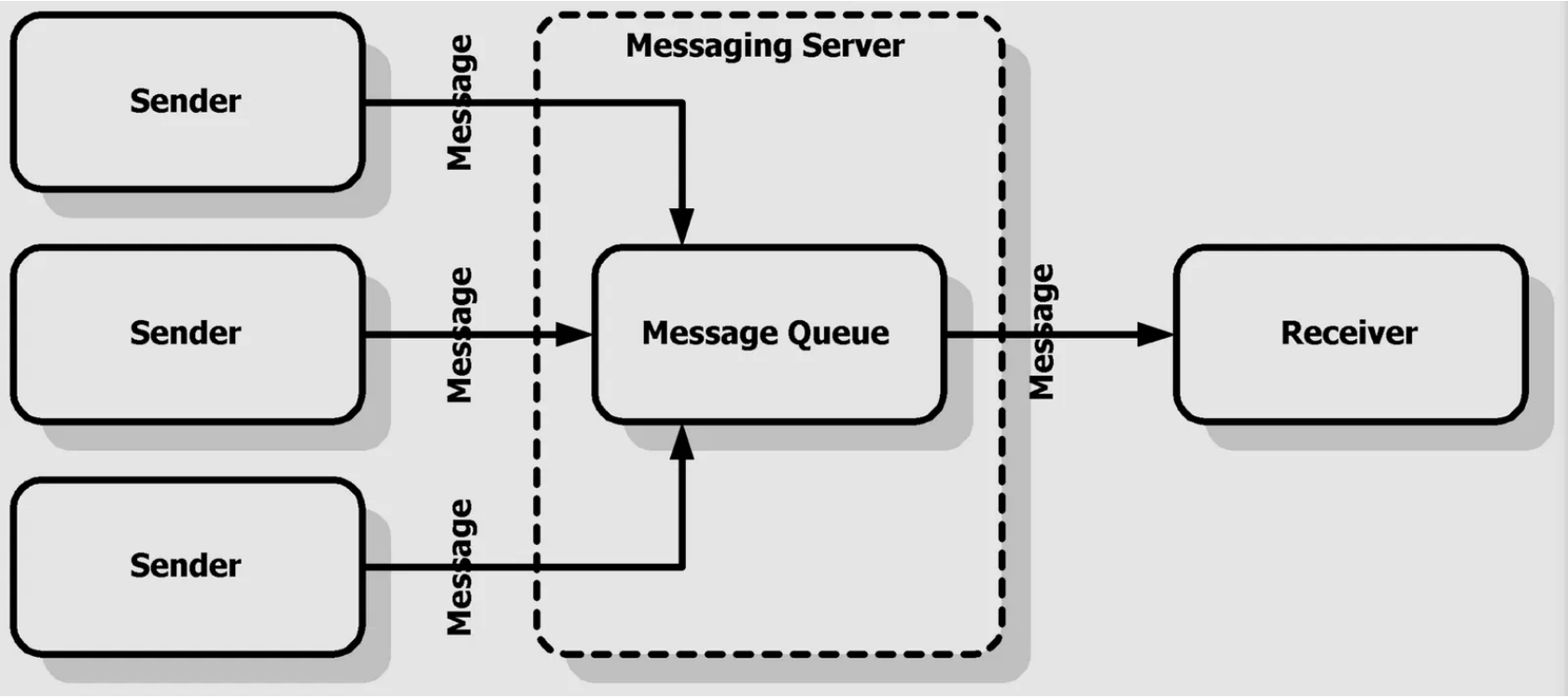
Microservices Architecture – Message broker

1. **Message broker** là một ứng dụng phần mềm đóng vai trò là trung gian trong việc truyền đạt thông điệp giữa các ứng dụng, hệ thống, hoặc component. Nó cho phép các ứng dụng gửi và nhận thông điệp mà không cần biết về sự tồn tại lẫn nhau, từ đó giảm thiểu sự phụ thuộc và tăng cường khả năng mở rộng và linh hoạt của hệ thống.
2. **Các loại mô hình Message broker:** 3 mô hình phổ biến:
   - Mô hình Point-to-Point (P2P)
   - Mô hình Publisher/Subscriber (Pub/Sub)
   - Mô hình Hybrid giữa Pub/Sub và P2P

