

MICROSERVICES Architecture

PHAN THỊ TƯỜNG VI
NGUYỄN TRUNG QUÂN

Trường Đại học Khoa Học Tự Nhiên - Khoa Công Nghệ Thông Tin
2024-2025

NỘI DUNG



01.

Về FTGO

02.

Tổng quan

03.

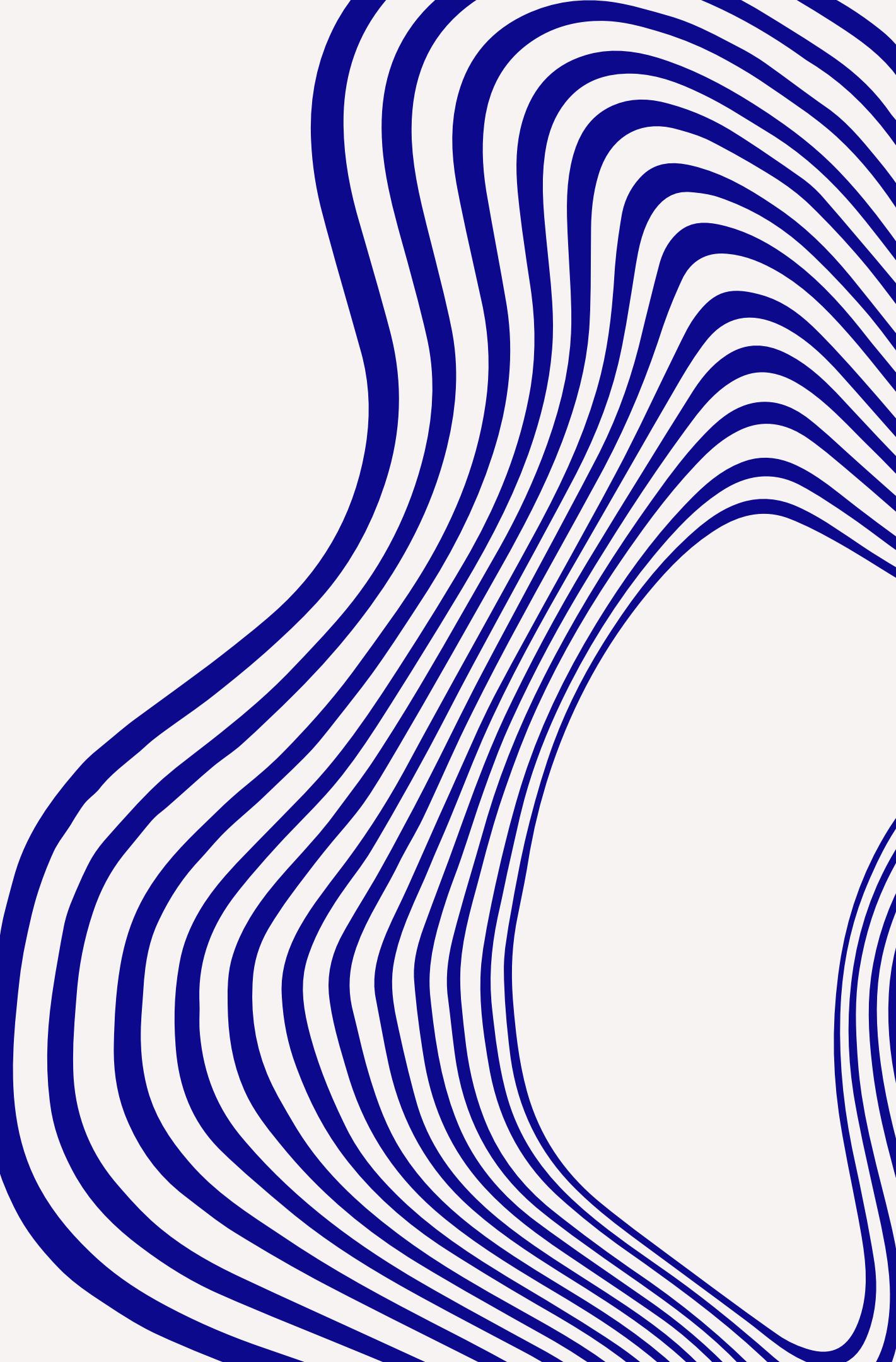
Decomposition
Pattern

04.

Data Consistency
Pattern

05.

Deployment
Pattern



01.

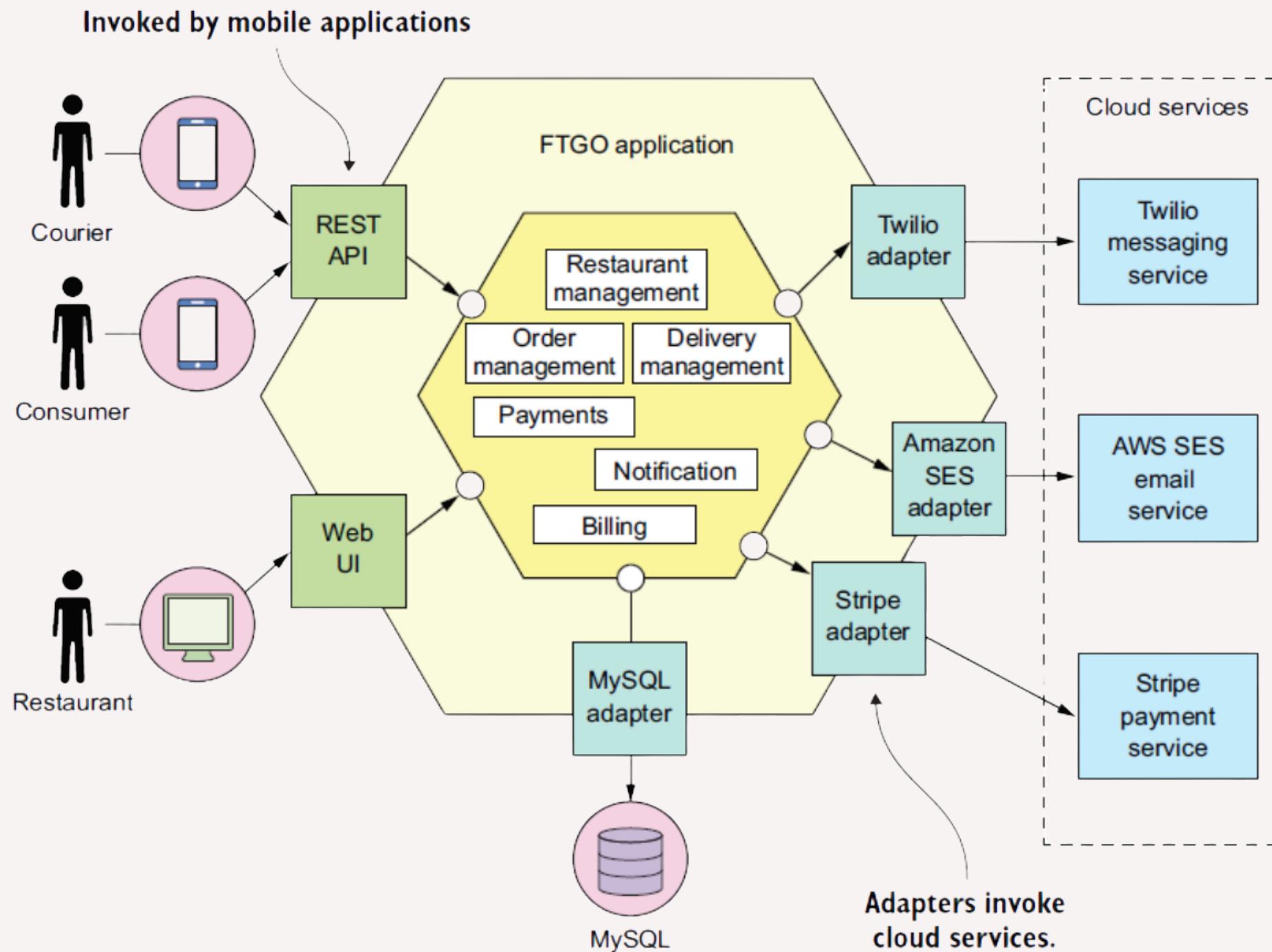
Food To Go (FTGO)



Food To Go (FTGO) là một ứng dụng giao đồ ăn ra mắt năm 2005, từng là một trong những công ty hàng đầu tại Mỹ. Ban đầu, FTGO hoạt động đơn giản: người dùng đặt món qua web hoặc app, hệ thống tìm tài xế giao hàng. Dù nhanh chóng mở rộng ra nước ngoài, kế hoạch bị đình trệ do bộ phận phát triển không đáp ứng được yêu cầu từ kinh doanh. FTGO có thể xem là tiền thân của các ứng dụng như **Grab**, **ShopeeFood** hiện nay.



Vấn đề của Food To Go (FTGO)



FTGO ban đầu **sử dụng kiến trúc monolith**, gói gọn trong một file Java WAR. Theo thời gian, hệ thống trở nên phức tạp, dễ lỗi và chậm phát hành.

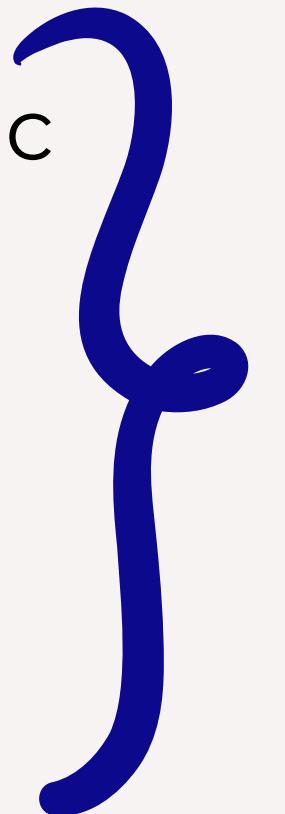
Ứng dụng áp dụng **hexagon architecture (DDD)**, với business logic trung tâm và các adapter kết nối UI (Web/Mobile) hoặc dịch vụ bên ngoài.

Ban đầu, monolith giúp phát triển nhanh, nhưng khi mở rộng, mô hình này trở thành rào cản, buộc FTGO phải chuyển sang Microservices.

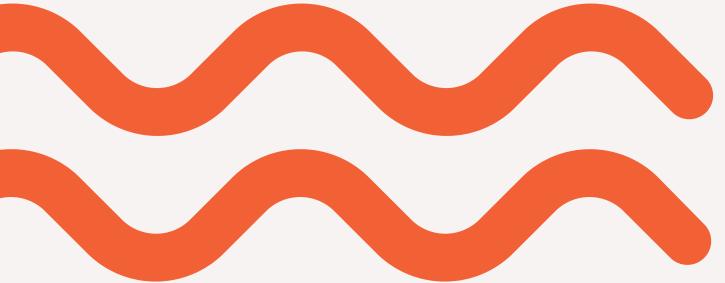
Đặc điểm của

Food To Go (FTGO)

- Khó mở rộng
- Khó đáp ứng nhiều khách hàng cùng lúc
- Chi phí cao
- Dễ sập toàn hệ thống:
- Khó thay đổi & cập nhật tech stack.
- Codebase lớn, chậm phát triển
- Mâu thuẫn code



**Cần chuyển đổi sang
Microservice**



Lợi ích của Microservices

Food To Go (FTGO)

Dễ phát triển

Nhiều IDE và công cụ hỗ trợ tốt.

Dễ thay đổi

Có thể nhanh chóng chỉnh sửa database schema.

Dễ kiểm thử

Viết test cho API, UI (Selenium) dễ dàng.

Dễ deploy

Chỉ cần chạy WAR file trên server có Tomcat.

Dễ mở rộng ban đầu

Có thể chạy nhiều instance và dùng load balancer để cân bằng tải





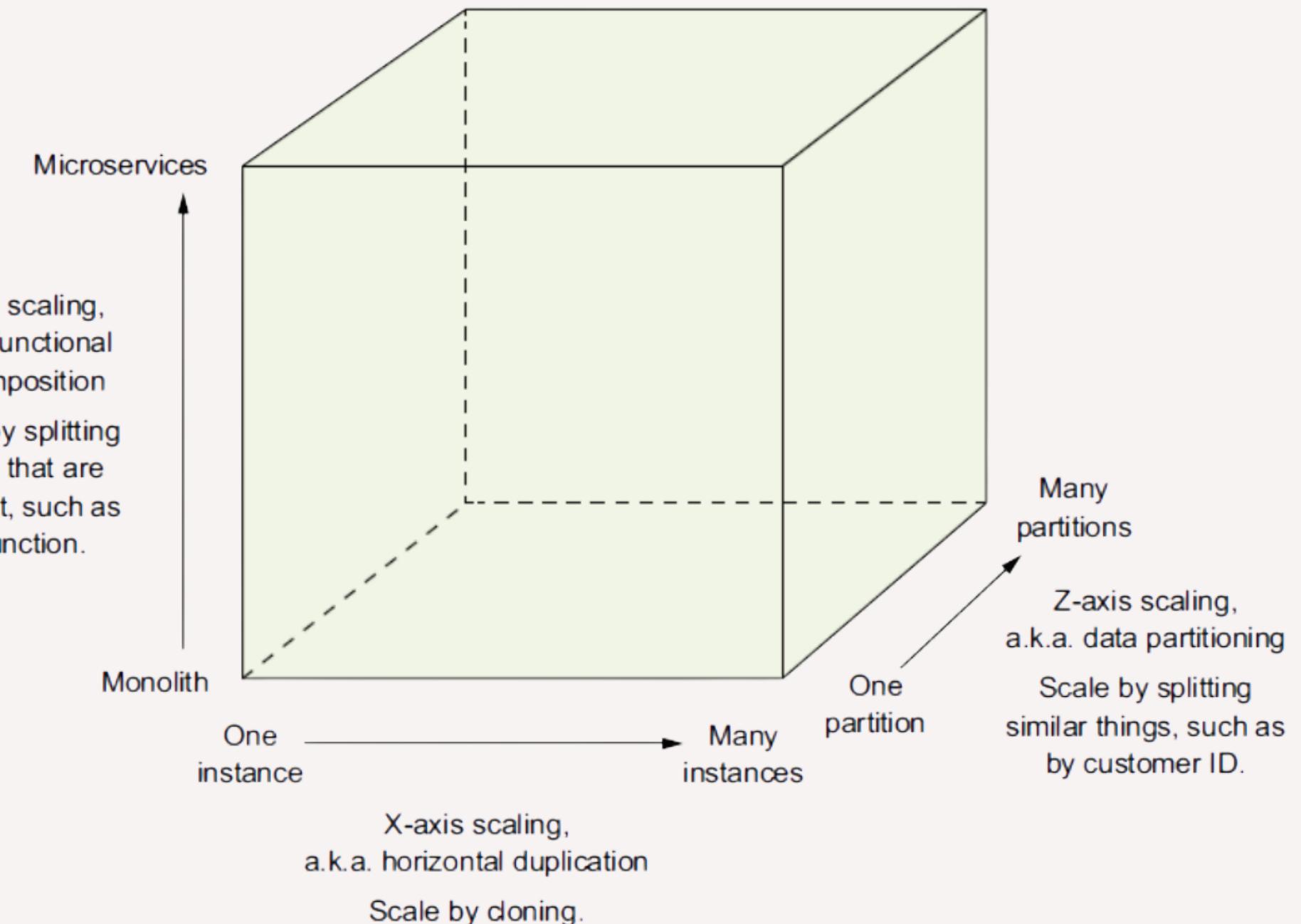
02.

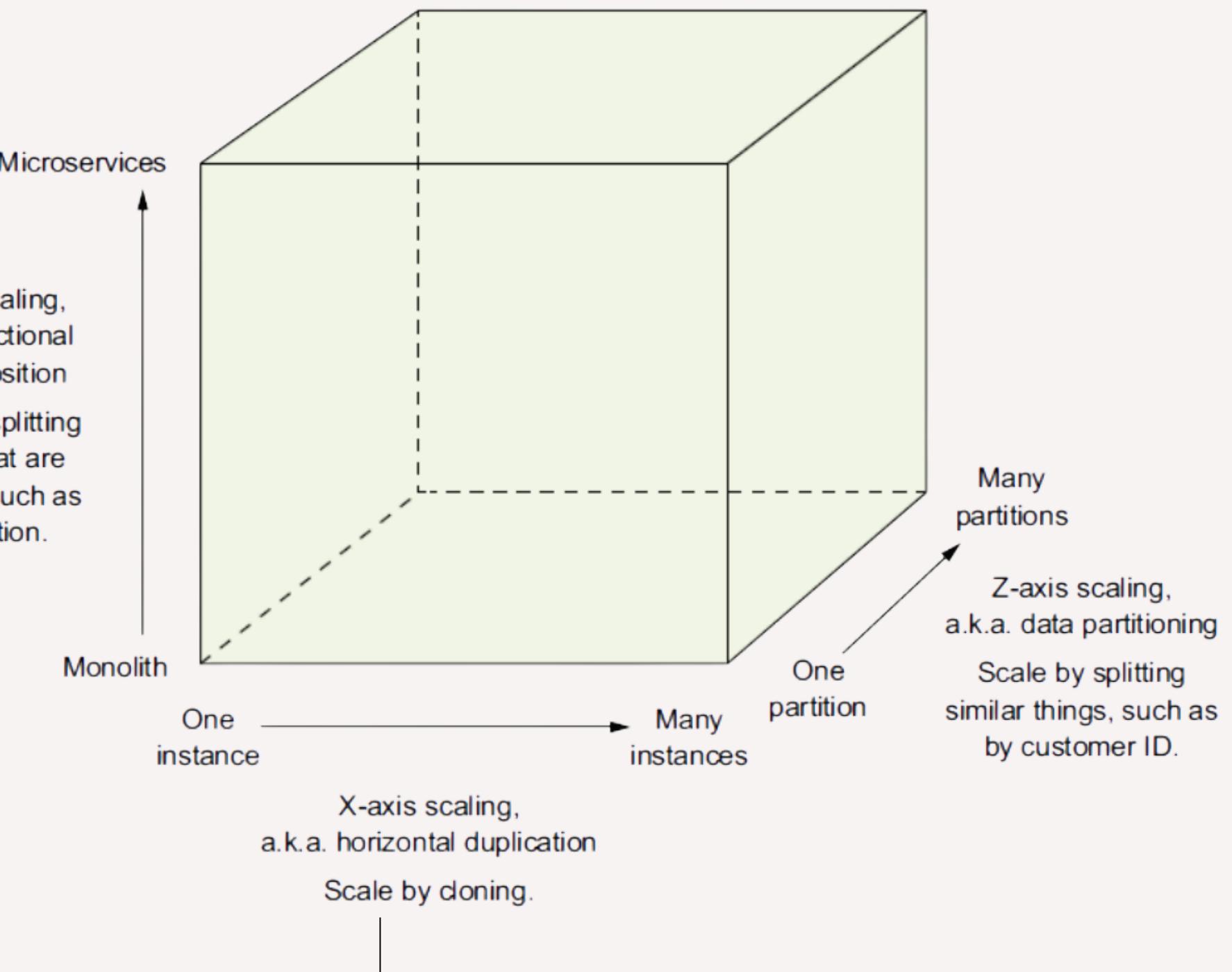
Overview

Thay vì cách tiếp cận **monolithic chậm chạp, phức tạp trong quá khứ**, developer và công ty ở khắp mọi nơi đang chuyển dịch sang kiến trúc microservice để đơn giản hóa và mở rộng cấu trúc. Trên thực tế, ngay cả các công ty như **Amazon, Netflix, Spotify và Uber** cũng đã thực hiện quá trình chuyển dịch này.

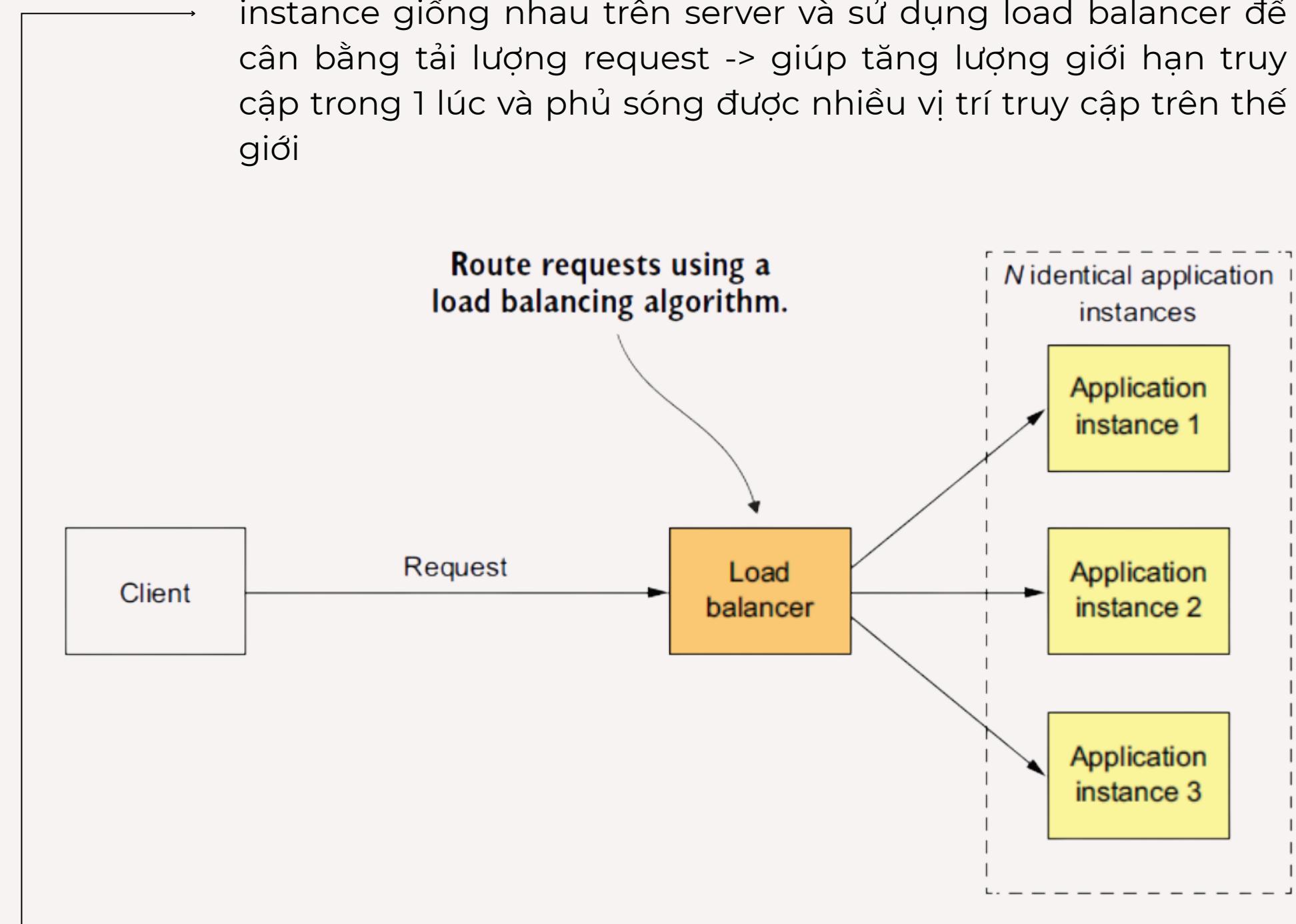
Định nghĩa

Microservices là kiến trúc phần mềm **phân chia ứng dụng lớn thành các dịch vụ nhỏ, độc lập**. Mỗi microservice đảm nhận **một chức năng cụ thể**, có thể phát triển, **triển khai và mở rộng riêng biệt**, thay vì gói gọn trong một hệ thống đồng nhất.

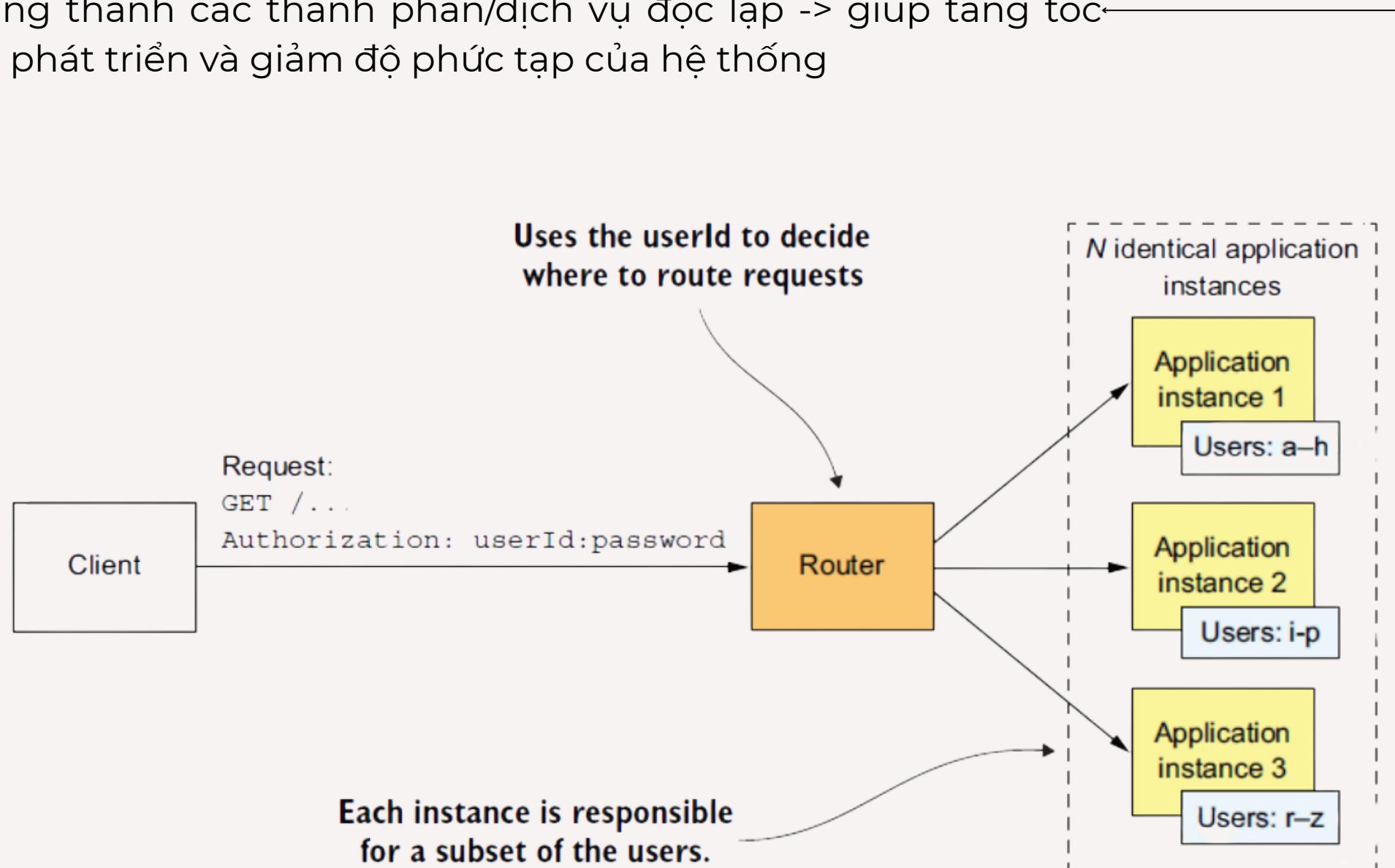




Trục X: thể hiện khả năng scale bằng việc tạo ra nhiều instance giống nhau trên server và sử dụng load balancer để cân bằng tải lượng request -> giúp tăng lượng giới hạn truy cập trong 1 lúc và phủ sóng được nhiều vị trí truy cập trên thế giới



Trục Y: thể hiện khả năng scale bằng việc phân tách nhỏ ứng dụng thành các thành phần/dịch vụ độc lập -> giúp tăng tốc độ phát triển và giảm độ phức tạp của hệ thống

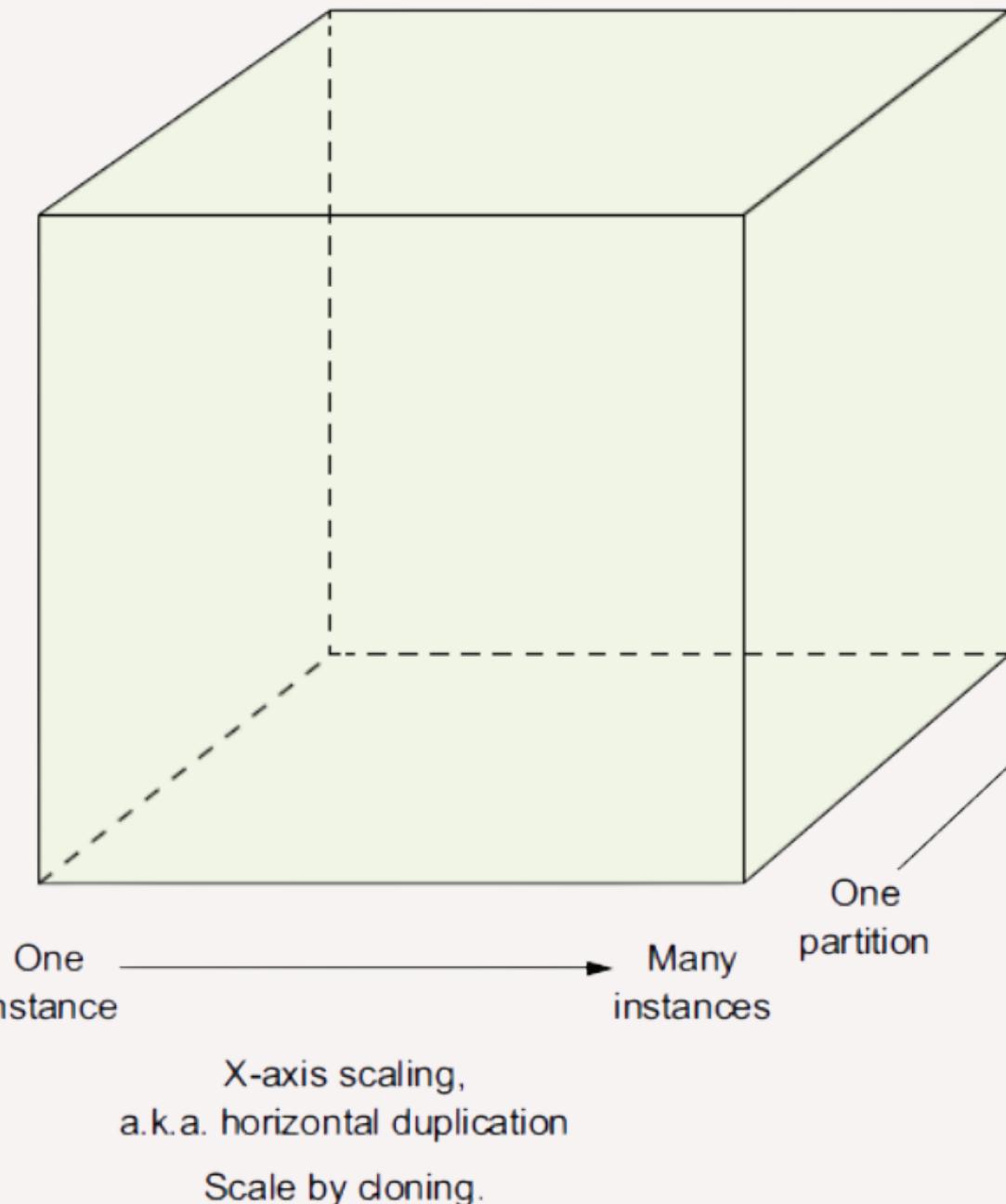


Microservices

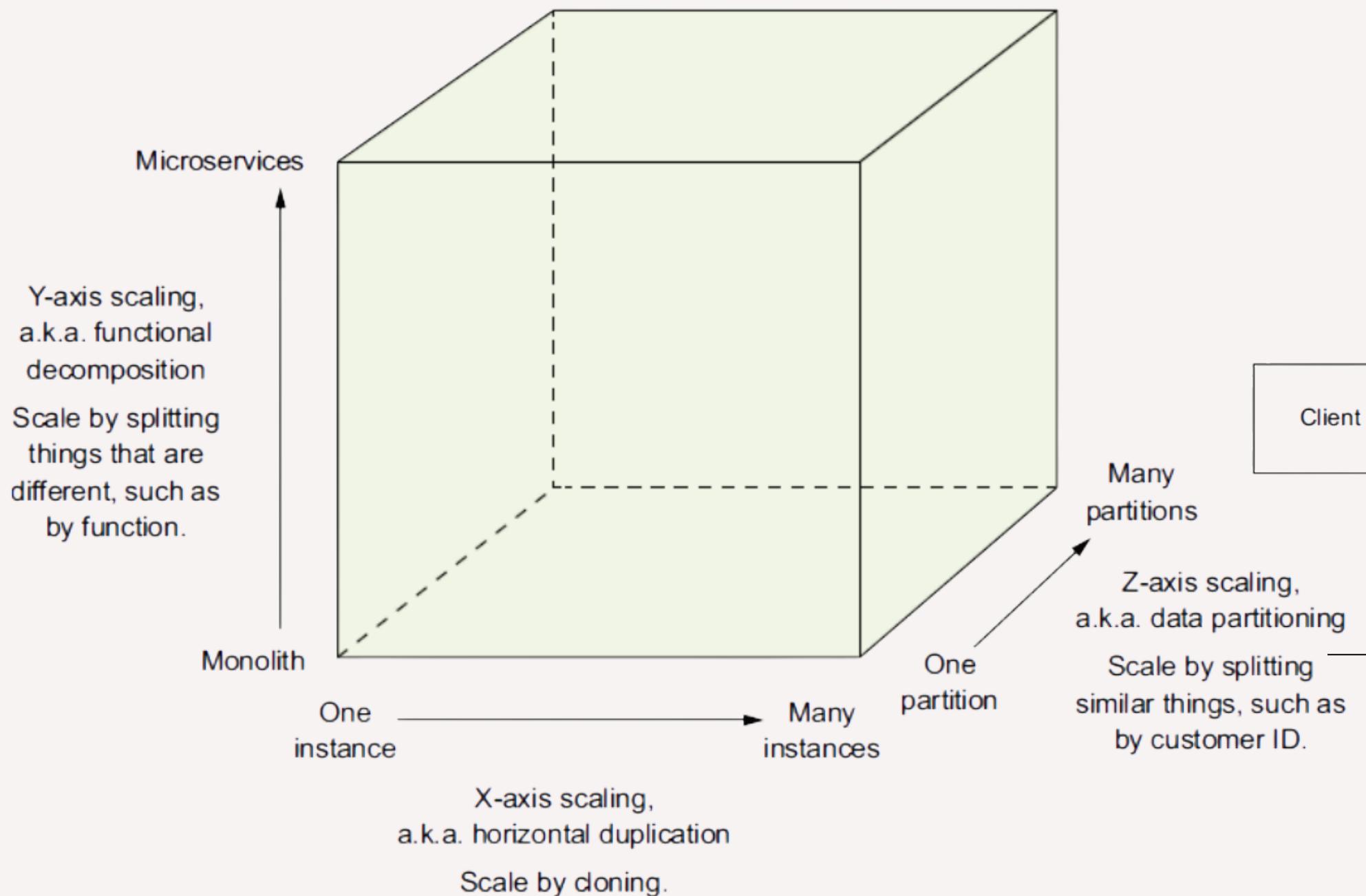
Y-axis scaling, a.k.a. functional decomposition

Scale by splitting things that are different, such as by function.

Monolith



Overview



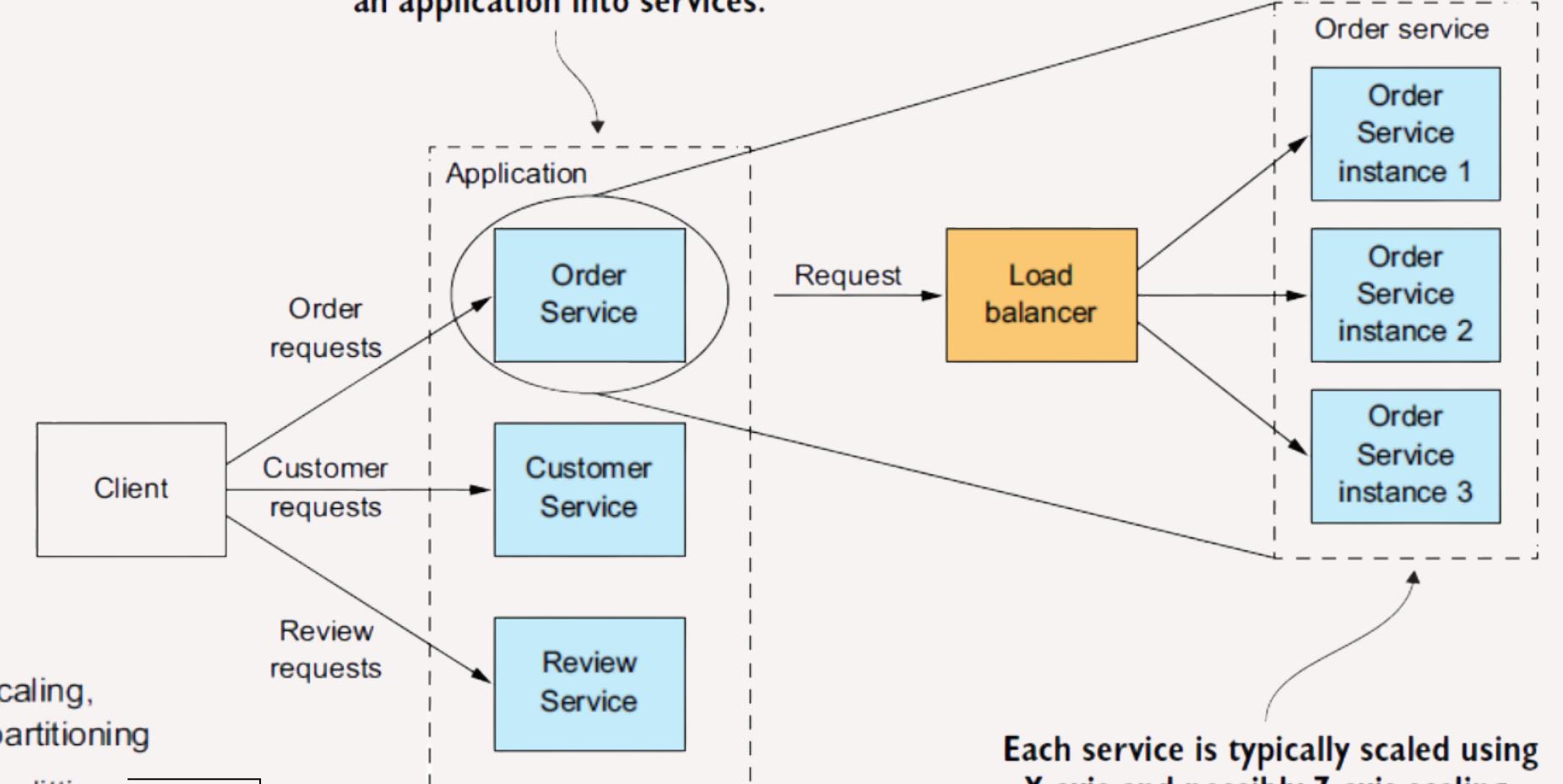
Decomposition Pattern

Data Consistency Pattern

Deployment Pattern



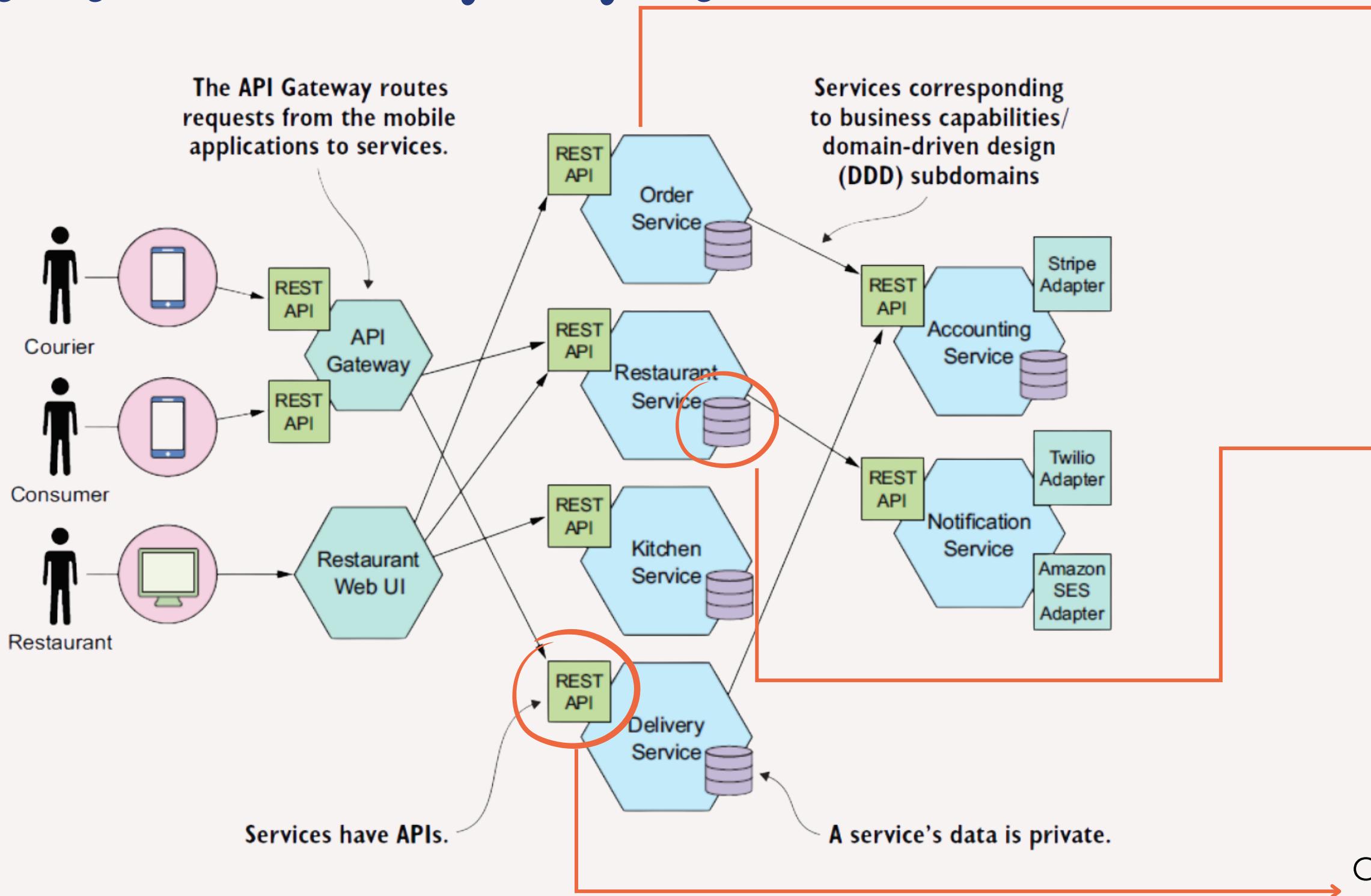
Y-axis scaling functionality decomposes an application into services.



Each service is typically scaled using X-axis and possibly Z-axis scaling.