Architecture Patterns

Microservices Architecture – Message broker (tt)

- Mô hình Point-to-Point (P2P):
  - Thường triển khai dưới dạng Queue
  - Yêu cầu xử lý chỉ 1 lần.
  - Ví dụ, yêu cầu thanh toán, gửi tiền vào tài khoản, chuyển tiền, giao hàng, …



Architecture Patterns

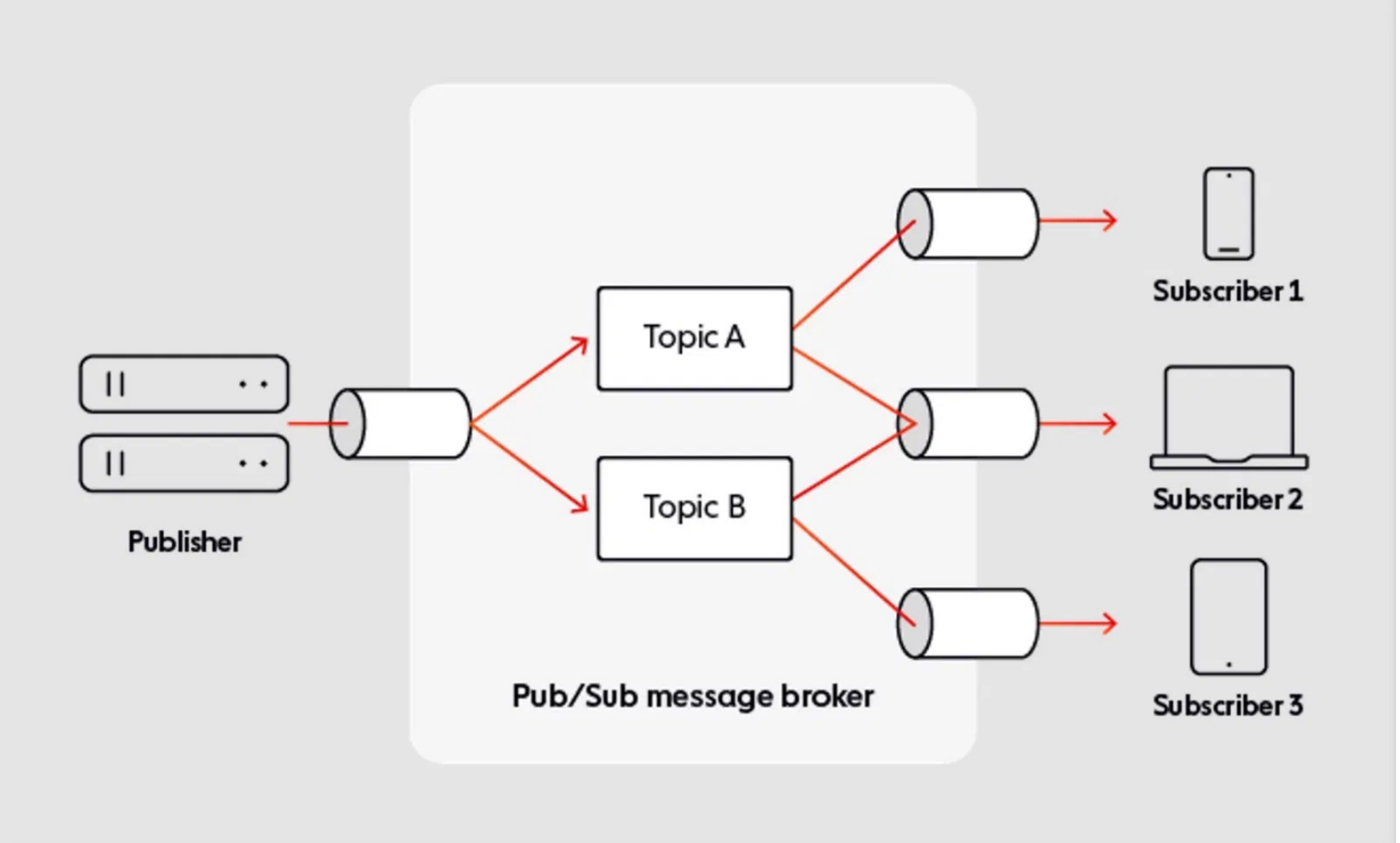
Microservices Architecture – Message broker (tt)