Trục Z: thể hiện khả năng scale bằng việc tạo ra nhiều instance dựa trên tính chất đặc biệt của dữ liệu và sử dụng router để điều hướng request đến instance phù hợp -> giúp tăng giới hạn transaction và lượng truy cập dữ liệu trên database

Nguyên tắc hoạt động

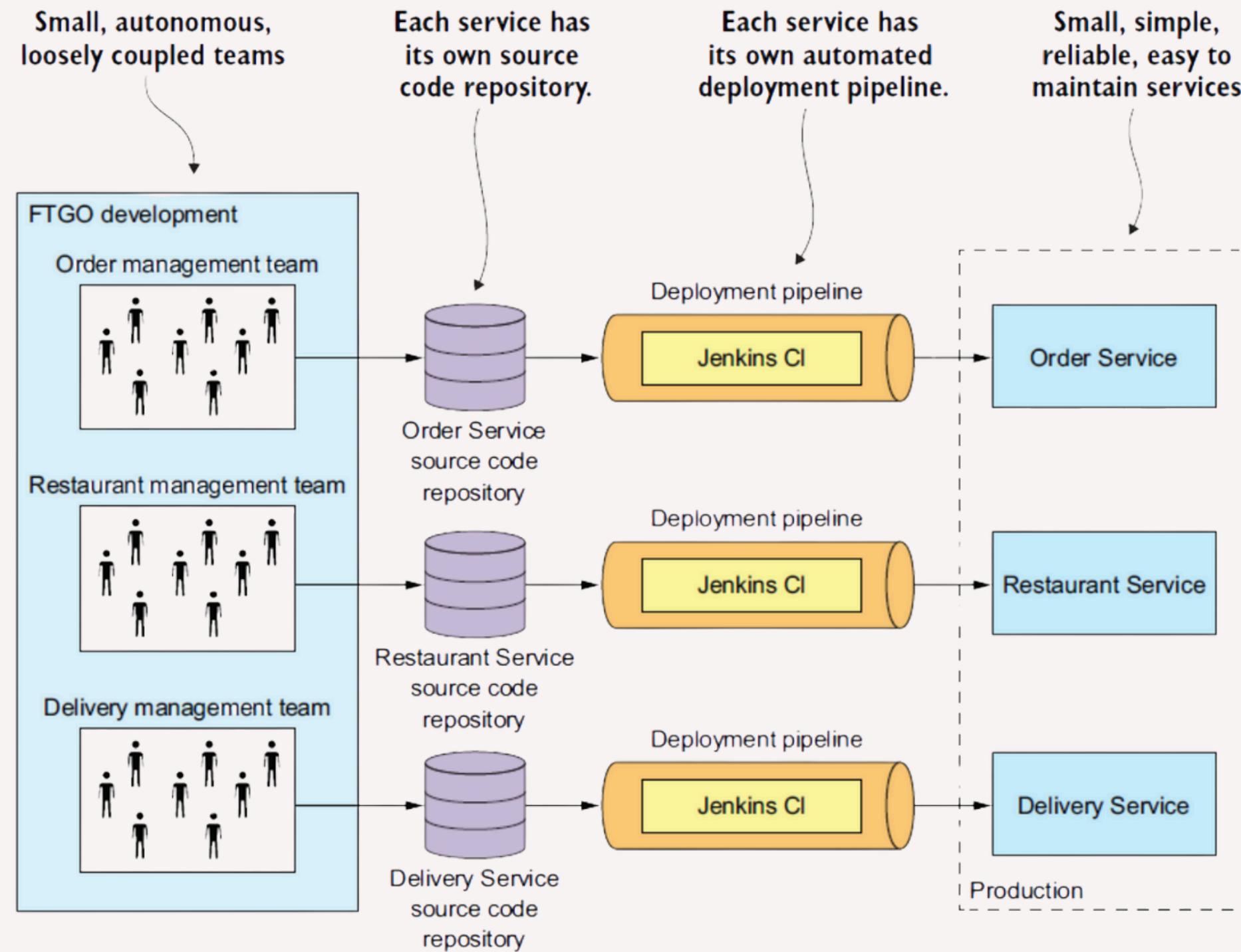


Ứng dụng được **chia thành nhiều dịch vụ nhỏ**, mỗi dịch vụ đảm nhận một nhiệm vụ

Một service sẽ có một database riêng, việc mỗi service có một database riêng sẽ giảm thiểu sự phụ thuộc của các service với nhau.

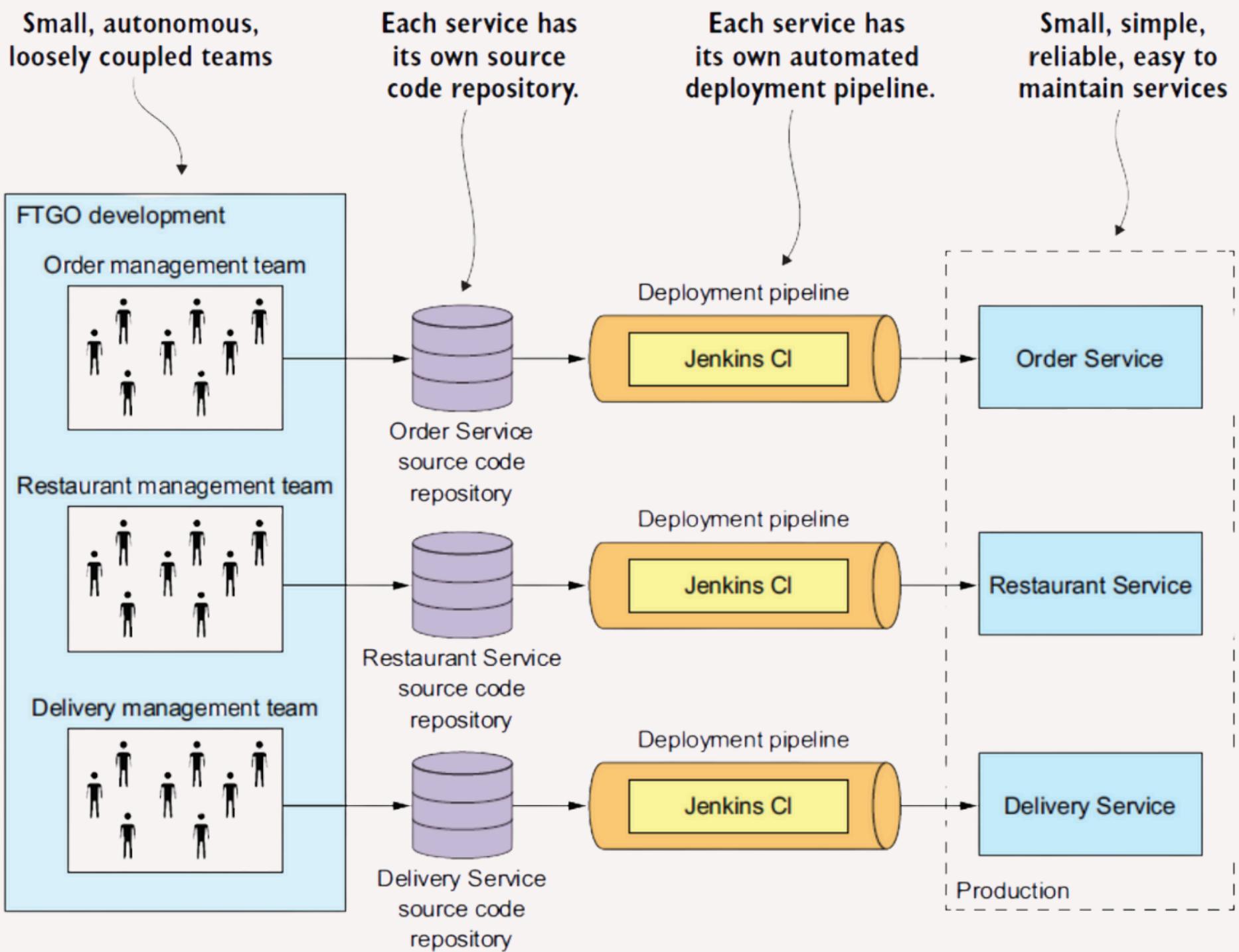
Giao tiếp với nhau **qua API** tiêu chuẩn

Ưu và Nhược điểm





Ưu và Nhược điểm



- **Dễ dàng phát triển & kiểm thử** nhờ tính mô-đun, tránh sự phức tạp của monolithic.
- **Tích hợp CI/CD toàn diện**
- Mỗi service có thể deploy và scale **độc lập**.
- **Đội nhóm hoạt động độc lập**, không phụ thuộc lẫn nhau.
- **Cập nhật & thay đổi công nghệ linh hoạt**, dễ nâng cấp lên phiên bản mới.
- **Ổn định & ít lỗi hơn**, giảm rủi ro crash trên môi trường khách hàng.
- **Tiết kiệm chi phí**, mỗi service chạy trên server phù hợp nhất.
- **Rút ngắn thời gian phát hành sản phẩm**, cải thiện tốc độ triển khai.
- **Hỗ trợ giám sát & xử lý lỗi hiệu quả**, giúp nhanh chóng khắc phục sự cố.

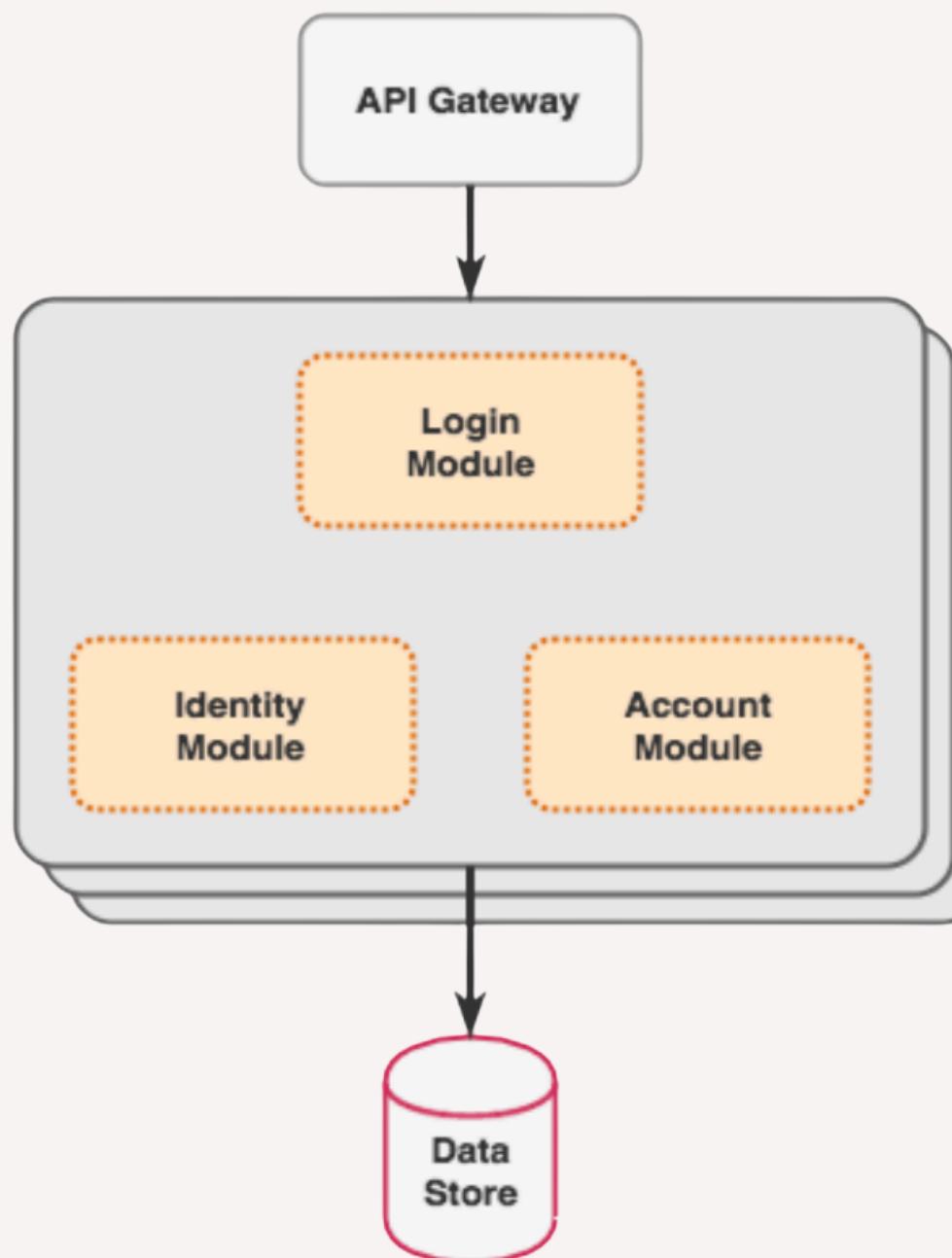


Ưu và Nhược điểm

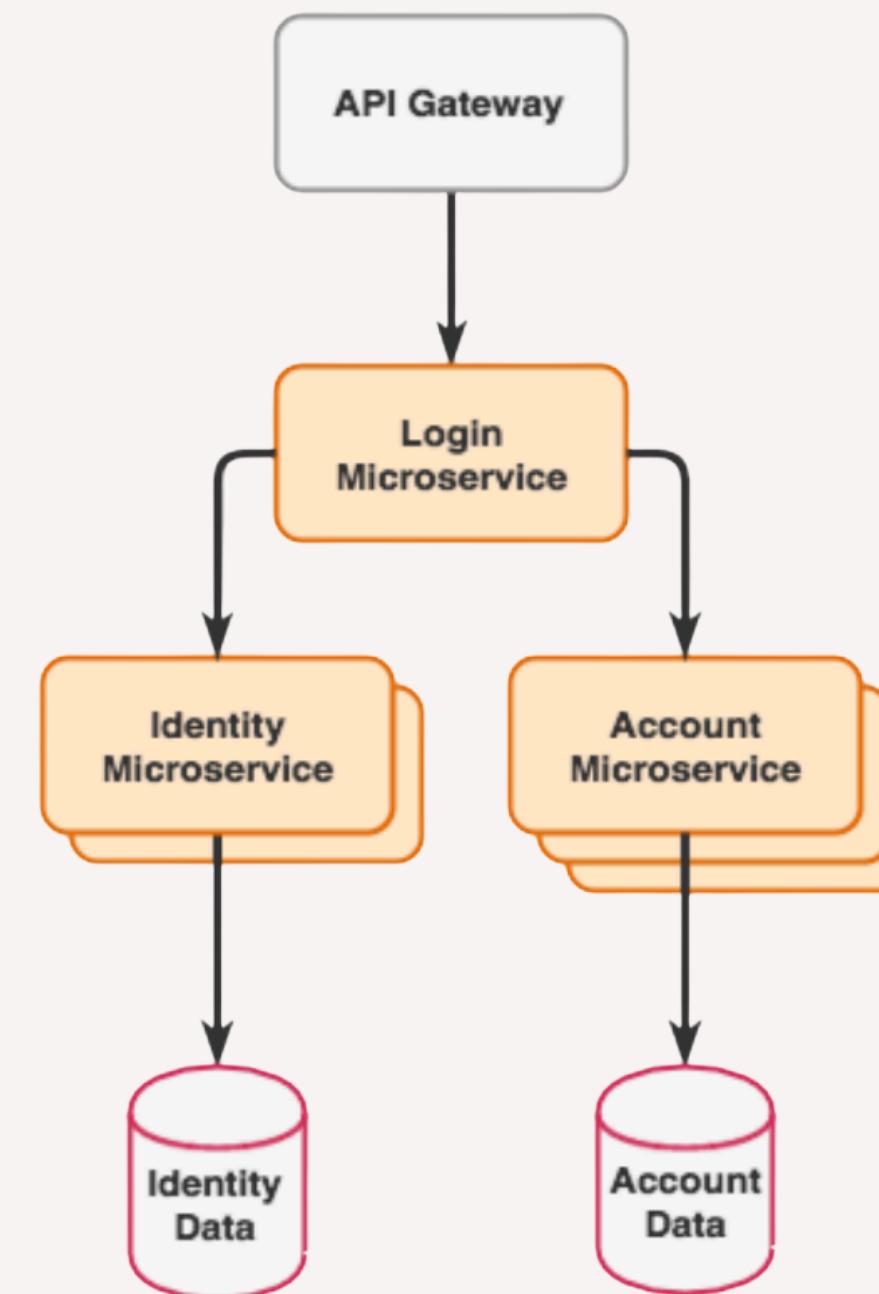
- **Xác định thành phần hệ thống:** Thiết kế microservices đòi hỏi hiểu biết sâu về business và kinh nghiệm. Nếu phân tách không hợp lý, có thể dẫn đến distributed monolith, mất đi lợi thế của microservices.
- **Test độc lập:** Mỗi microservice cần được test riêng lẻ và tích hợp, nhưng các công cụ phát triển thường không hỗ trợ tốt, gây khó khăn cho QA và dev.
- **Yêu cầu kỹ năng cao:**
 1. Dev cần hiểu về giao tiếp giữa các service và đảm bảo tính sẵn sàng khi có lỗi xảy ra.
 2. Do mỗi service có database riêng, cần hiểu về sagas để đảm bảo dữ liệu nhất quán.
 3. Cần biết về API composition hoặc CQRS views để truy vấn dữ liệu hiệu quả.
- **Thời điểm chuyển đổi:** Với startup, áp dụng microservices sớm có thể gây tổn kém và làm chậm tiến độ. Cần xác định đúng thời điểm chuyển từ monolith sang microservices.
- **Triển khai nhiều service:**
 - Cần công cụ tự động hóa deploy và giám sát.
 - Phải xác định thứ tự deploy để tránh lỗi do dependencies giữa các service.

Monolithic v/s Microservices

Monolithic Application



Microservices Architecture



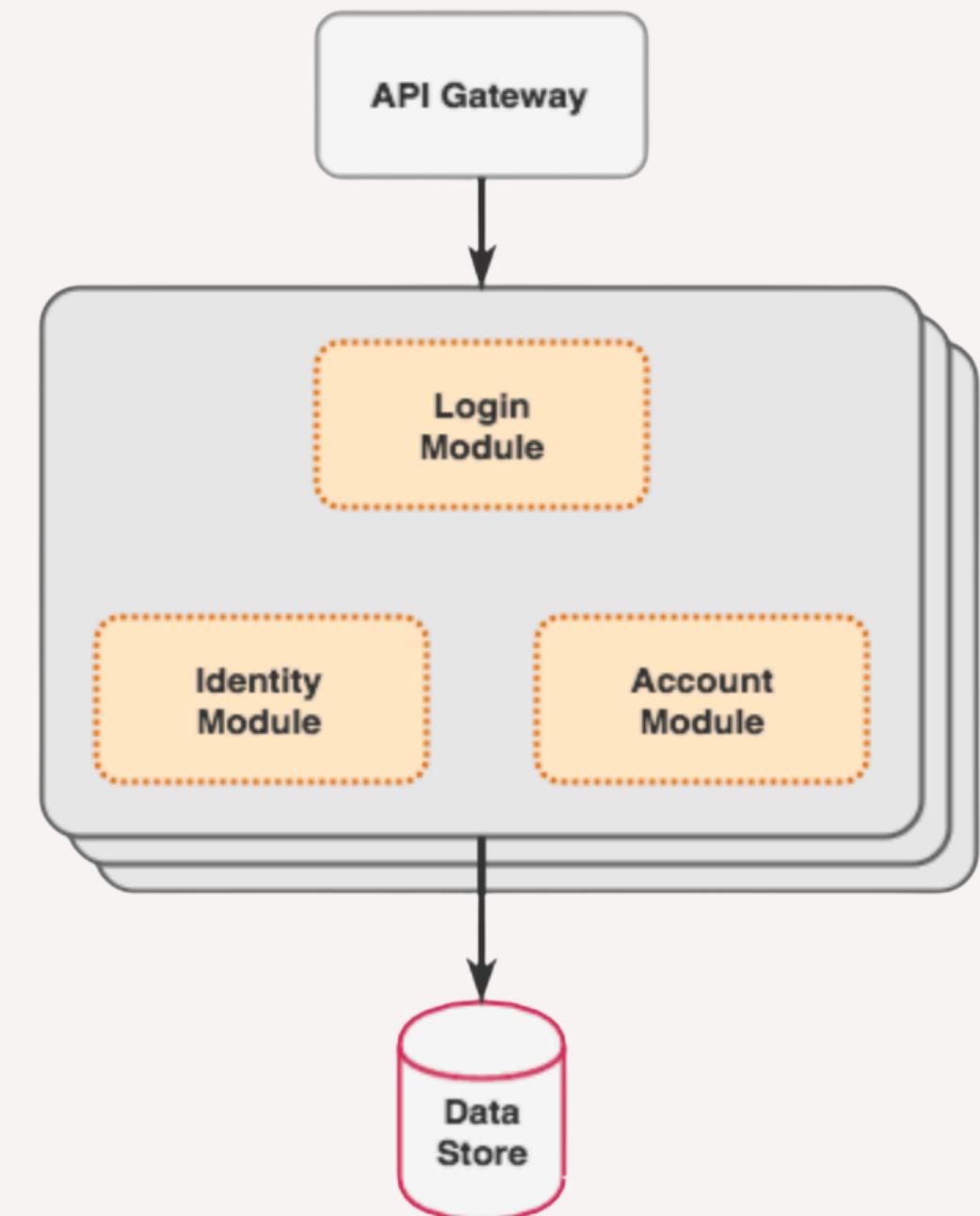
Monolithic v/s Microservices

- **Kiến trúc truyền thống**, mọi thành phần (server, client, database) nằm **chung trong một codebase duy nhất**.
- Mọi thay đổi ảnh hưởng **toàn hệ thống**, khó mở rộng.
- **Scale phức tạp**, làm chậm quá trình phát triển.
- **Vòng đời phát triển dài hơn**, khó duy trì và nâng cấp

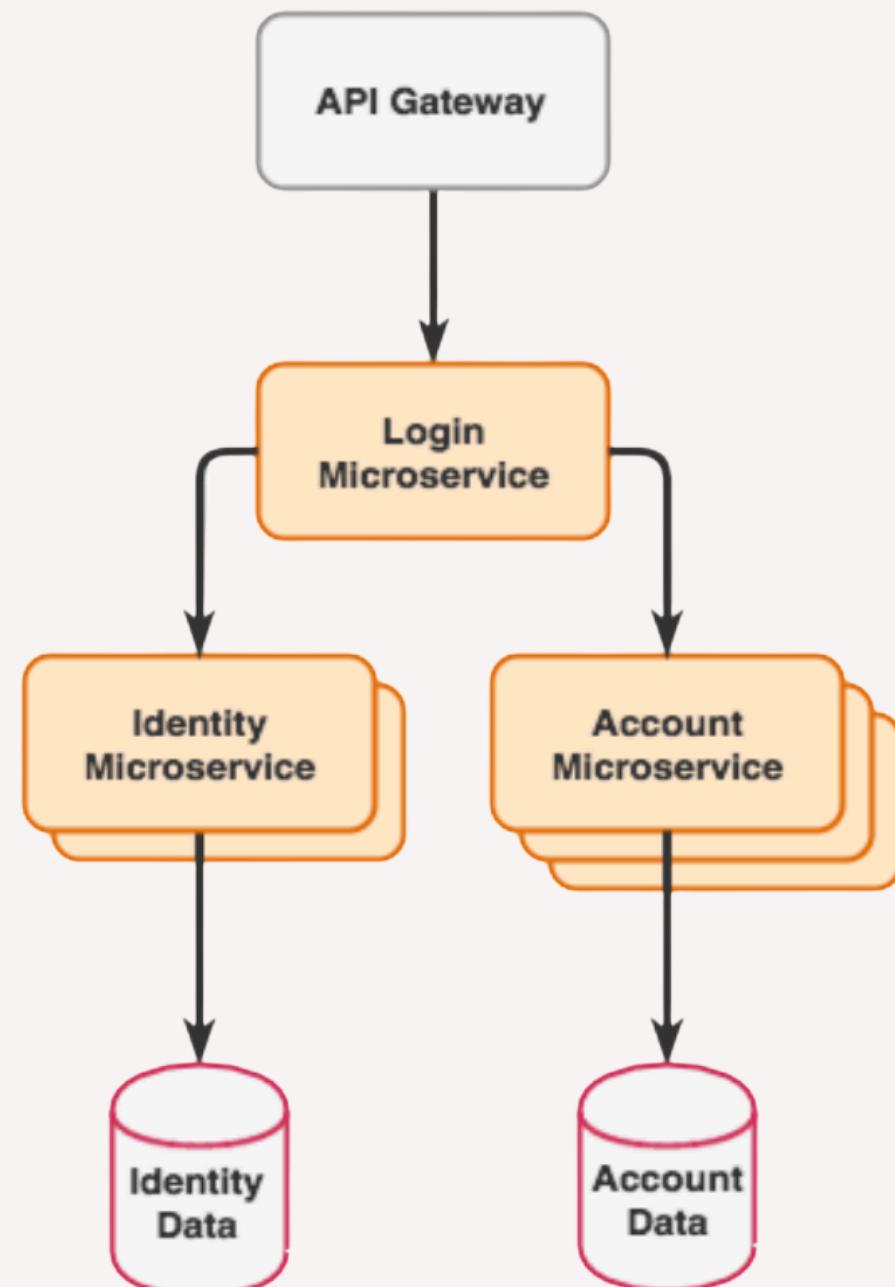
Chọn khi:

- **Nếu công ty bạn là một team nhỏ**, bằng cách này thì tránh được phức tạp khi deploy microservice.
- **Nếu bạn muốn triển khai nhanh**. Hệ thống sẽ cần thêm thời gian để cập nhật về sau, nhưng khởi đầu thì nó phát triển nhanh hơn.