- **Mô hình Publisher/Subscriber (Pub/Sub)**:
  - Ví dụ điển hình cho **này là các khoá học online**. Google Meet hoạt động như một cơ chế phát sóng (broadcast mechanism). Những người trong Meet đóng vai vừa là publisher, vừa là subcriber.
  - **Pub/Sub** thường được triển khai thông qua topic. Topic cung cấp cùng một loại cơ chế phát sóng như Meet. Khi một message được gửi từ ***publisher*** vào một topic, nó sẽ được phân phối đến tất cả những ***consumer*** đã đăng ký. Các consumer sẽ bỏ lỡ message khi consumer đó offline. Vậy nên, Pub/Sub chỉ đảm bảo mỗi message được gửi ***tối đa một lần*** cho mỗi consumer đã đăng ký.

Architecture Patterns

Microservices Architecture – Message broker (tt)

- **Mô hình Publisher/Subscriber (tt)**



Architecture Patterns

Microservices Architecture – Frontends – Monolithic UI

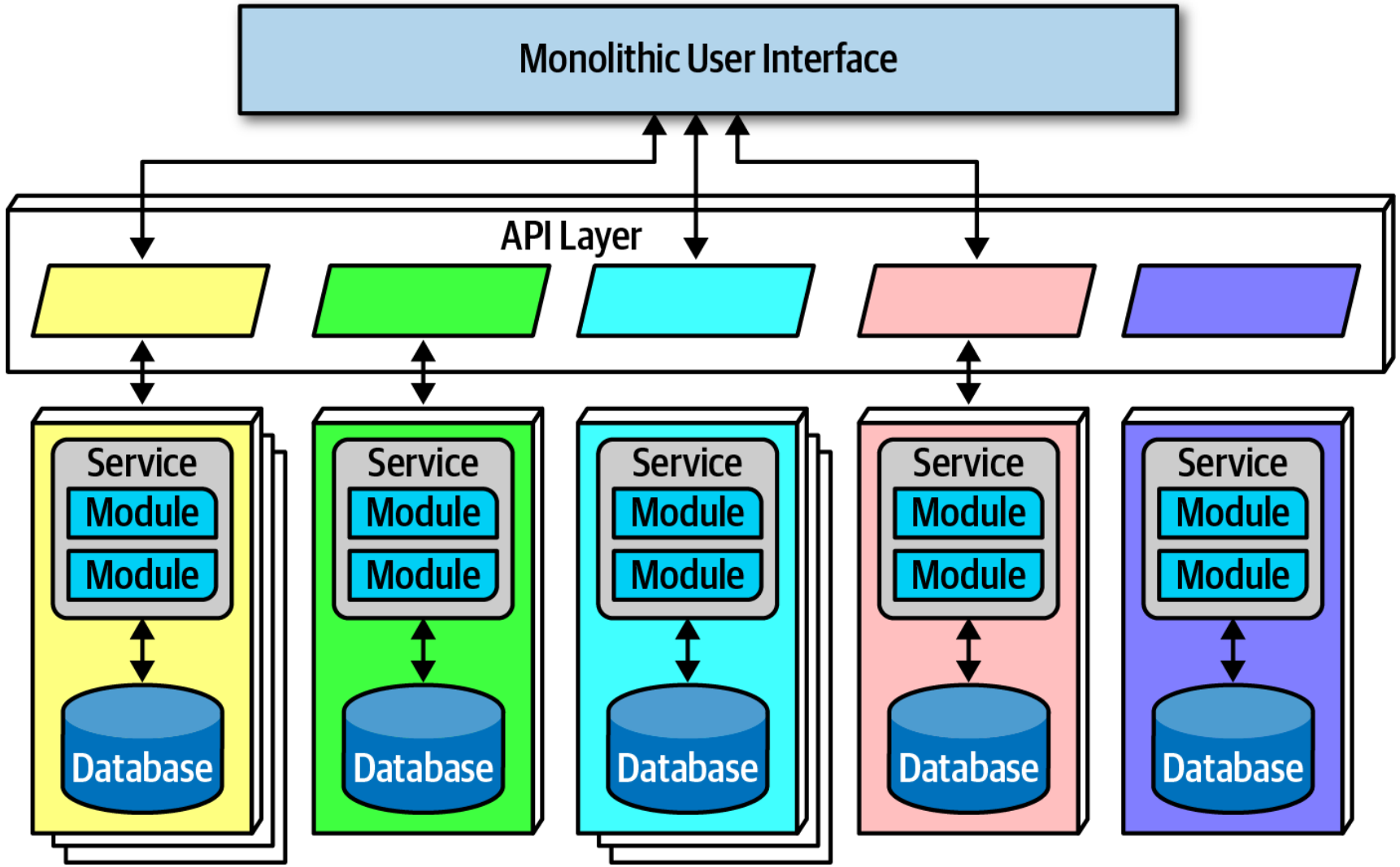


Fig. Microservices architecture with a monolithic user interface

Architecture Patterns

Microservices Architecture – Frontends - Microfrontend

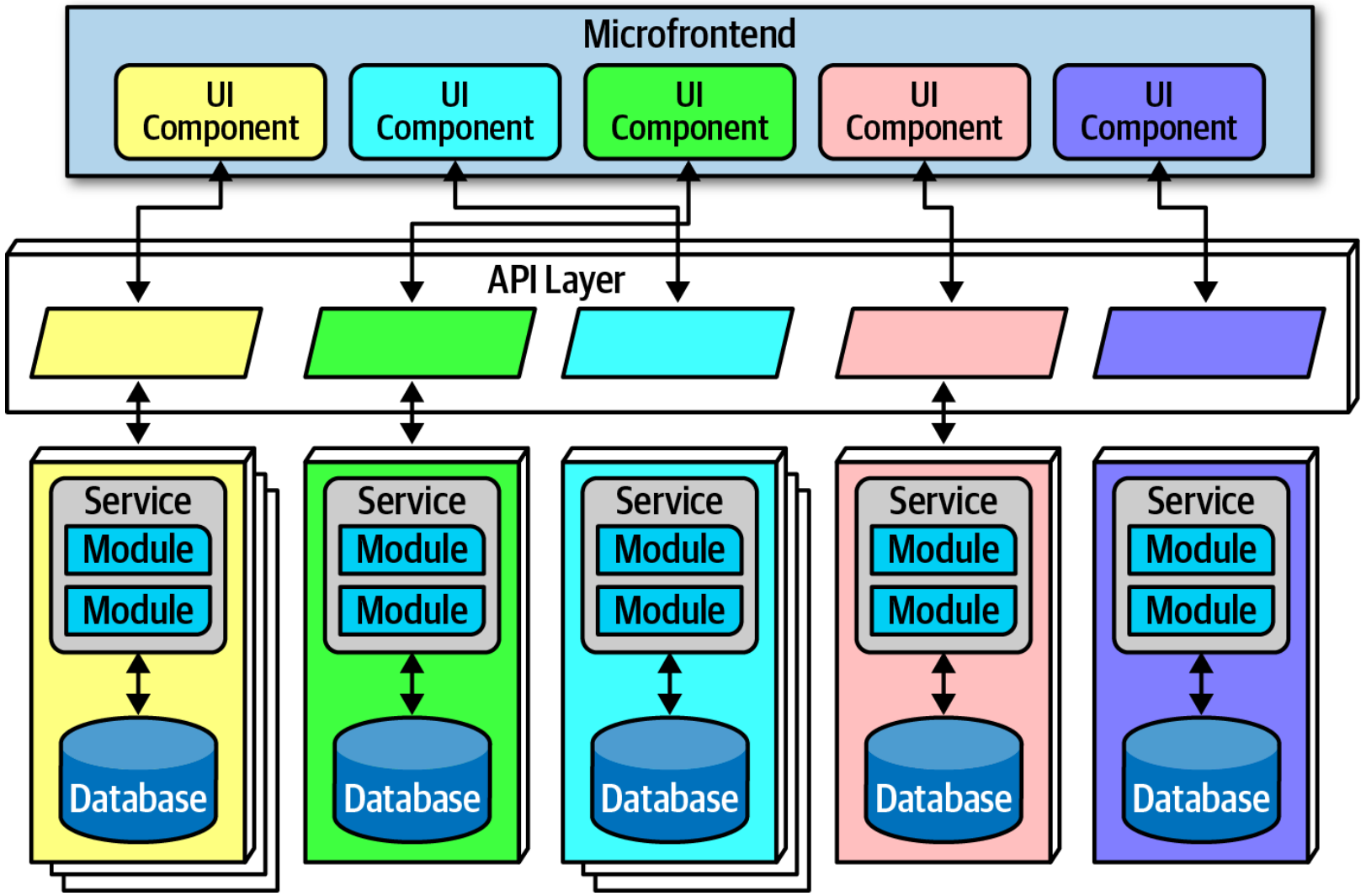


Fig. Microfrontend pattern in microservices

Architecture Patterns

Microservices Architecture (tt)