Monolithic Application



Microservices Architecture

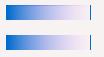


Monolithic v/s Microservices

- Kiến trúc microservice chia nhỏ thể duy nhất ấy thành các đơn vị độc lập có chức năng như những service riêng biệt.
- **Mỗi service có logic và codebase riêng.**
- Chúng liên lạc với nhau **through API**

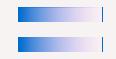
Chọn khi:

- **Nếu bạn muốn phát triển ứng dụng có khả năng scale.** Scale ứng dụng microservice dễ hơn monolithic rất nhiều. Tính năng và module mới được thêm vào một cách đơn giản và nhanh gọn.
- **Nếu công ty lớn hoặc có kế hoạch mở rộng quy mô.**



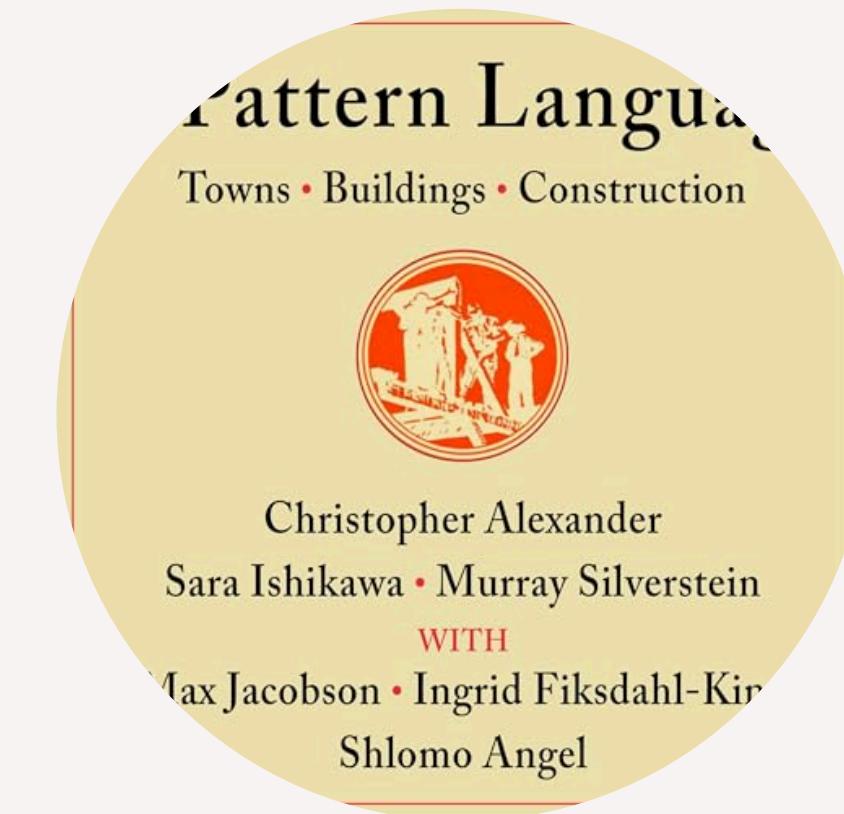
SOA v/s Microservices

	SOA	Microservices
Inter-service communication	Smart pipes, such as Enterprise Service Bus, using heavyweight protocols, such as SOAP and the other WS* standards.	Dumb pipes, such as a message broker, or direct service-to-service communication, using lightweight protocols such as REST or gRPC
Data	Global data model and shared databases	Data model and database per service
Typical service	Larger monolithic application	Smaller service



Pattern Language

Pattern là một giải pháp có thể tái sử dụng cho cùng một vấn đề trong một trường hợp cụ thể. Design patterns được thiết kế bởi Four of Gang cũng chính là một trong những Pattern Language đầu tiên trong thiết kế phần mềm. **Pattern language là một giải pháp ở mức cao hơn design pattern, khi tập trung vào việc thiết kế và giao tiếp giữa các service với nhau.**



1 Mục tiêu

Liệt kê ra những vấn đề cần phải giải quyết của ứng dụng

2 Kết quả

Ưu và nhược điểm của pattern, sau khi áp dụng thì nảy sinh những vấn đề nào

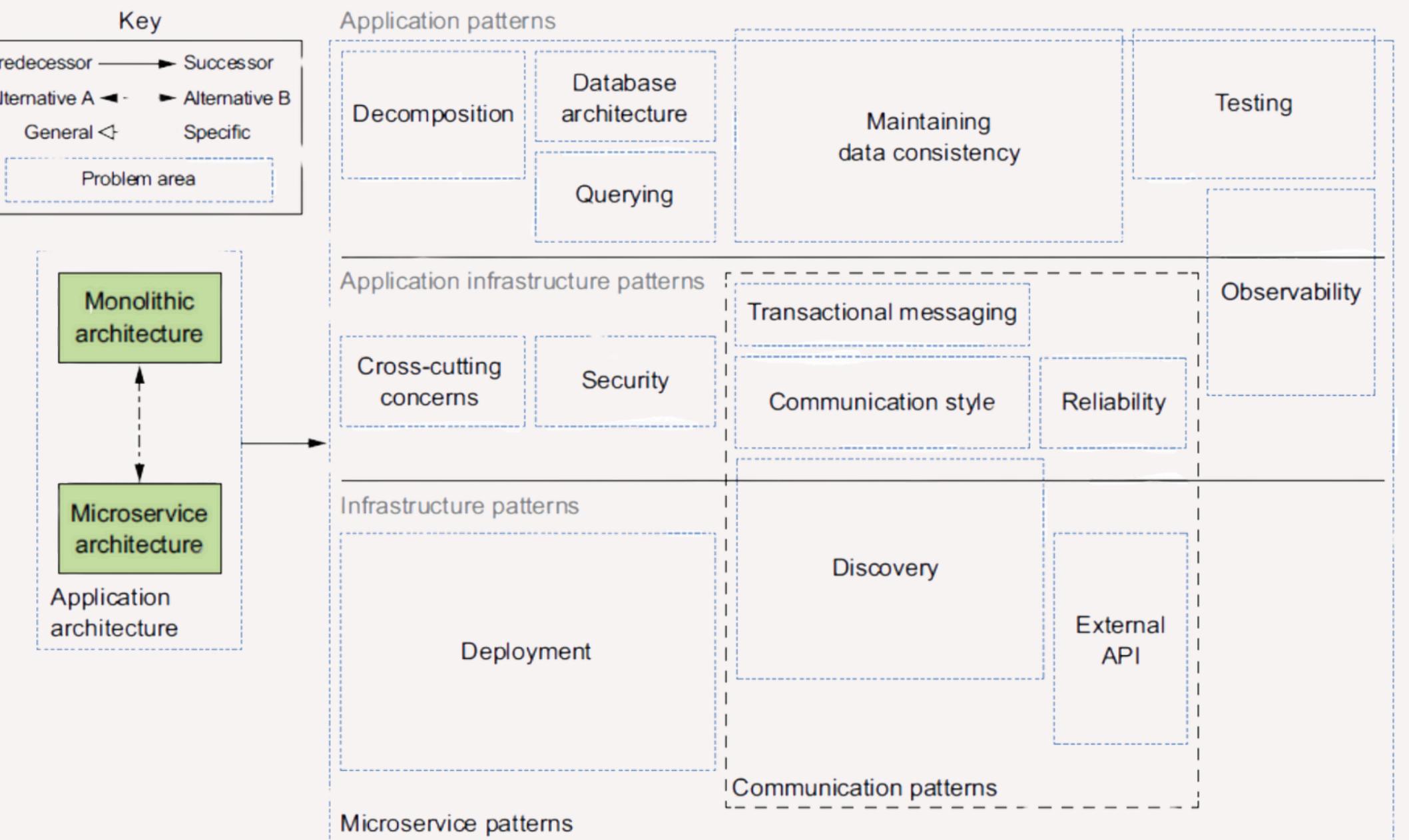
3 Các pattern kết hợp cùng

Các pattern khi được kết hợp với nhau sẽ bao gồm 5 mối quan hệ:

1. Predecessor
2. Successor
3. Alternative
4. Generalization
5. Specialization



Pattern Language



Application patterns: Giải quyết các vấn đề chỉ liên quan đến quá trình phát triển

Application infrastructure patterns: Giải quyết các vấn đề vừa liên quan đến hạ tầng, vừa liên quan đến quá trình phát triển

Infrastructure patterns: Giải quyết các vấn đề liên quan đến hạ tầng, thường không liên quan đến quá trình phát triển



9 pattern bao gồm:

1

Decomposition patterns

Phân tách hệ thống thành các service nhỏ theo chức năng hoặc dữ liệu để dễ quản lý.

2

Communication patterns

Xác định cách các service giao tiếp (REST, gRPC, Message Broker).

3

Data consistency patterns

Đảm bảo tính nhất quán dữ liệu giữa các service (Sagas, Event Sourcing).

4

Querying data patterns

Truy vấn dữ liệu từ nhiều service hiệu quả (API Composition, CQRS).

5

Service deployment patterns

Triển khai service linh hoạt

6

Observability patterns

Giám sát hiệu suất và lỗi hệ thống (Logging, Distributed Tracing, Metrics).

7

Automated testing patterns

Kiểm thử tự động các service

8

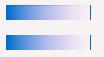
Cross-cutting patterns

Quản lý cấu hình chung của các service

9

Security patterns

Đảm bảo an toàn hệ thống (OAuth, JWT, Zero Trust Security).



9 pattern bao gồm:

1

Decomposition patterns

Phân tách hệ thống thành các service nhỏ theo chức năng hoặc dữ liệu để dễ quản lý.

2

Communication patterns

Xác định cách các service giao tiếp (REST, gRPC, Message Broker).

3

Data consistency patterns

Đảm bảo tính nhất quán dữ liệu giữa các service (Sagas, Event Sourcing).

4

Querying data patterns

Truy vấn dữ liệu từ nhiều service hiệu quả (API Composition, CQRS).

5

Service deployment patterns

Triển khai service linh hoạt

6

Observability patterns

Giám sát hiệu suất và lỗi hệ thống (Logging, Distributed Tracing, Metrics).

7

Automated testing patterns

Kiểm thử tự động các service

8

Cross-cutting patterns

Quản lý cấu hình chung của các service

9

Security patterns

Đảm bảo an toàn hệ thống (OAuth, JWT, Zero Trust Security).



03.

Decomposition Pattern

Khi cân nhắc chuyển một hệ thống từ monolith sang microservices, điều ta cần làm đầu tiên là **định nghĩa các service cần có trong hệ thống**. Quá trình này được gọi là **phân rã hệ thống thành các service**.

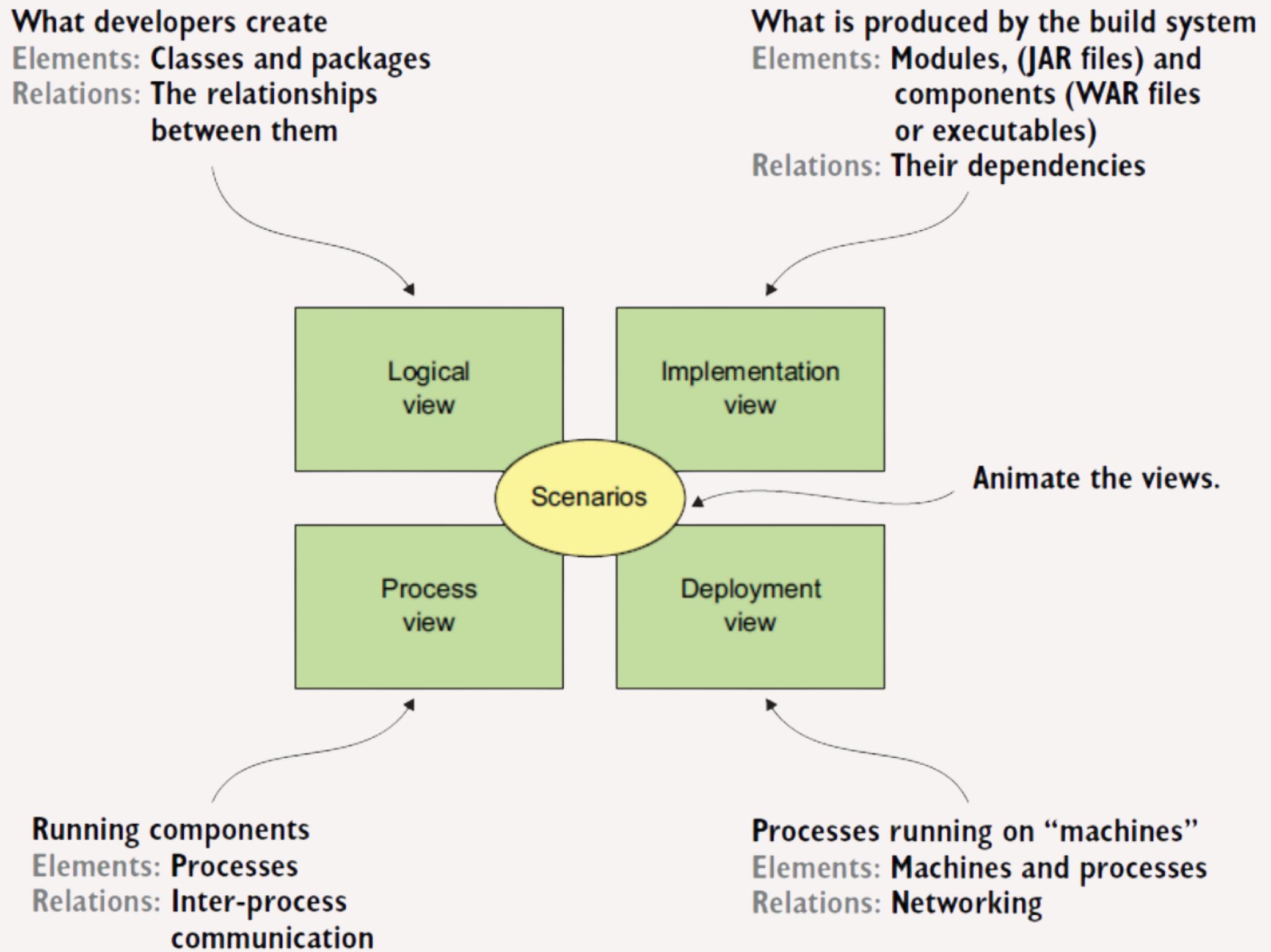
Kiến trúc hệ thống

1

Các thành phần trong hệ
thống

2

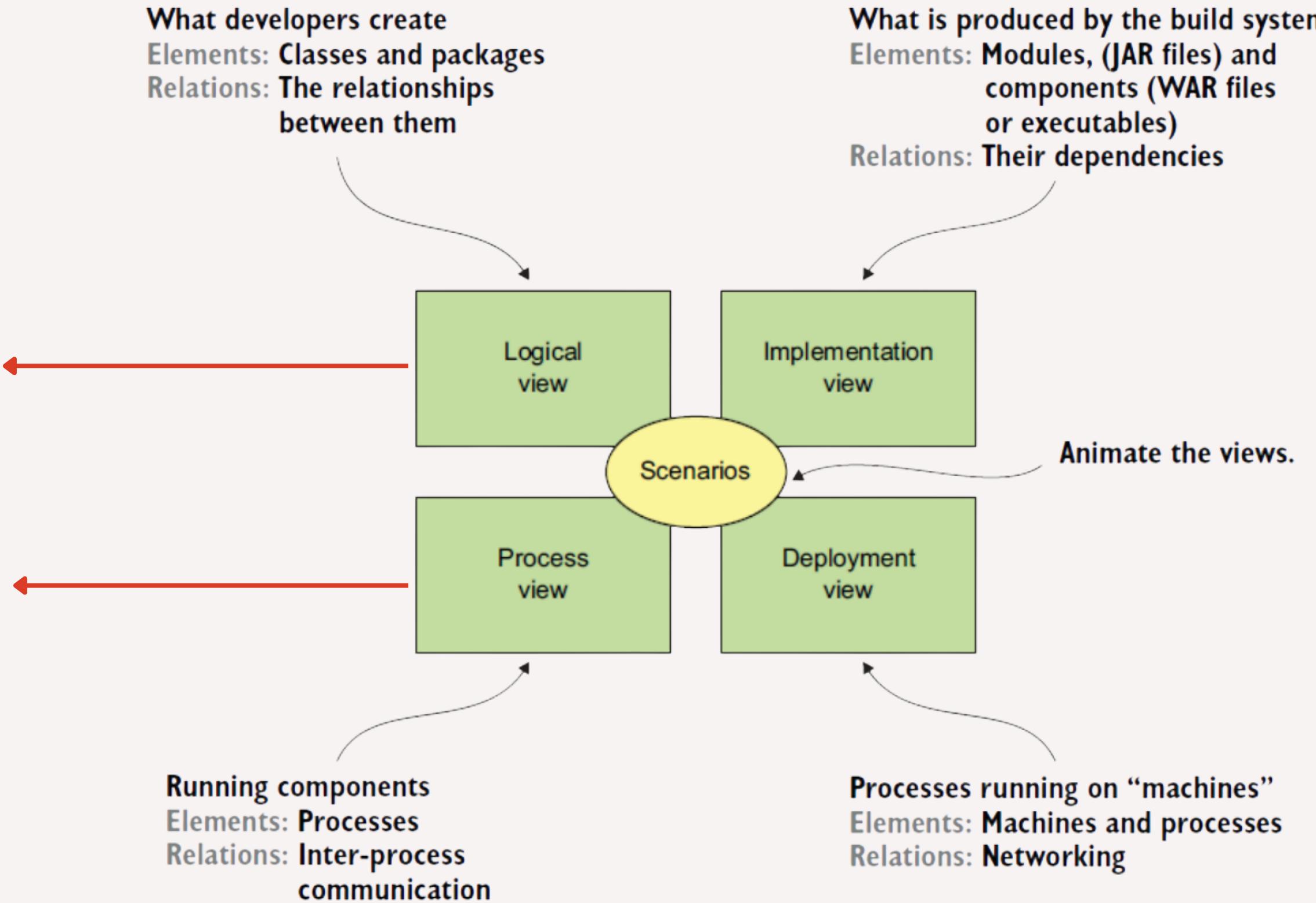
Mối quan hệ của các thành
phần với nhau trong hệ
thống





Được mô tả bằng ngôn ngữ hướng đối tượng, cấu trúc bên trong thường là **các class** cùng các **quan hệ như kế thừa, phụ thuộc hay tương tác với nhau**

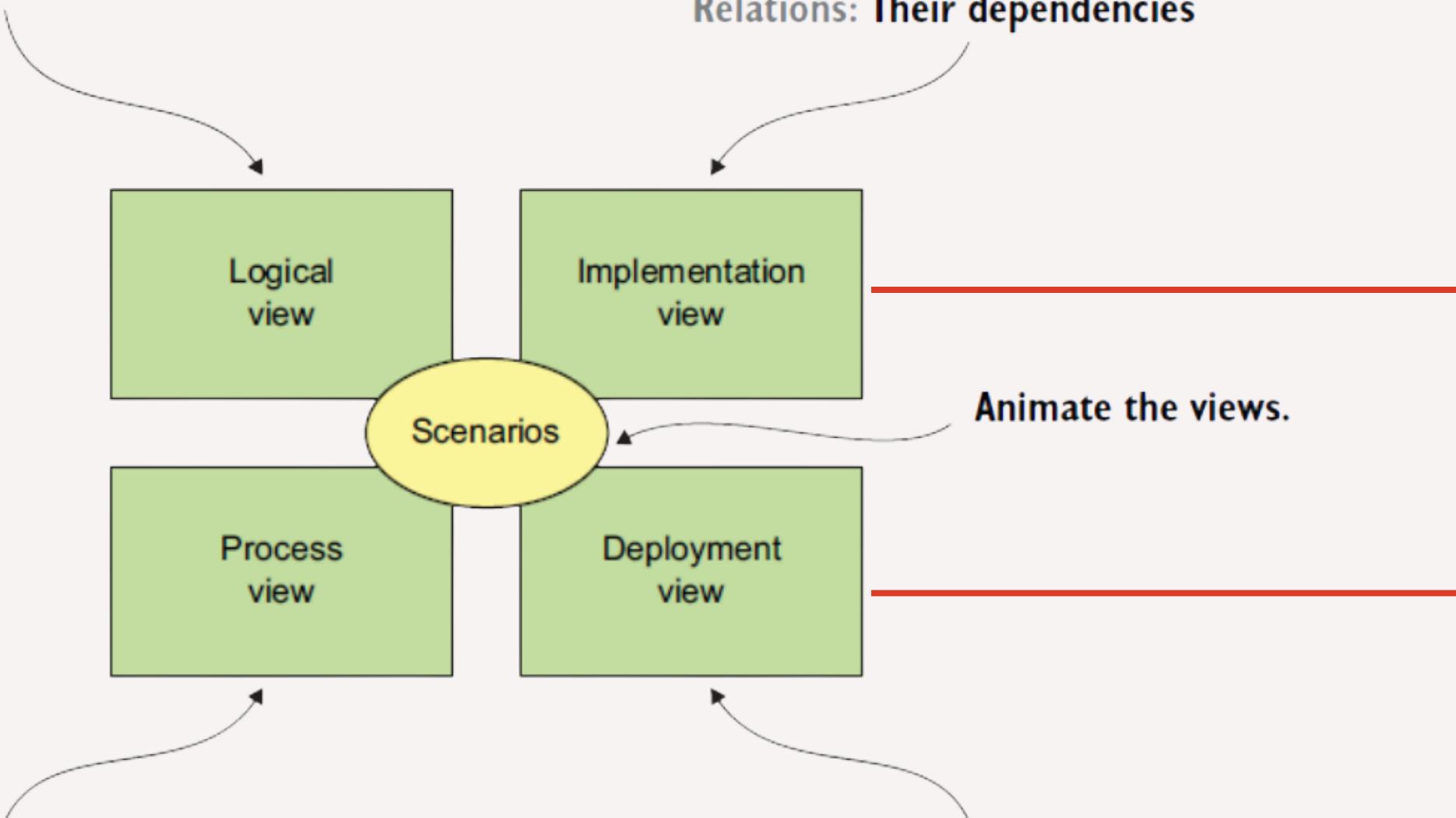
Được mô tả bằng **các tiến trình**, các tiến trình sẽ **giao tiếp với nhau bằng các phương pháp IPC khác nhau như REST API, gRPC, ...**



**What developers create**

Elements: Classes and packages

Relations: The relationships between them

**Running components**

Elements: Processes

Relations: Inter-process communication

Processes running on “machines”

Elements: Machines and processes

Relations: Networking

Được mô tả bằng các **package** và **module**. VD: Một hệ thống web có thể chia thành các package như core/, api/, ui/, mỗi package lại chứa các module con (ví dụ api/user, api/order).

Cách các thành phần phần mềm được triển khai lên phần cứng và mạng, **liên quan đến nhiều kiến trúc hạ tầng của phần mềm**



Nguyên tắc xây dựng Decomposition

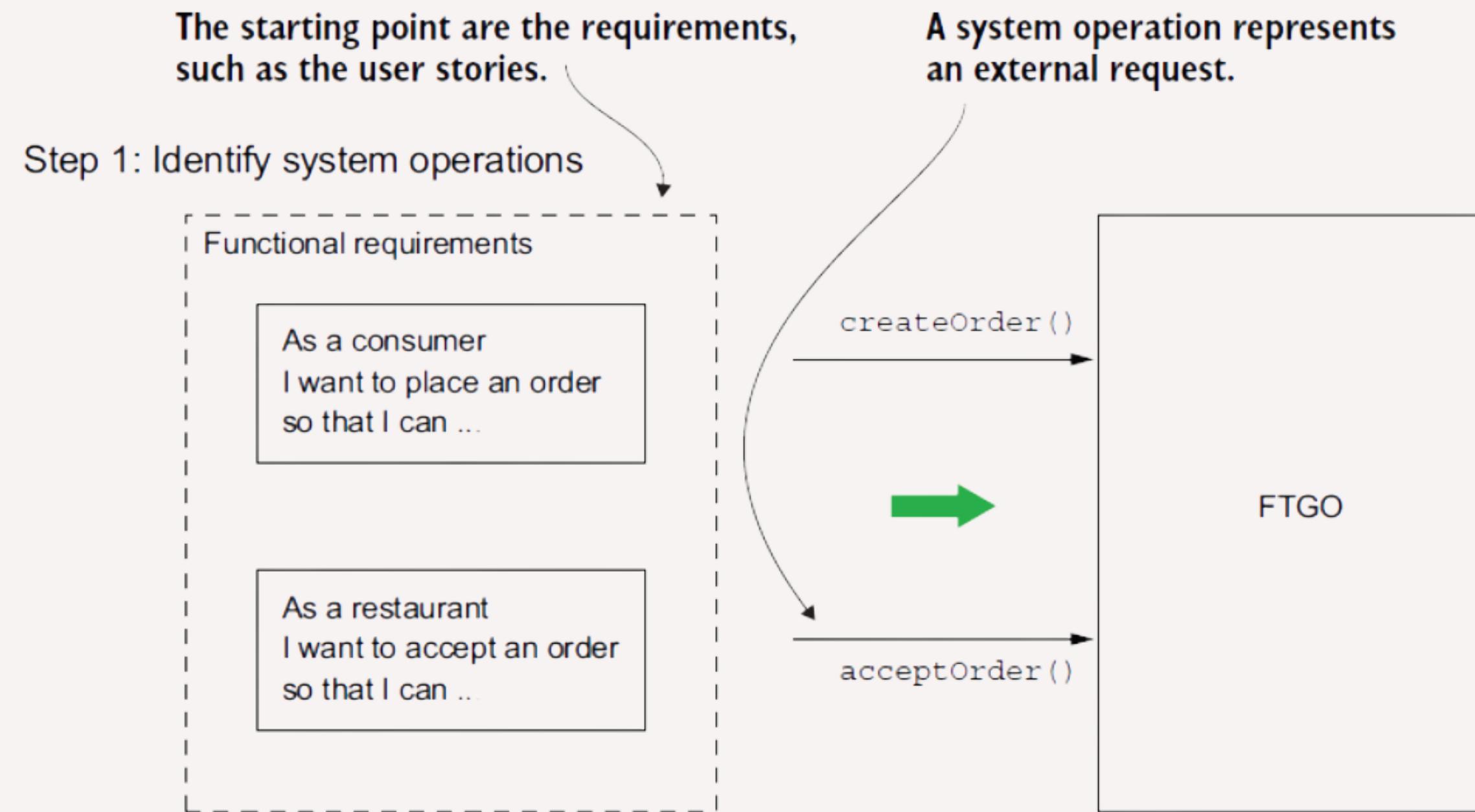
Loosely Coupling (Kết nối lỏng lẻo)

- Các **service giao tiếp qua API**, ẩn chi tiết cài đặt bên trong.
- Thông tin bên trong service **phải private, tránh phụ thuộc chéo** khi thay đổi schema.
- Không dùng chung database giữa các service giúp:
 - 1.Tránh lock truy cập,
 - 2.Tăng hiệu suất chạy,
 - 3.Nhưng cần giải pháp đảm bảo tính nhất quán dữ liệu.

Quy mô của một service

- “Micro” không chỉ kích thước nhỏ mà là:
 - 1.Có thể được phát triển bởi team 8-12 người độc lập.
 - 2.Nếu cần nhiều người hơn → **nên tách nhỏ hơn nữa.**
- Nếu một service thay đổi kéo theo service khác phải thay đổi → **vì phạm loosely coupling.**
- Nếu nhiều service xử lý cùng một quy tắc nghiệp vụ → **nên gom lại thành một service.**

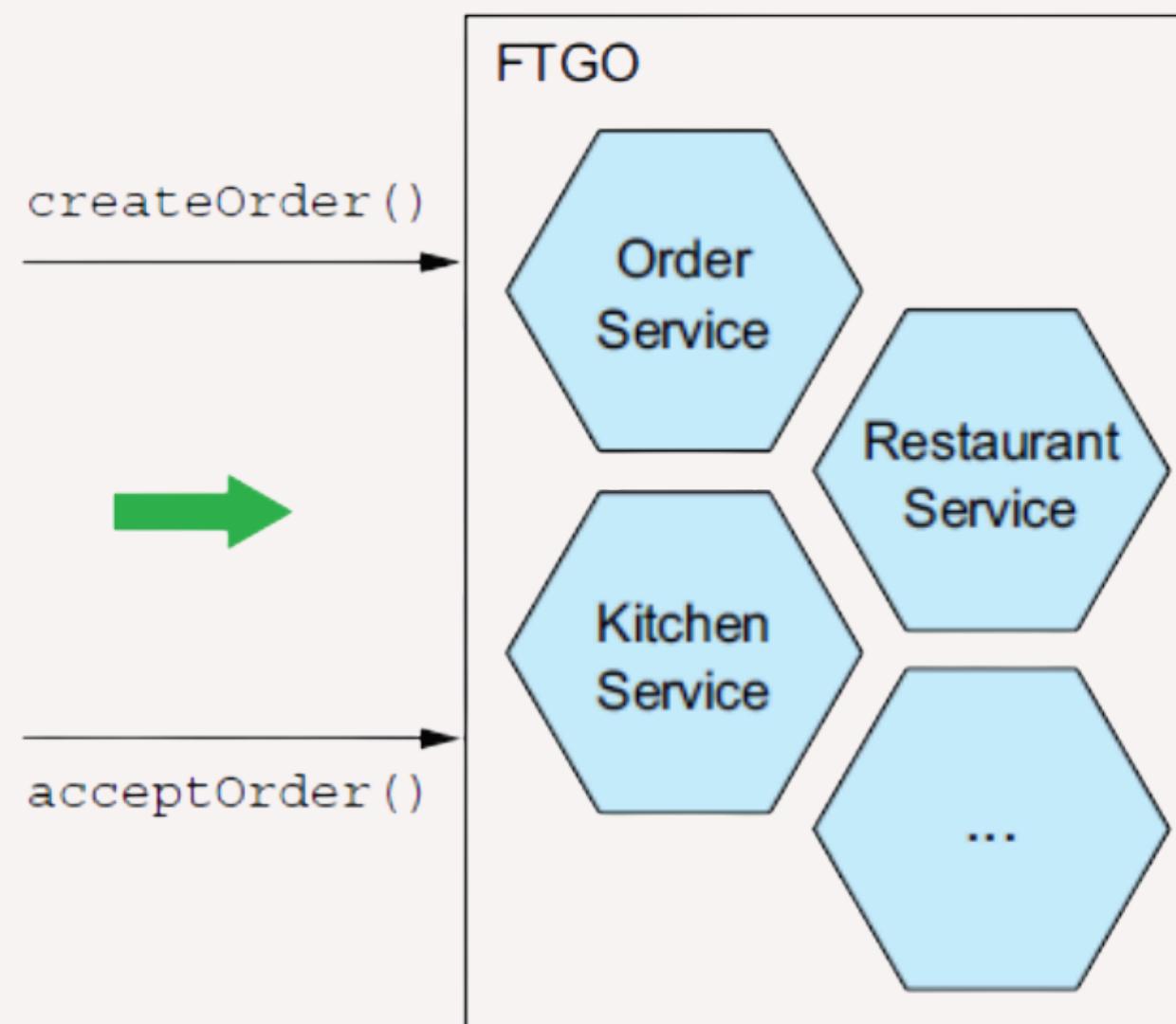
Xây dựng kiến trúc Decomposition



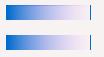
Bước 1: Xác định các thao tác của người dùng

Xây dựng kiến trúc Decomposition

Step 2: Identify services



Bước 2: Xác định các service cần thiết để xử lý thao tác của người dùng



Xây dựng kiến trúc Decomposition

Bước 2: Xác định các service cần thiết để xử lý thao tác của người dùng

Phân tách dựa trên nghiệp vụ: mỗi service chịu trách nhiệm end-to-end cho một nghiệp vụ cụ thể, ví dụ: Quản lý Đơn hàng, Thanh toán, Giao hàng, Báo cáo...

Phân tách dựa trên sub-domain: sử dụng tư tưởng của Domain-Driven Design, tách hệ thống theo các bounded context tương ứng với sub-domain trong domain model. Thường chia thành:

- + Core Domain: phần quan trọng nhất, tạo lợi thế cạnh tranh.
- + Supporting Domain: hỗ trợ core, không phải là trọng tâm.
- + Generic Domain: dịch vụ chung, có thể tái sử dụng (ví dụ: Authentication, Notification).



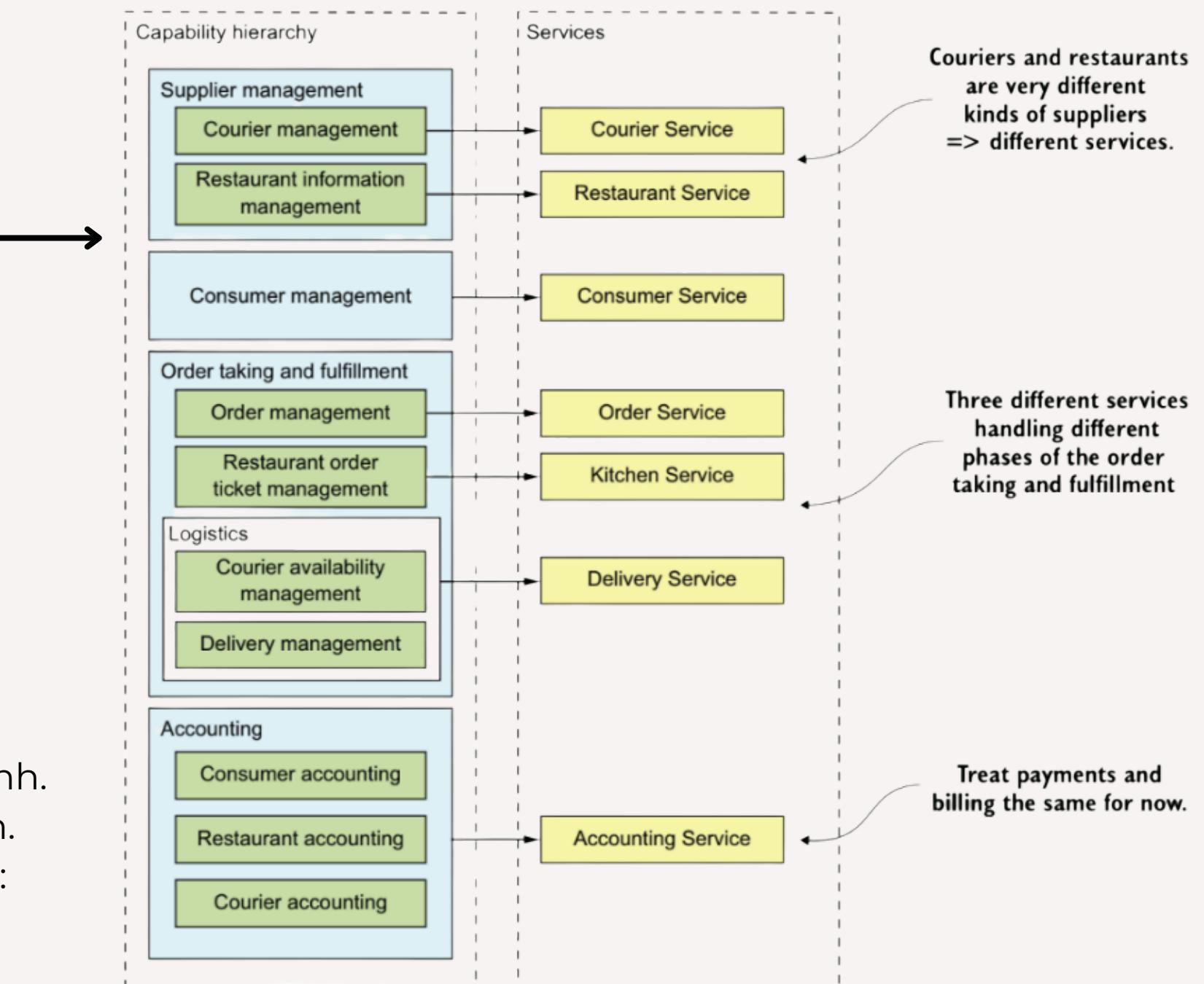
Xây dựng kiến trúc Decomposition

Phân tách dựa trên nghiệp vụ: mỗi service chịu trách nhiệm end-to-end cho một nghiệp vụ cụ thể, ví dụ: Quản lý Đơn hàng, Thanh toán, Giao hàng, Báo cáo...

Phân tách dựa trên sub-domain: sử dụng tư tưởng của Domain-Driven Design, tách hệ thống theo các bounded context tương ứng với sub-domain trong domain model.

Thường chia thành:

- + Core Domain: phần quan trọng nhất, tạo lợi thế cạnh tranh.
- + Supporting Domain: hỗ trợ core, không phải là trọng tâm.
- + Generic Domain: dịch vụ chung, có thể tái sử dụng (ví dụ: Authentication, Notification).



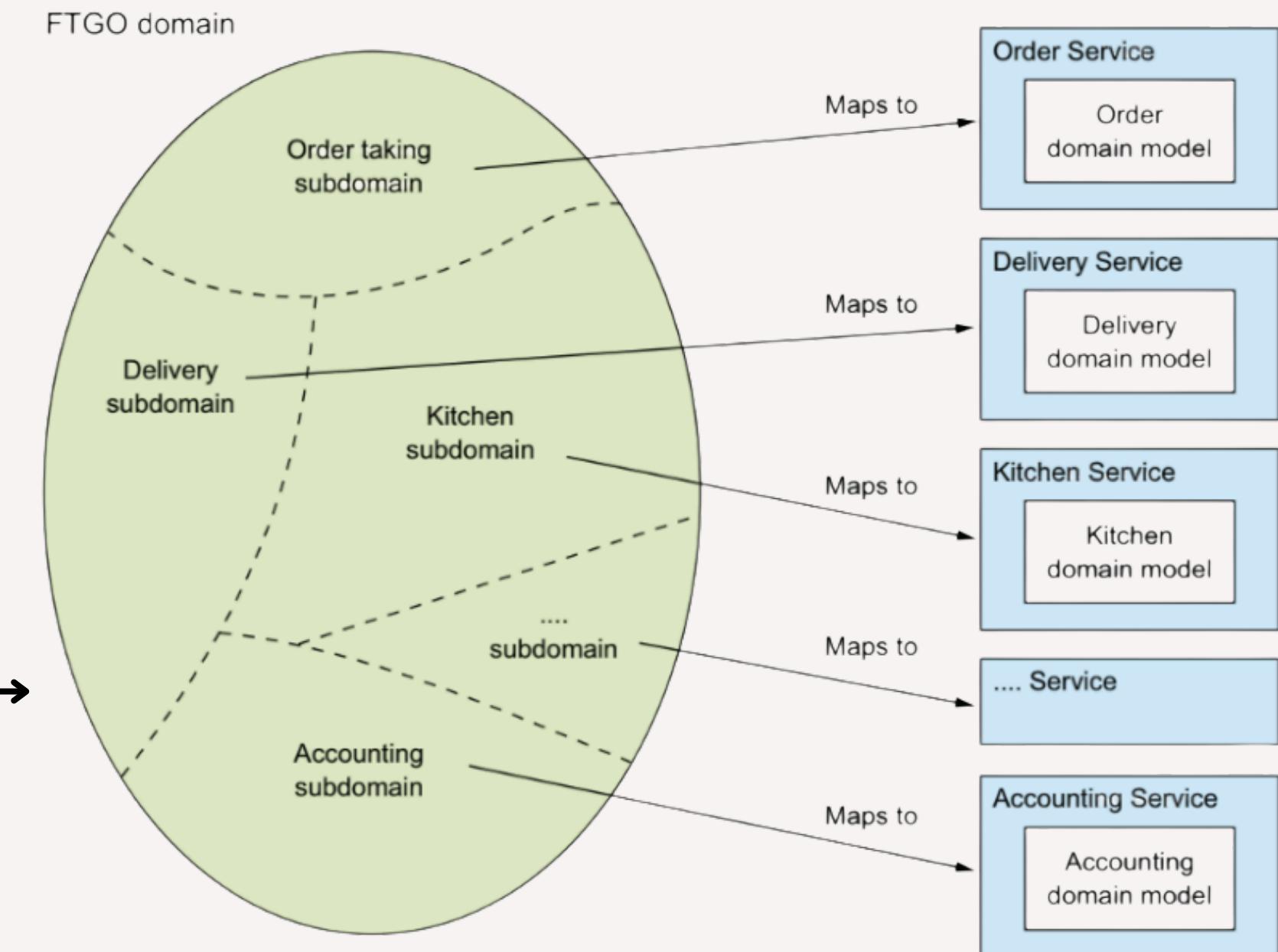


Xây dựng kiến trúc Decomposition

Phân tách dựa trên nghiệp vụ: mỗi service chịu trách nhiệm end-to-end cho một nghiệp vụ cụ thể, ví dụ: Quản lý Đơn hàng, Thanh toán, Giao hàng, Báo cáo...

Phân tách dựa trên sub-domain: sử dụng tư tưởng của Domain-Driven Design, tách hệ thống theo các bounded context tương ứng với sub-domain trong domain model. Thường chia thành:

- + Core Domain: phần quan trọng nhất, tạo lợi thế cạnh tranh.
- + Supporting Domain: hỗ trợ core, không phải là trọng tâm.
- + Generic Domain: dịch vụ chung, có thể tái sử dụng (ví dụ: Authentication, Notification).



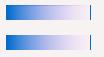


Xây dựng kiến trúc Decomposition

Những khó khăn trong quá trình phân tách

- Nghẽn mạng
- Giảm tính sẵn sàng của các service do giao tiếp đồng bộ
- Duy trì tính nhất quán của dữ liệu giữa các dịch vụ
- Các lớp "thần thánh" (God classes) cản trở việc chia nhỏ

- Nghẽn mạng là vấn đề phổ biến **khi các service trao đổi dữ liệu quá nhiều qua mạng.**
- Có thể giảm độ trễ bằng cách dùng **batch API** (gộp nhiều yêu cầu trong một lần gọi).
- **Trong một số trường hợp, giải pháp tốt hơn là:**
 1. **Hợp nhất các service lại,**
 2. **Giảm thiểu các cuộc gọi liên tiến trình (IPC),**
 3. **Thay thế bằng gọi nội bộ (RPC) để tăng hiệu suất.**



Xây dựng kiến trúc Decomposition

Những khó khăn trong quá trình phân tách

- Nghẽn mạng
- Giảm tính sẵn sàng của các service do giao tiếp đồng bộ
- Duy trì tính nhất quán của dữ liệu giữa các dịch vụ
- Các lớp "thần thánh" (God classes) cản trở việc chia nhỏ

- **IPC đồng bộ (như REST)** dễ gây giảm tính sẵn sàng của hệ thống.
- VD: Order Service gọi các service khác để tạo đơn → nếu 1 service lỗi → toàn bộ request thất bại.
=> Giải pháp tốt hơn: **sử dụng giao tiếp bất đồng bộ qua message queue** → tăng tính chống phụ thuộc và sẵn sàng.



Xây dựng kiến trúc Decomposition

Những khó khăn trong quá trình phân tách

- Nghẽn mạng
- Giảm tính sẵn sàng của các service do giao tiếp đồng bộ
- **Duy trì tính nhất quán của dữ liệu giữa các dịch vụ**
- Các lớp "thần thánh" (God classes) cản trở việc chia nhỏ

- Một số hành động **cần cập nhật dữ liệu trên nhiều service** (VD: khi đơn hàng được chấp nhận).
- **Giao dịch 2 pha (2PC) không phù hợp** cho các hệ thống hiện đại vì phức tạp và kém hiệu quả.