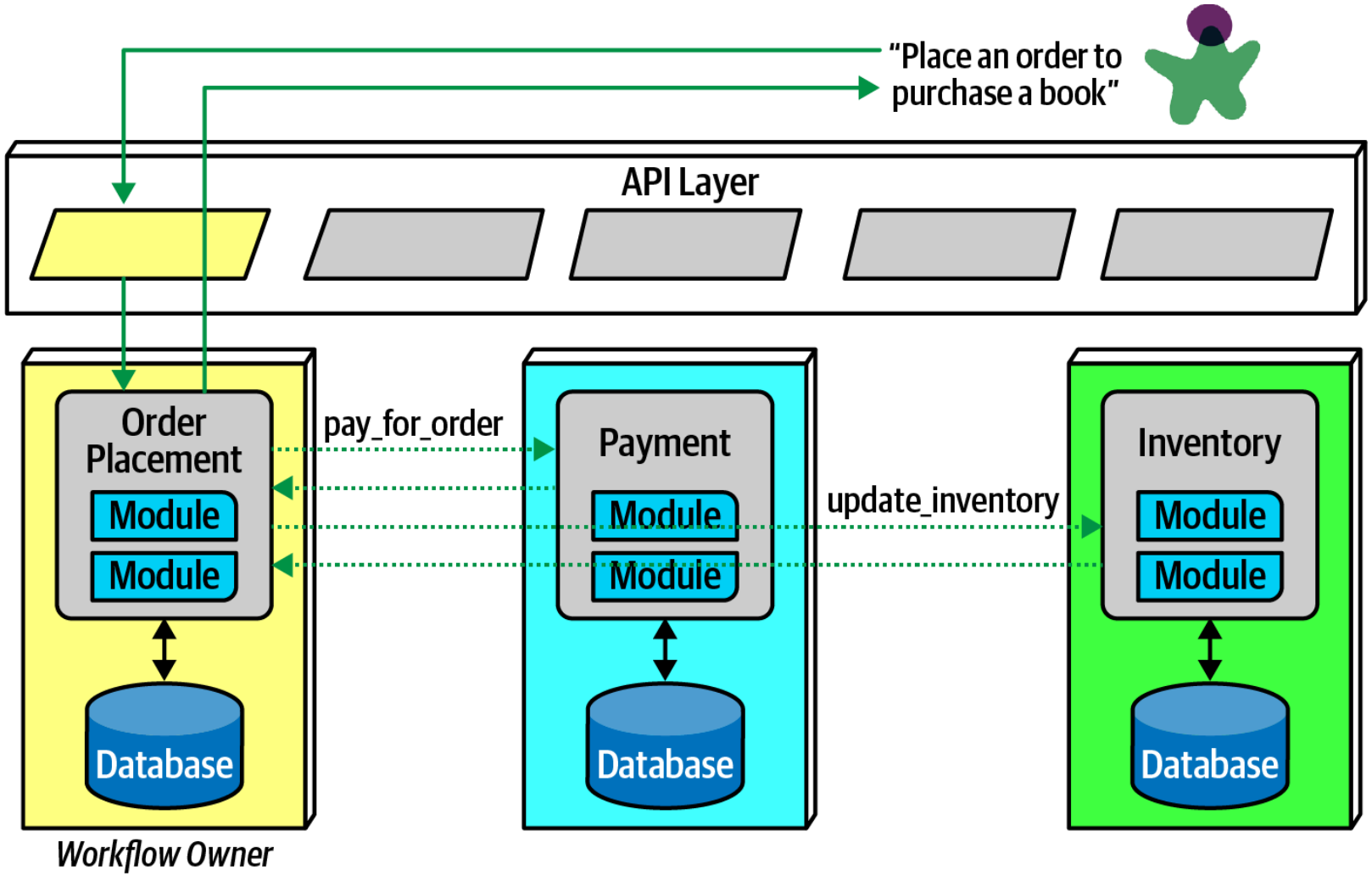


Fig. Using choreography for a complex business process

Architecture Patterns

Microservices Architecture (tt)