=> **Saga pattern là giải pháp thay thế**



Xây dựng kiến trúc Decomposition

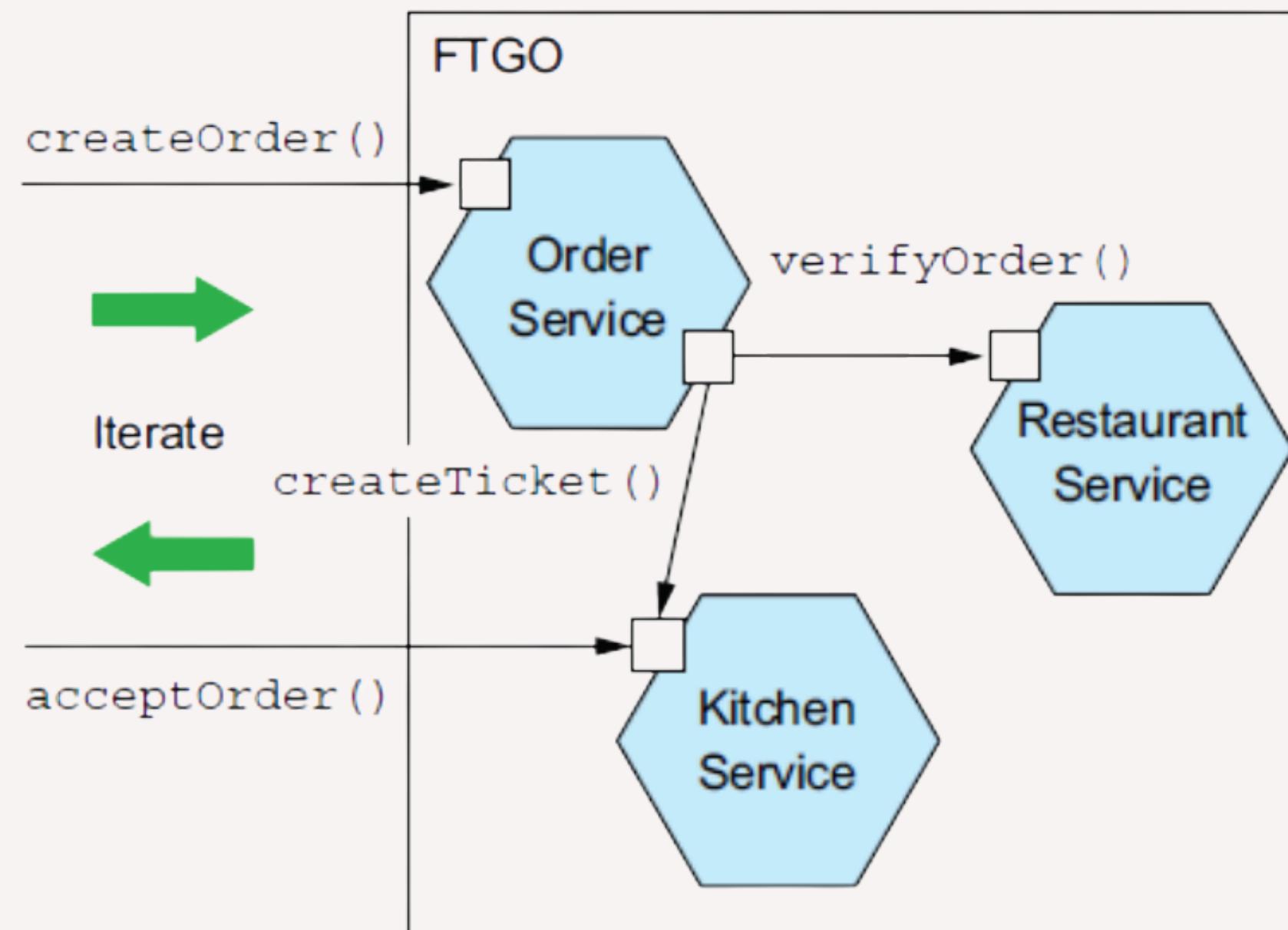
Những khó khăn trong quá trình phân tách

- Nghẽn mạng
- Giảm tính sẵn sàng của các service do giao tiếp đồng bộ
- Duy trì tính nhất quán của dữ liệu giữa các dịch vụ
- **Các lớp "thần thánh" (God classes) cản trở việc chia nhỏ**

- Là **các lớp quá lớn, chứa** nhiều logic nghiệp vụ phức tạp và trạng thái đa dạng.
 - Thường gặp ở các **entity trung tâm** như Order, Account, Policy,...
 - VD: lớp Order trong hệ thống giao đồ ăn có thể gộp cả: Xử lý đơn hàng, Quản lý đơn hàng nhà hàng, Giao hàng, Thanh toán
- **Những lớp này gây khó khăn khi muốn chia tách thành các service riêng vì phụ thuộc chằng chịt và có trạng thái phức tạp.**

Xây dựng kiến trúc Decomposition

Step 3: Define service APIs and collaborations



Bước 3: Định nghĩa các API và thiết lập tương tác giữa các service



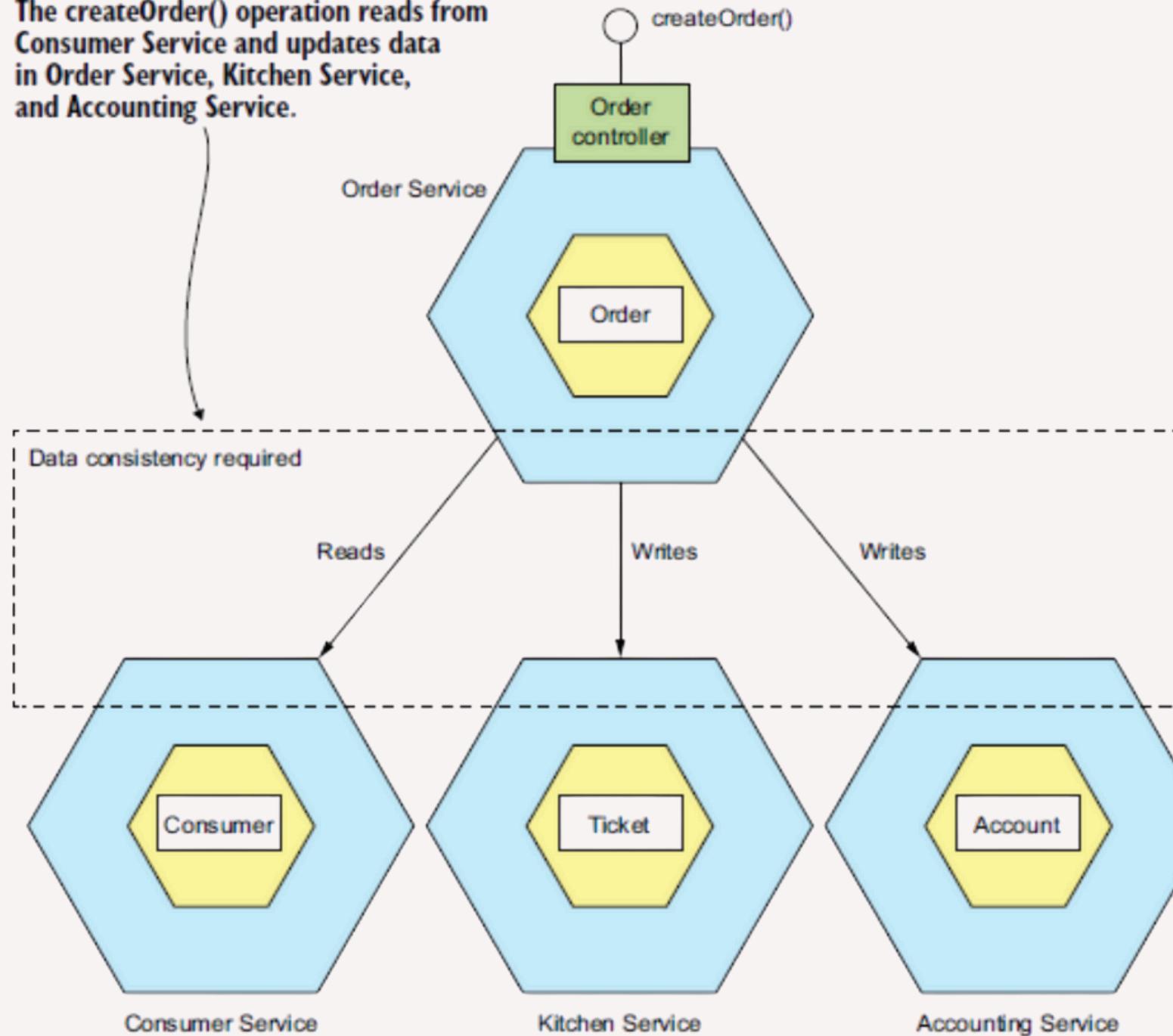
04.

Data Consistency Pattern (Saga Pattern)



Đặt vấn đề

The `createOrder()` operation reads from Consumer Service and updates data in Order Service, Kitchen Service, and Accounting Service.



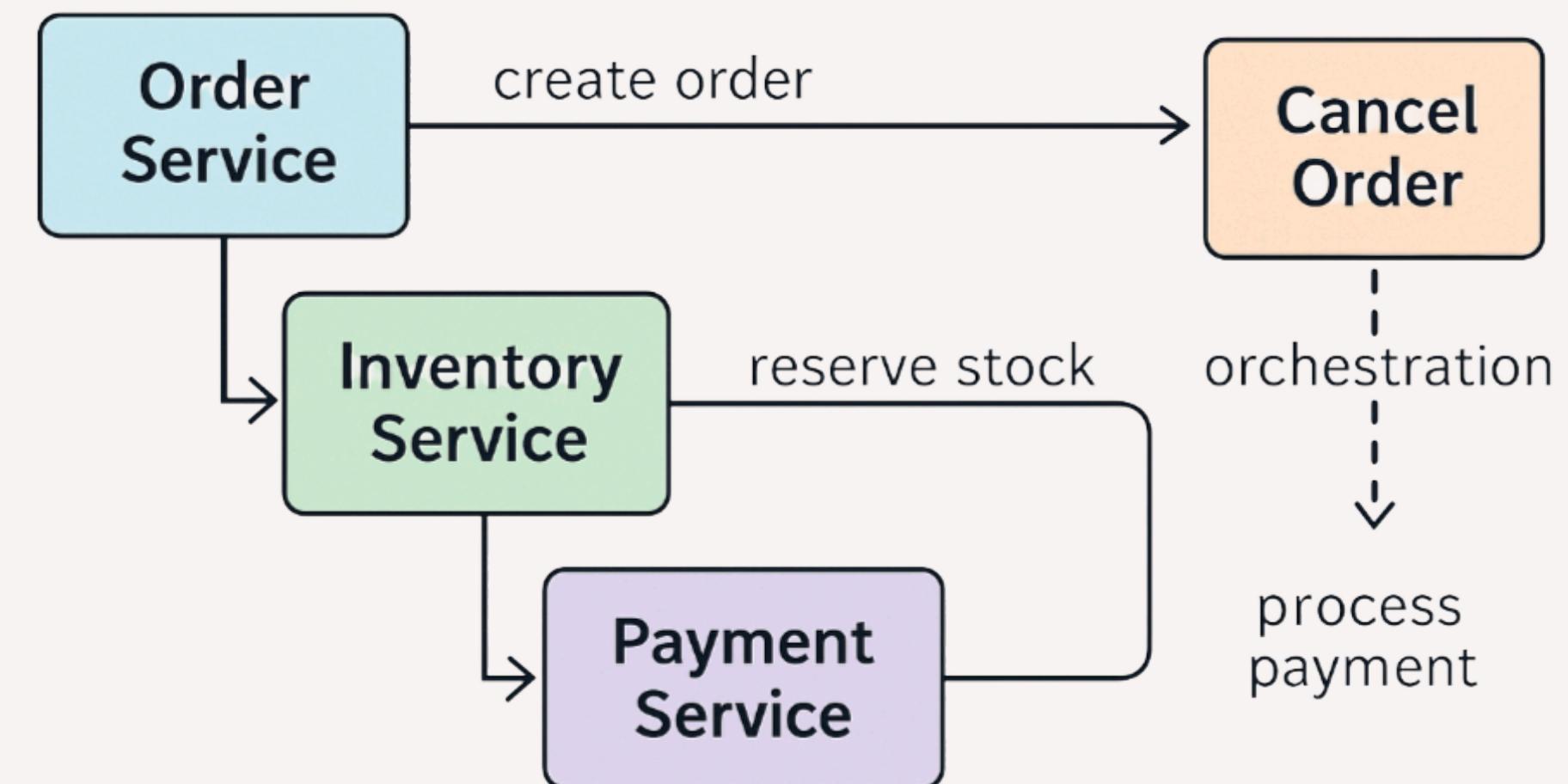
Trong kiến trúc microservice, khi transaction cần cập nhật dữ liệu của nhiều dịch vụ, ACID không còn phù hợp..

Saga Pattern (Data Consistency Pattern)

Định nghĩa

- Sagas là một cơ chế thay thế giao dịch phân tán, **giúp duy trì tính nhất quán dữ liệu trong hệ thống microservice.**
- Một saga được định nghĩa cho **mỗi lệnh hệ thống cần cập nhật dữ liệu ở nhiều dịch vụ.**
- Saga bao gồm **một chuỗi các giao dịch cục bộ**, mỗi giao dịch:
 1. Chỉ **cập nhật dữ liệu trong một dịch vụ duy nhất**
 2. Sử dụng **giao dịch ACID** nội bộ quen thuộc (có hỗ trợ commit/rollback)
 3. Các giao dịch này được liên kết với nhau thông qua **message (thông điệp bất đồng bộ)**.

Saga Pattern



Giao dịch bồi thường

Giao dịch bồi thường là **cơ chế "hoàn tác"** các thay đổi của các giao dịch cục bộ trong saga **nếu có lỗi xảy ra**. Khi một bước trong saga thất bại, các giao dịch bồi thường sẽ được **thực hiện theo thứ tự ngược lại để khôi phục trạng thái ban đầu**.



Step 1:
Create
order

Step 2:
Withdraw
payment

Step 3:
Confirm
shipment

Giao dịch bồi thường



**Step 1:
Create
order**



**Step 2:
Withdraw
payment**



**Step 3:
Confirm
shipment**

**Nếu bước 3 thất bại
(hết hàng,...)**

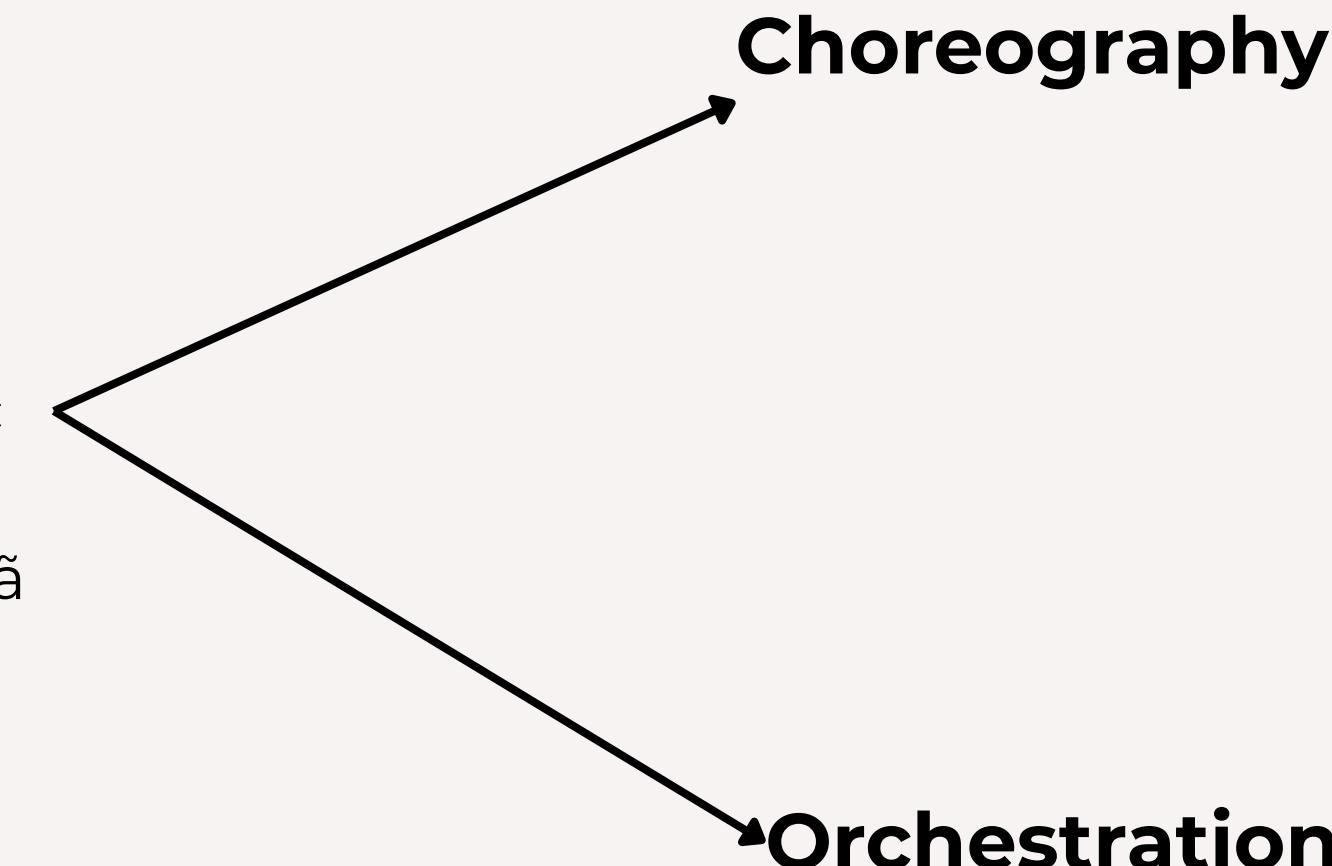
**Giao dịch bồi thường
cho bước 1: Hủy đơn**

**Giao dịch bồi thường
cho bước 2: Hoàn trả
tiền**



Cài đặt Saga

Triển khai saga bao gồm **logic điều phối thực hiện tuần tự các giao dịch cục bộ**. Khi được khởi tạo bởi một lệnh hệ thống, saga sẽ thực hiện bước đầu tiên, sau đó tiếp tục lần lượt các bước kế tiếp. Nếu có bước nào thất bại, hệ thống sẽ thực hiện các giao dịch bồi thường theo thứ tự ngược lại để hoàn tác các bước đã thực hiện trước đó.



Choreography

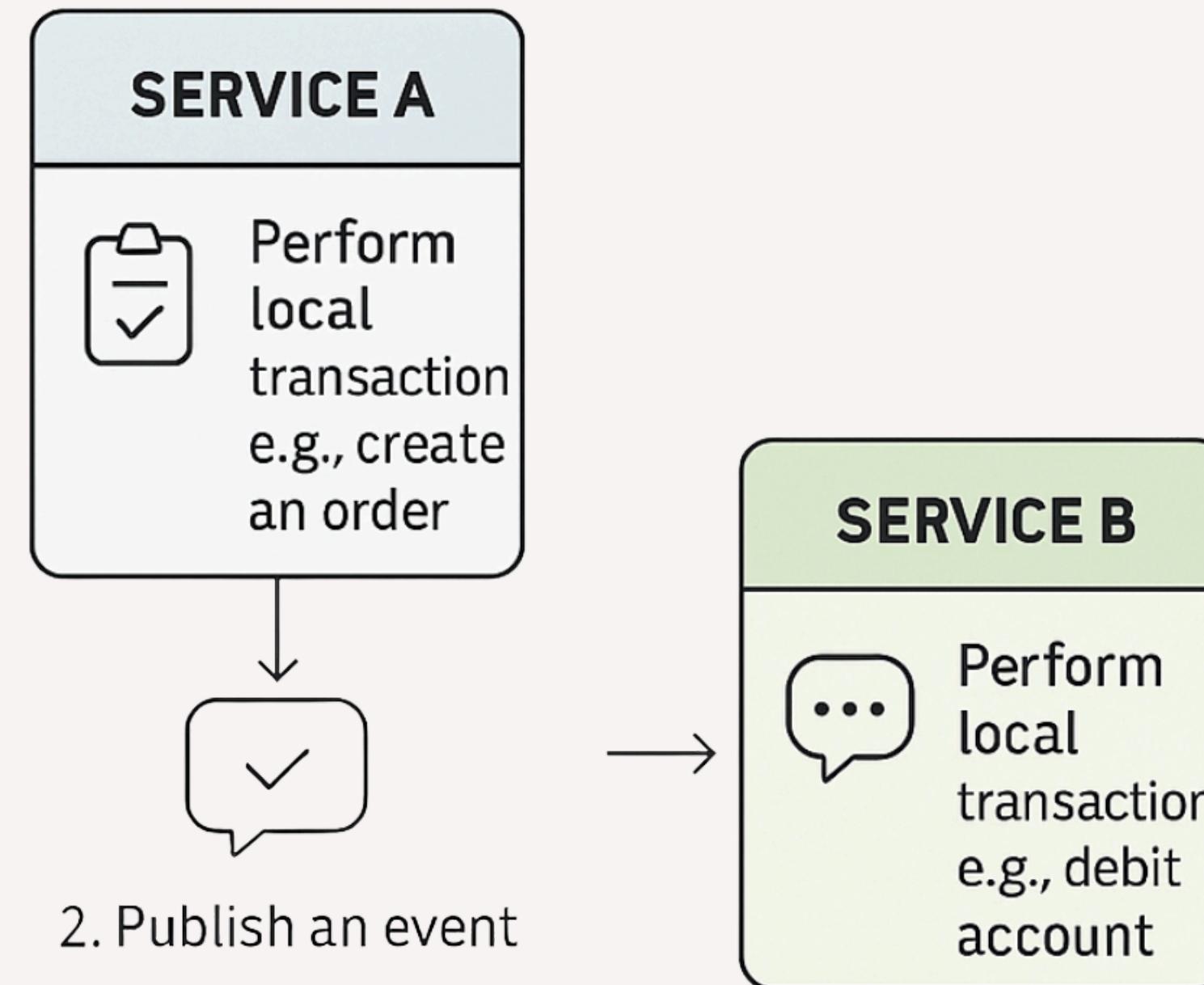
Orchestration

Cài đặt Saga

Choreography

Orchestration

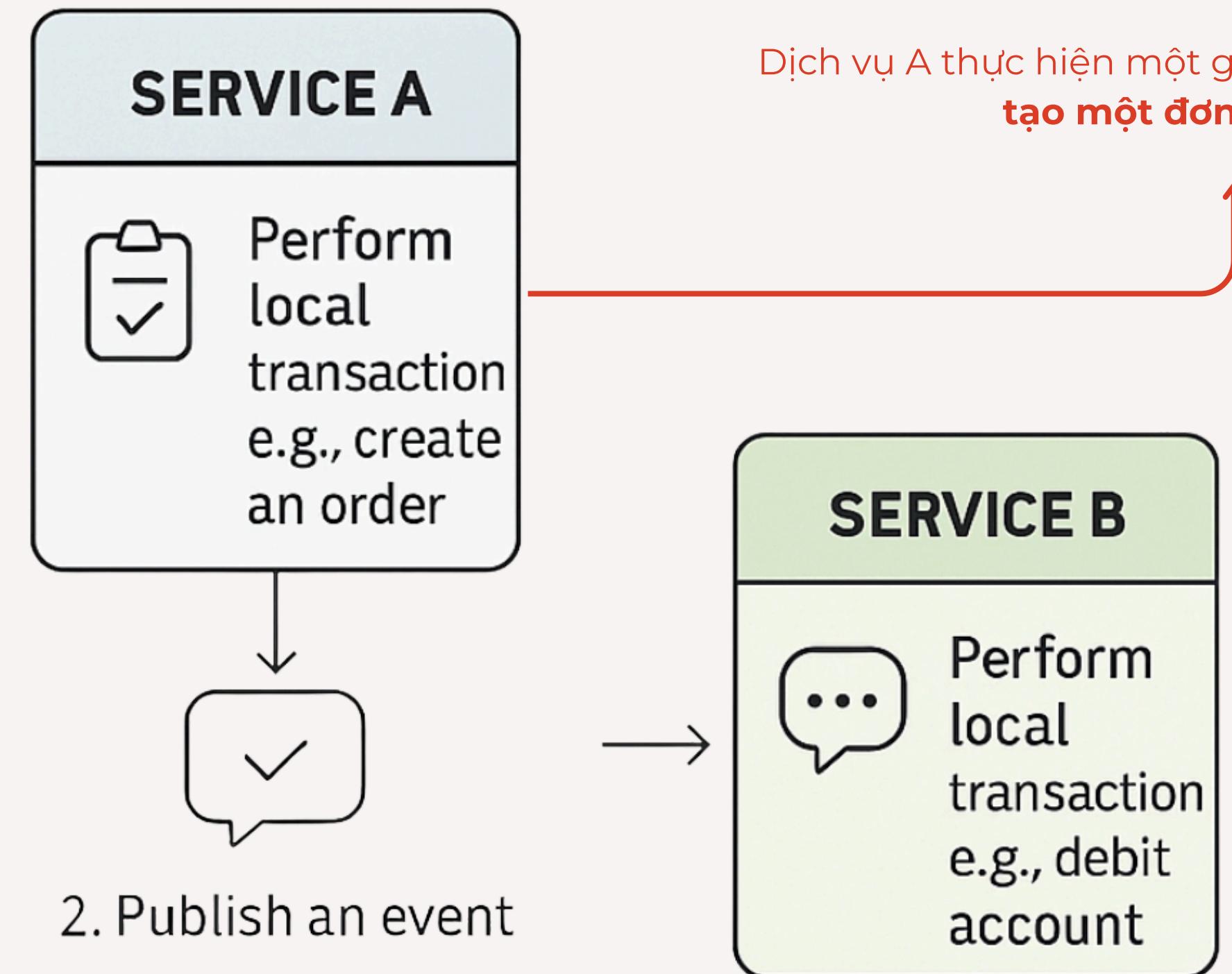
Choreography-based sagas là **phương pháp điều phối saga phi tập trung**, trong đó **các dịch vụ tự quyết định và thực hiện bước tiếp theo dựa trên các sự kiện nhận được**. Mỗi dịch vụ tham gia sẽ trao đổi sự kiện với nhau để phối hợp và tiến hành toàn bộ quy trình mà **không cần một điểm điều phối trung tâm**.



Cài đặt Saga

Choreography

Orchestration

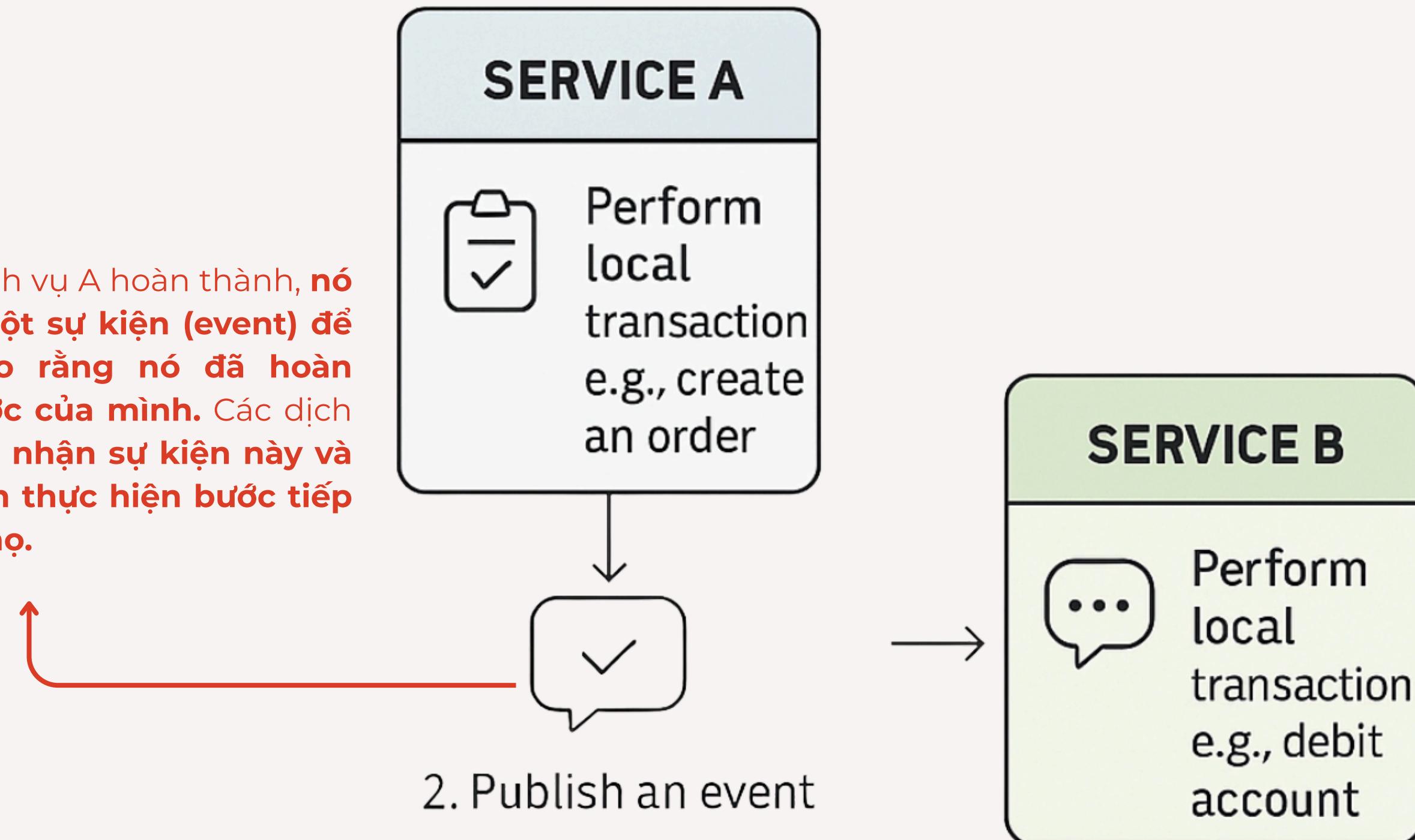


Cài đặt Saga

Choreography

Sau khi dịch vụ A hoàn thành, **nó sẽ phát một sự kiện (event) để thông báo rằng nó đã hoàn thành bước của mình.** Các dịch vụ khác sẽ nhận sự kiện này và quyết định thực hiện bước tiếp theo của họ.

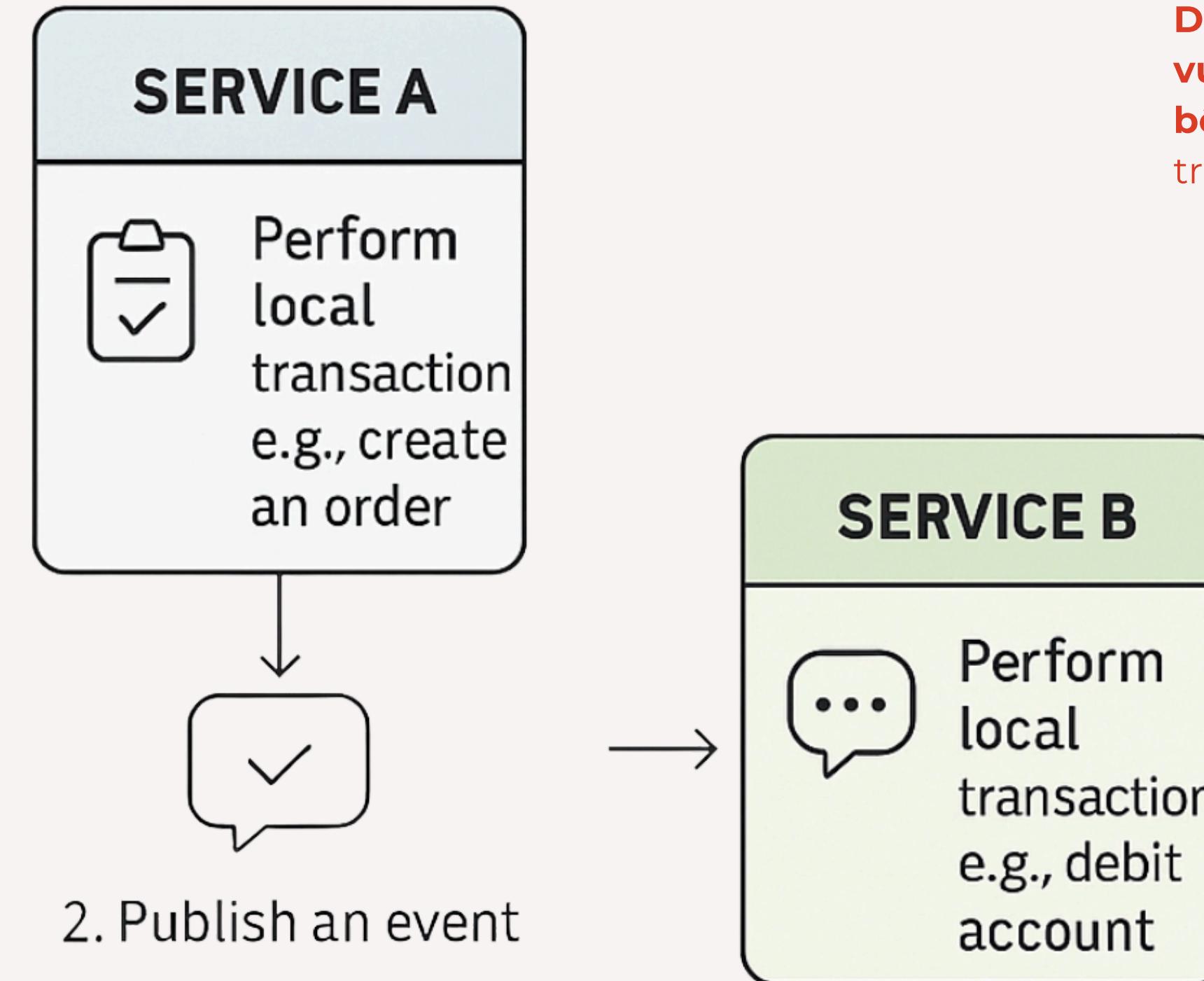
Orchestration



Cài đặt Saga

Choreography

Orchestration



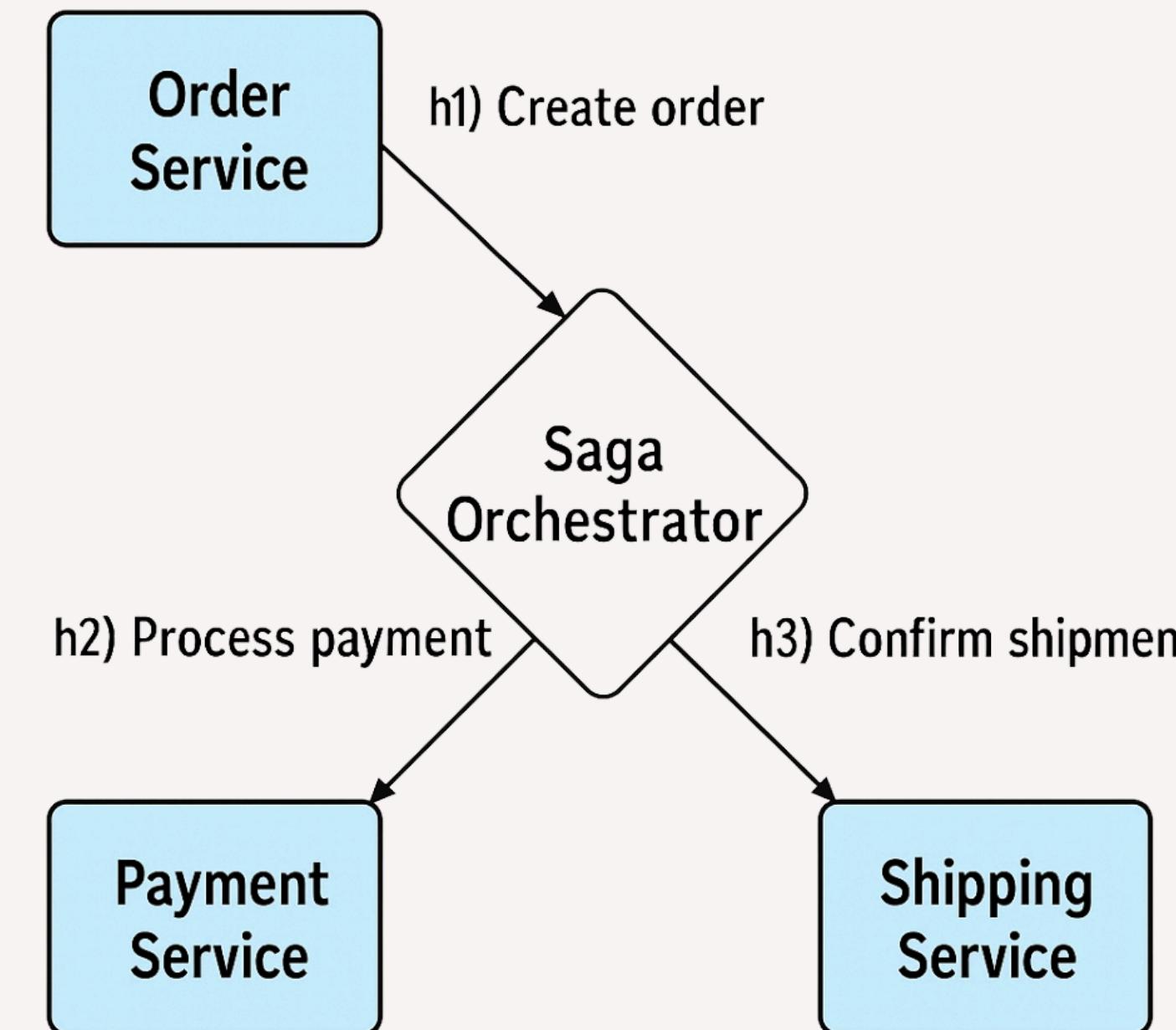
Dịch vụ B nhận sự kiện từ dịch vụ A và thực hiện giao dịch cục bộ của mình, ví dụ như trừ tiền trong tài khoản của khách hàng.

Cài đặt Saga

Choreography

Orchestration

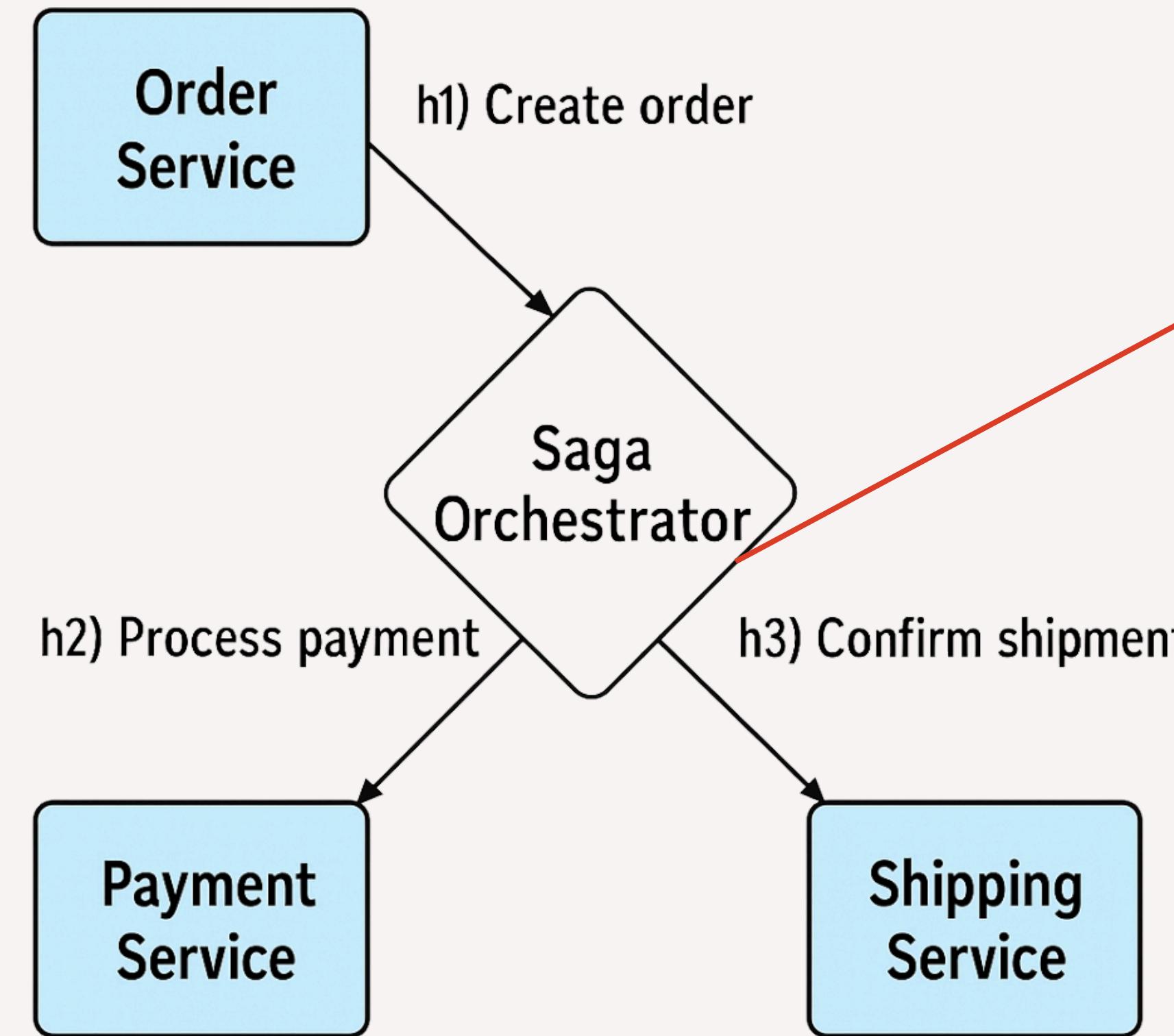
Orchestration-based sagas là **phương pháp điều phối saga theo kiểu tập trung**, trong đó một **thành phần trung tâm (saga orchestrator)** chịu trách nhiệm quản lý và điều phối toàn bộ quy trình. Orchestrator sẽ gửi yêu cầu đến các dịch vụ để thực hiện từng bước trong saga, thay vì để các dịch vụ tự quyết định như trong choreography-based sagas.



Cài đặt Saga

Choreography

Orchestration

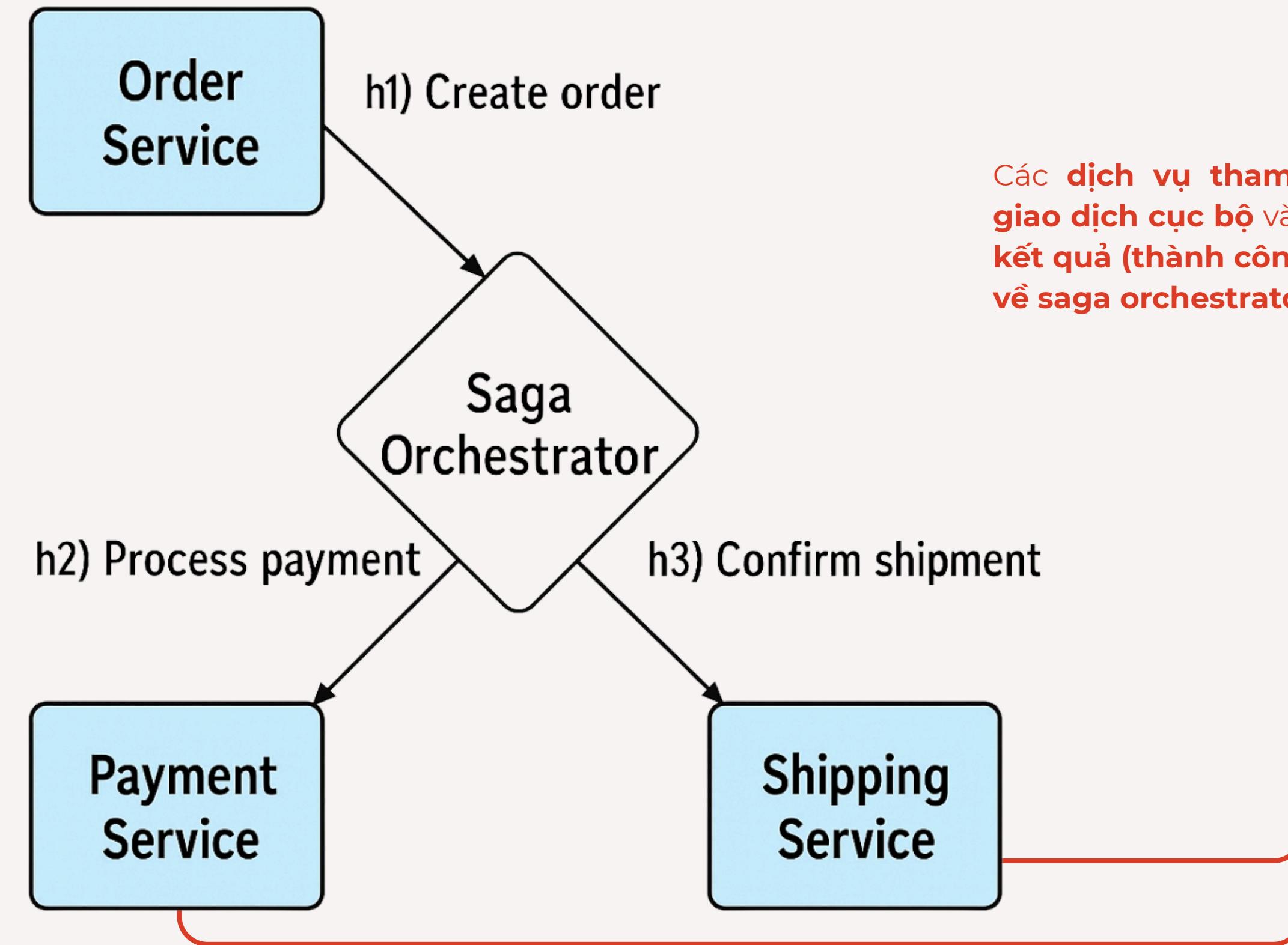


Dịch vụ trung tâm điều phối:
Thường gọi là **saga orchestrator**, sẽ xác định thứ tự các bước trong saga.
Dịch vụ này sẽ gửi các lệnh (commands) tới các dịch vụ tham gia trong saga, yêu cầu họ thực hiện các giao dịch cục bộ.