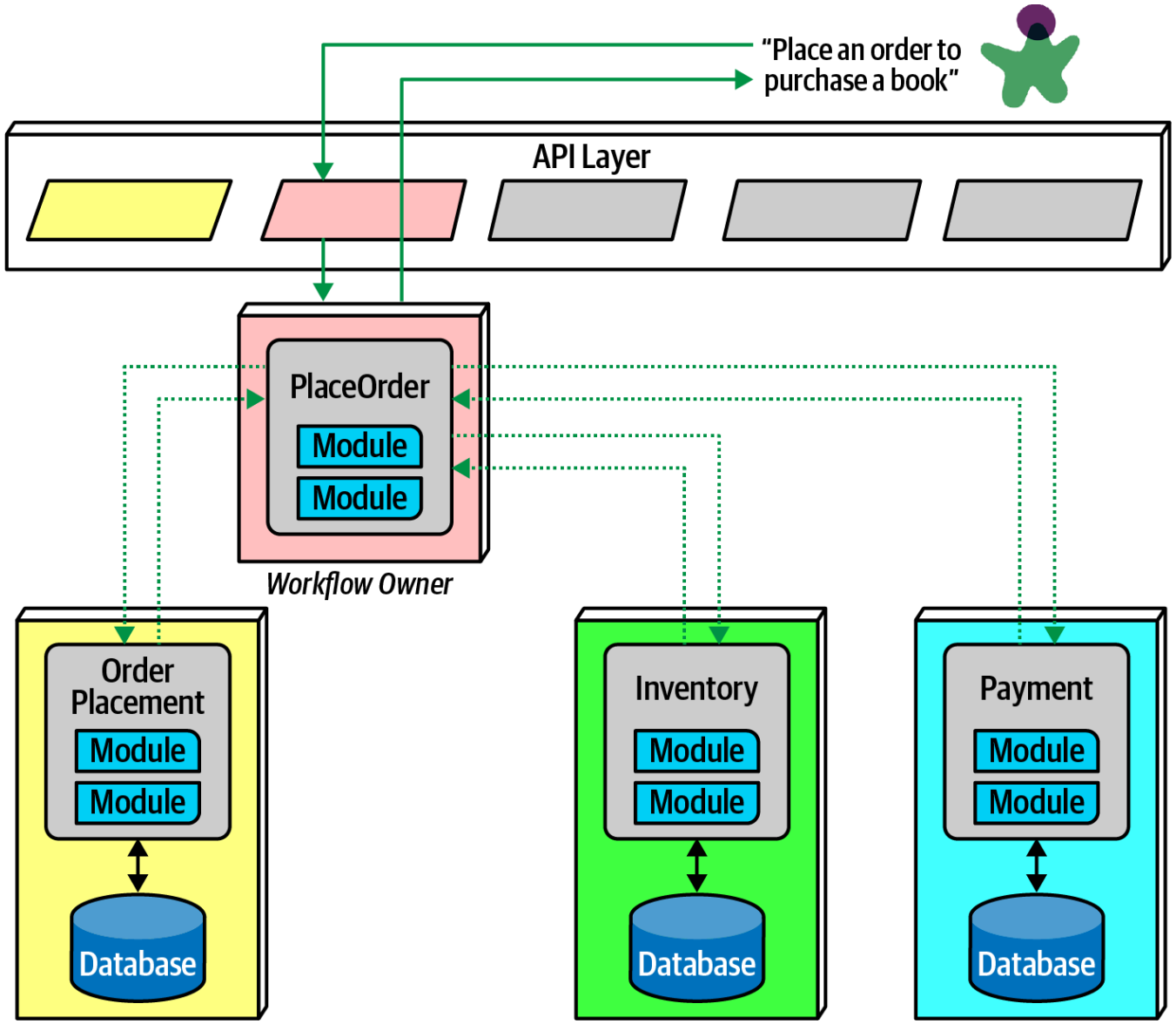


Fig. Using orchestration for a complex business process

Architecture Patterns

Microservices Architecture (tt)

Trường hợp nên dùng kiến trúc microservice:

- Kiến trúc microservice sẽ phù hợp với các ứng dụng có thể chia chức năng thành hàng chục hoặc hàng trăm phần riêng biệt và độc lập với nhau.
- Phù hợp khi ứng dụng của đòi hỏi sự phản ứng nhanh với thay đổi. Khi ứng dụng được chia thành các thành phần riêng biệt, việc thay đổi mã nguồn ít gây ảnh hưởng tới cả hệ thống. Việc testing cũng trở nên dễ dàng hơn vì phạm vi test được giới hạn.
- Nếu project đang có kế hoạch mở rộng rất nhiều thành phần trong hệ thống hiện tại, microservice cũng đáp ứng tốt. Trong vài trường hợp, việc tạo service mới chỉ bao gồm viết code tạo API, đóng gói trong container và deploy lên môi trường phù hợp. Các service khác chỉ việc gọi tới đúng endpoint.

Architecture Patterns

Microservices Architecture (tt)

Trường hợp không nên dùng kiến trúc microservice:

- Nếu bản chất dữ liệu được liên kết chặt chẽ và không thể chia dữ liệu ra thành các database chạy độc lập (Ví dụ: foreign key constraints, triggers, views, và stored procedures).
- Phải có nguồn lực đủ mạnh về con người, thời gian và tài chính. Vậy nên, nếu tổ chức đang chạy dự án với ngân sách tài chính hoặc con người hoặc cả hai đều eo hẹp, áp dụng microservice sẽ khiến mọi thứ trở nên rất phức tạp.
- Các service kết nối từ xa nên sẽ có độ trễ bao gồm độ trễ mạng, độ trễ bảo mật, độ trễ dữ liệu sẽ là một điểm cần được cân nhắc khi áp dụng microservice.

Q&A