Cài đặt Saga

Choreography

Orchestration



Các dịch vụ tham gia thực hiện giao dịch cục bộ và sau đó báo cáo kết quả (thành công hoặc thất bại) về saga orchestrator.



05.

Deployment Pattern



Giới thiệu

Khái niệm

Deployment là quá trình **đưa sản phẩm phần mềm từ môi trường phát triển đến tay người dùng.**

Liên quan

Deployment gắn liền với quy trình **phát triển phần mềm và kiến trúc phần mềm.**

Vai trò tham gia

- **Developer (Lập trình):** Phát triển sản phẩm.
- **Operation (Vận hành/IT Admin):** Nhận sản phẩm và thực hiện triển khai.

Quy trình cũ

Developer hoàn thành và gửi phiên bản cho Operation để deploy.

Xu hướng mới

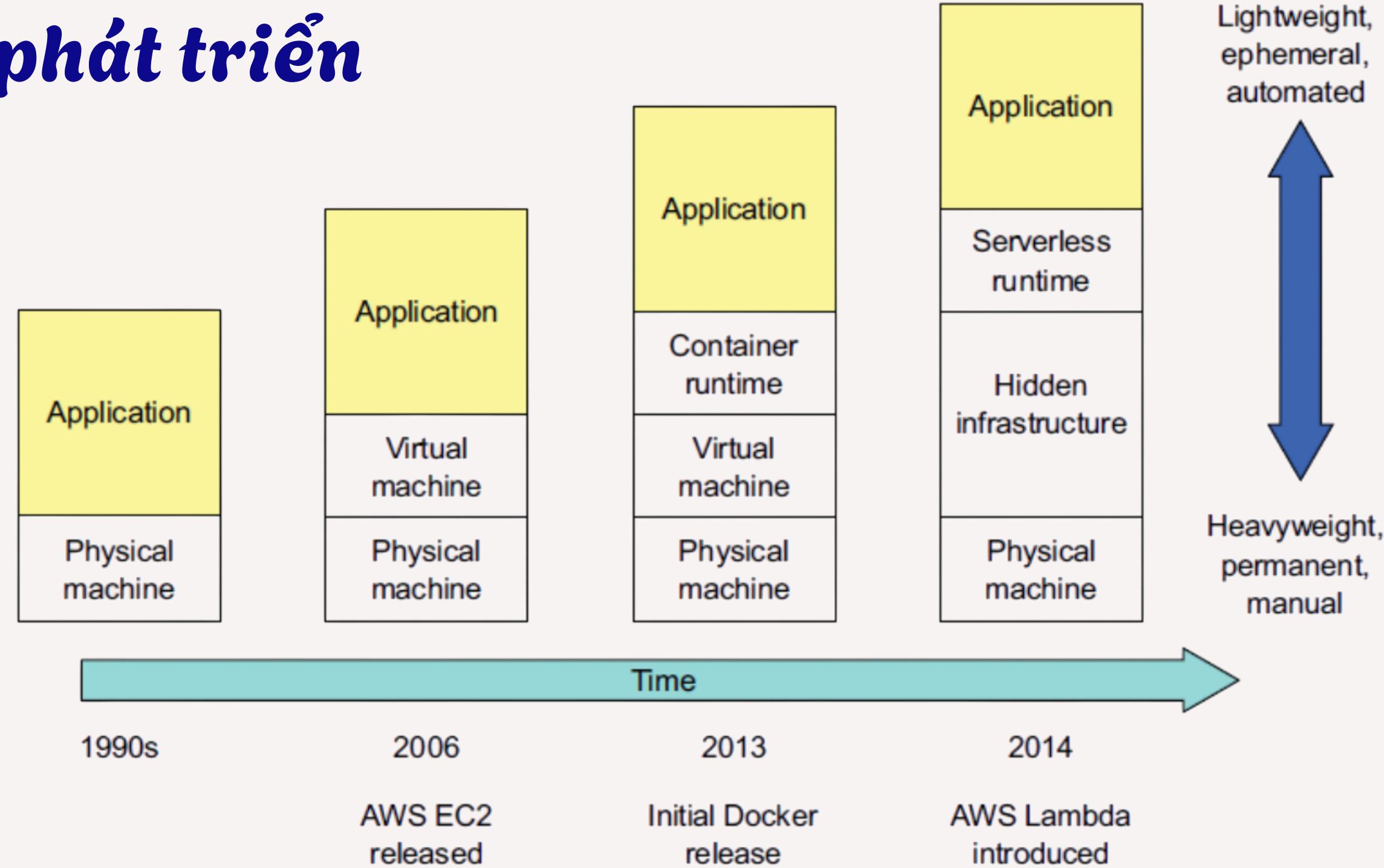
Với sự ra đời của Agile/Scrum và Microservices, quá trình deployment giờ đây có sự tham gia chủ động từ cả Developer và Operation, giúp:

- Tăng tốc độ triển khai.
- Dễ dàng triển khai sản phẩm hơn.

Phương pháp thực hiện

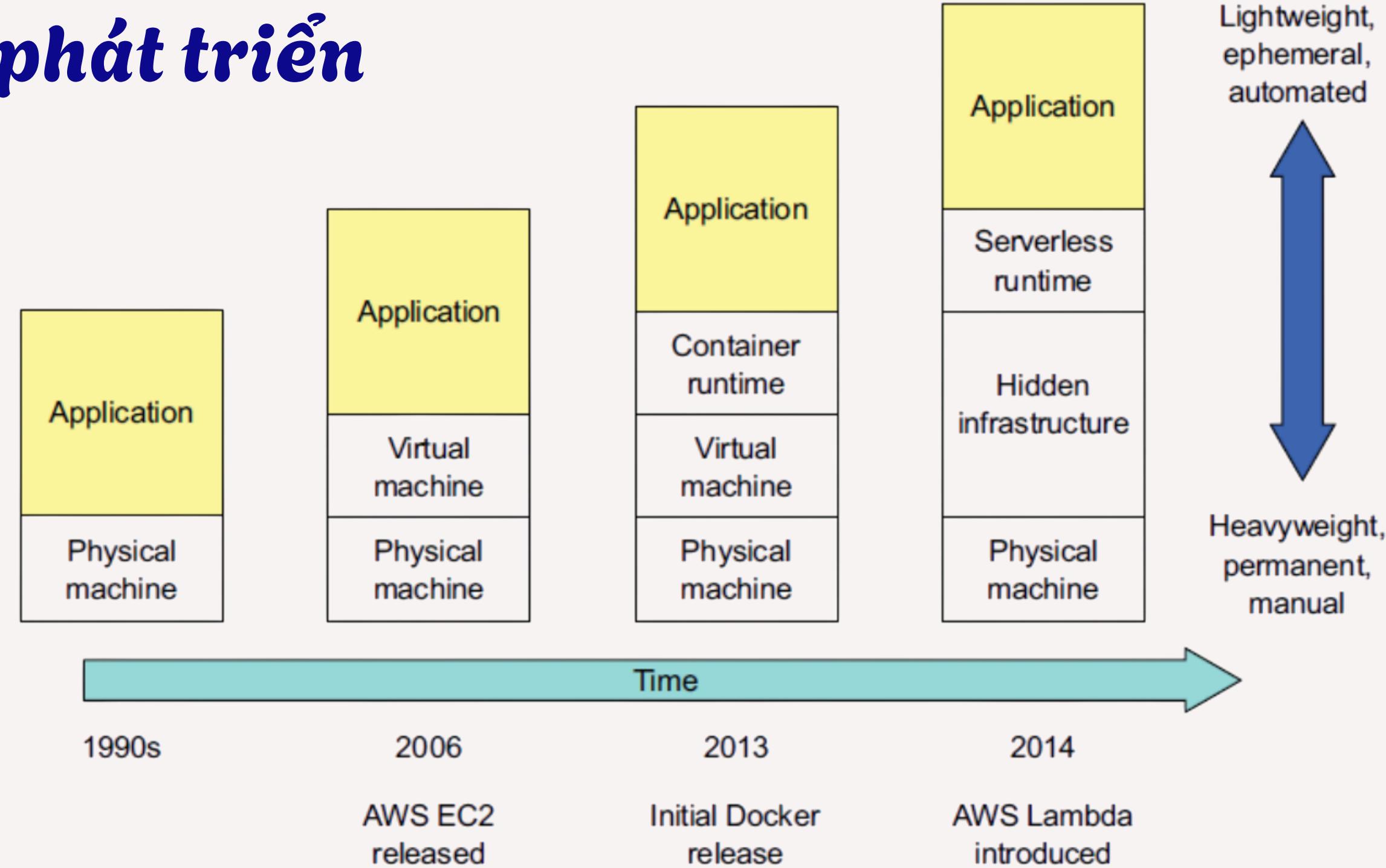
- Tạo một môi trường deploy
- Triển khai ứng dụng trên môi trường deploy đó

Lịch sử phát triển



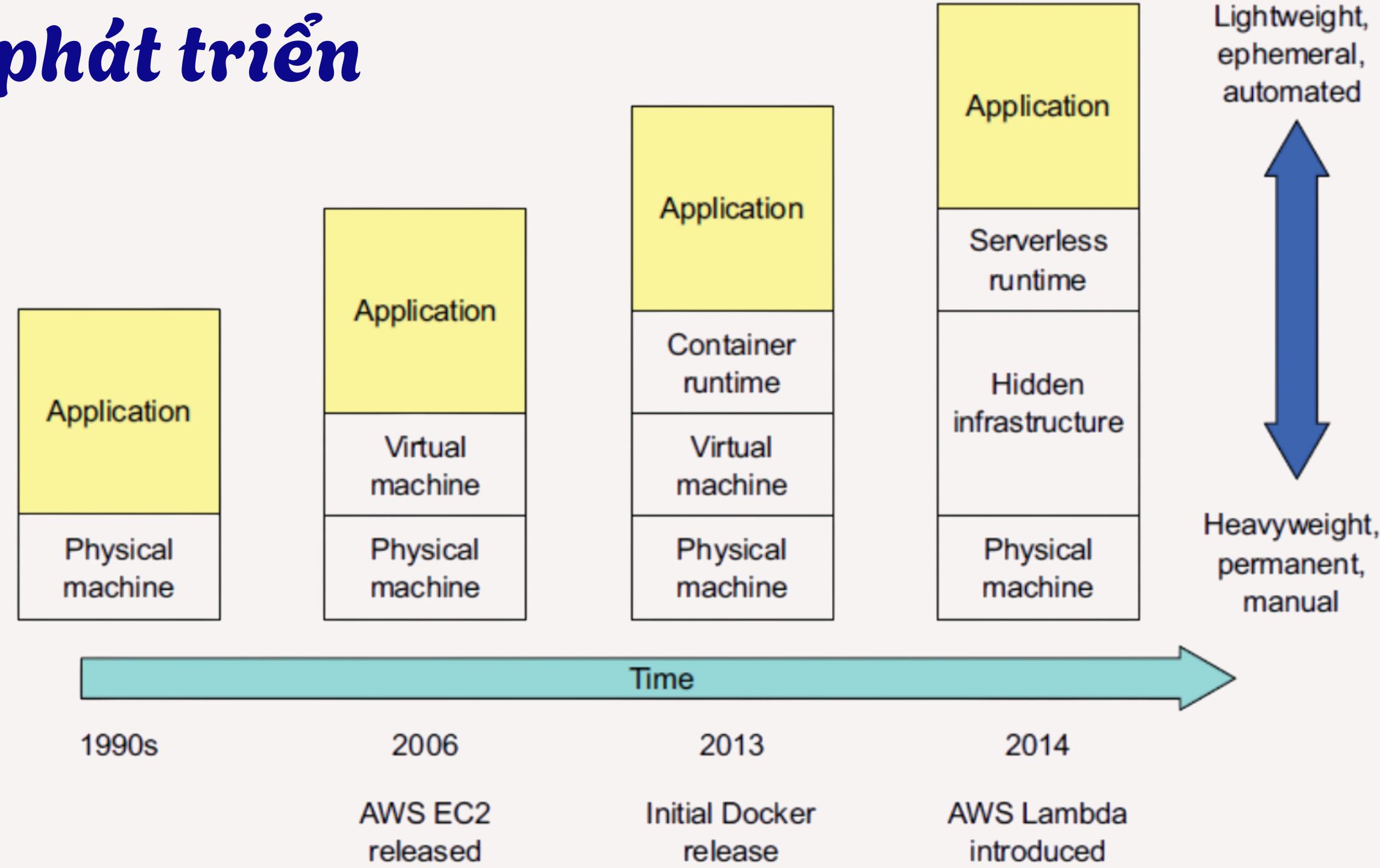
1990s: Developer phải hẹn với Operation để gửi mã nguồn và hướng dẫn deploy thủ công → chậm, dễ lỗi.

Lịch sử phát triển



2000s: Máy ảo ra đời →
deploy nhiều app trên
cùng một máy, nhưng
vẫn chưa tự động hóa.

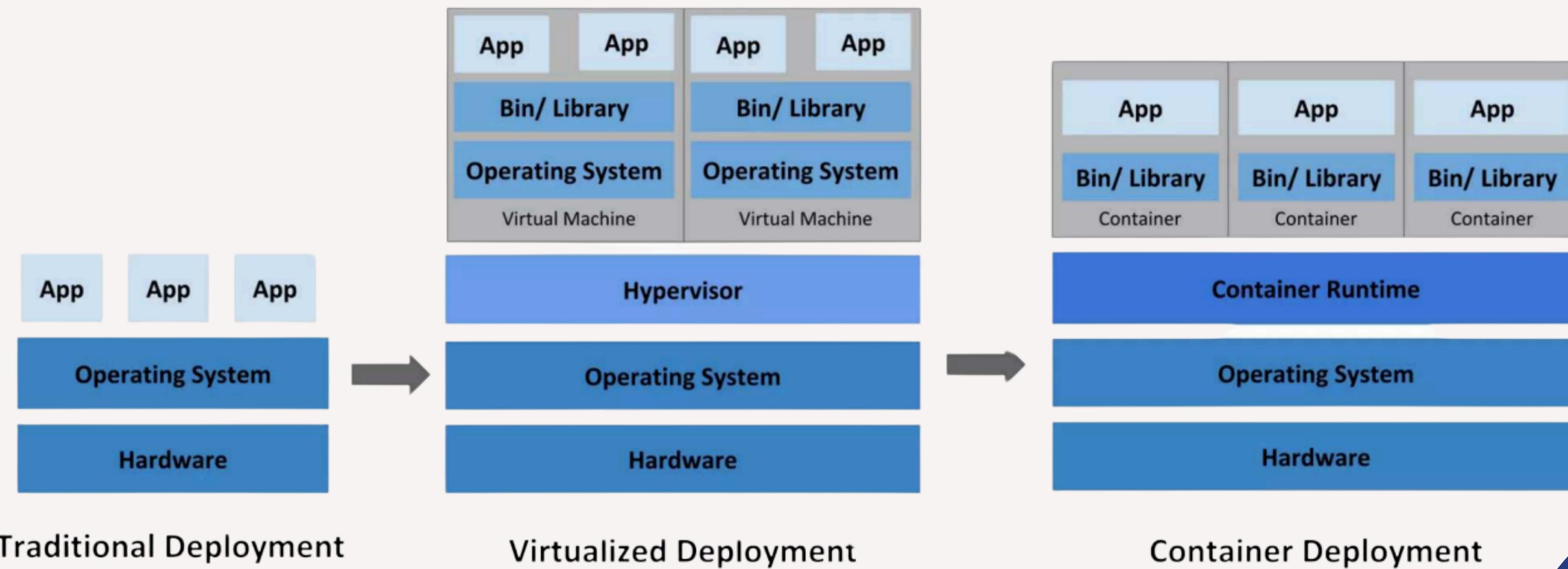
Lịch sử phát triển



2010s: Cloud & Docker bùng nổ → DevOps ra đời, giúp quy trình triển khai tự động, nhanh chóng và hiệu quả.

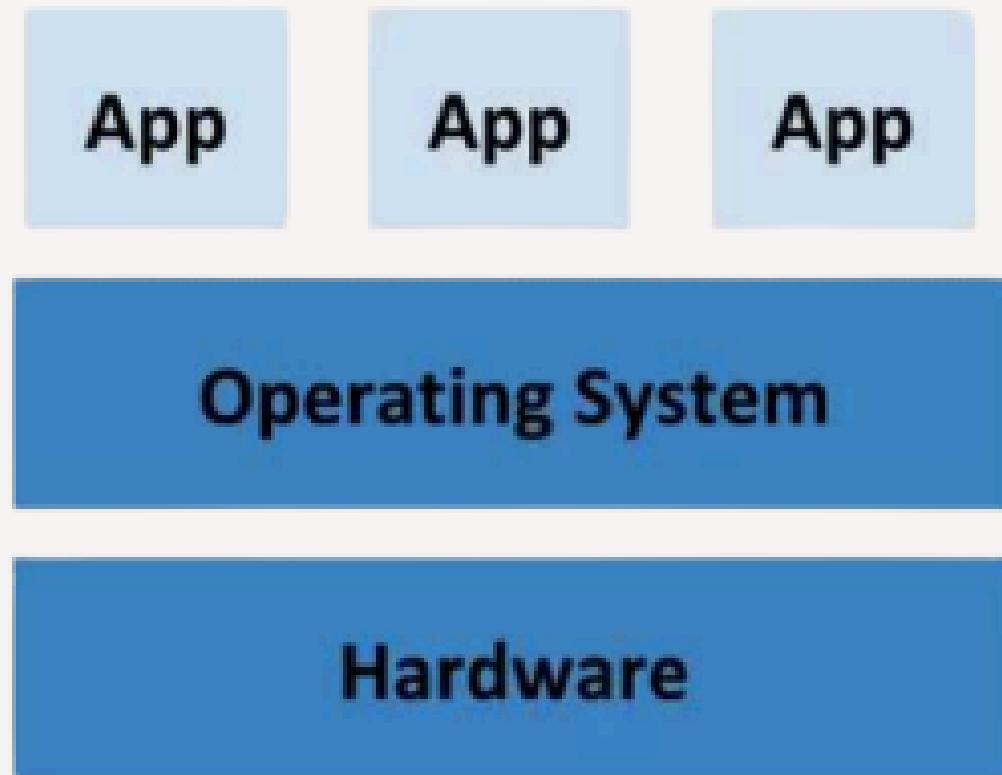


Hình thức deploy





Hình thức deploy



Traditional Deployment

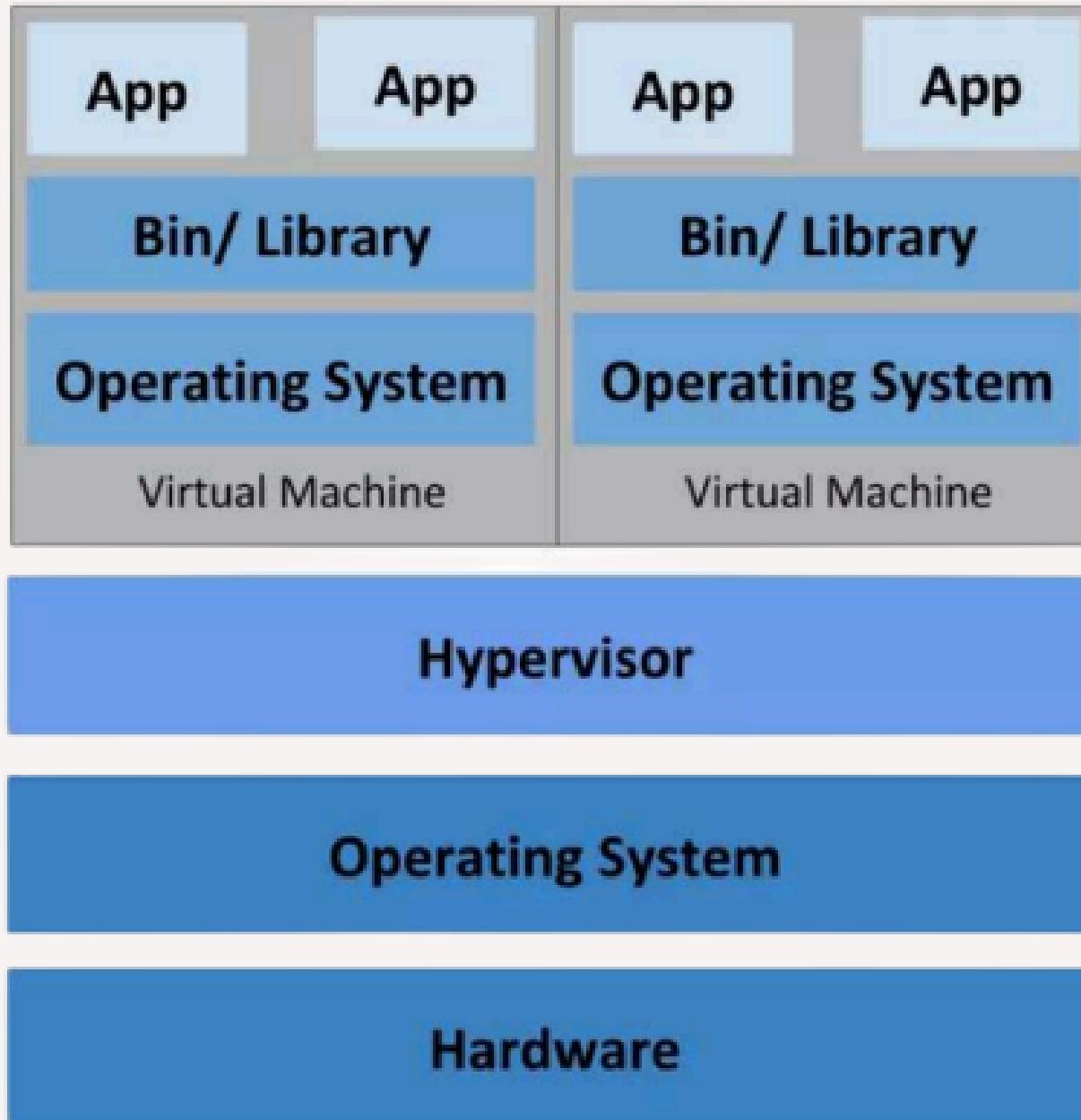
Traditional deployment:

Nhiều ứng dụng **chia sẻ chung tài nguyên** trên 1 máy → **dễ xung đột, kém hiệu quả.**





Hình thức deploy



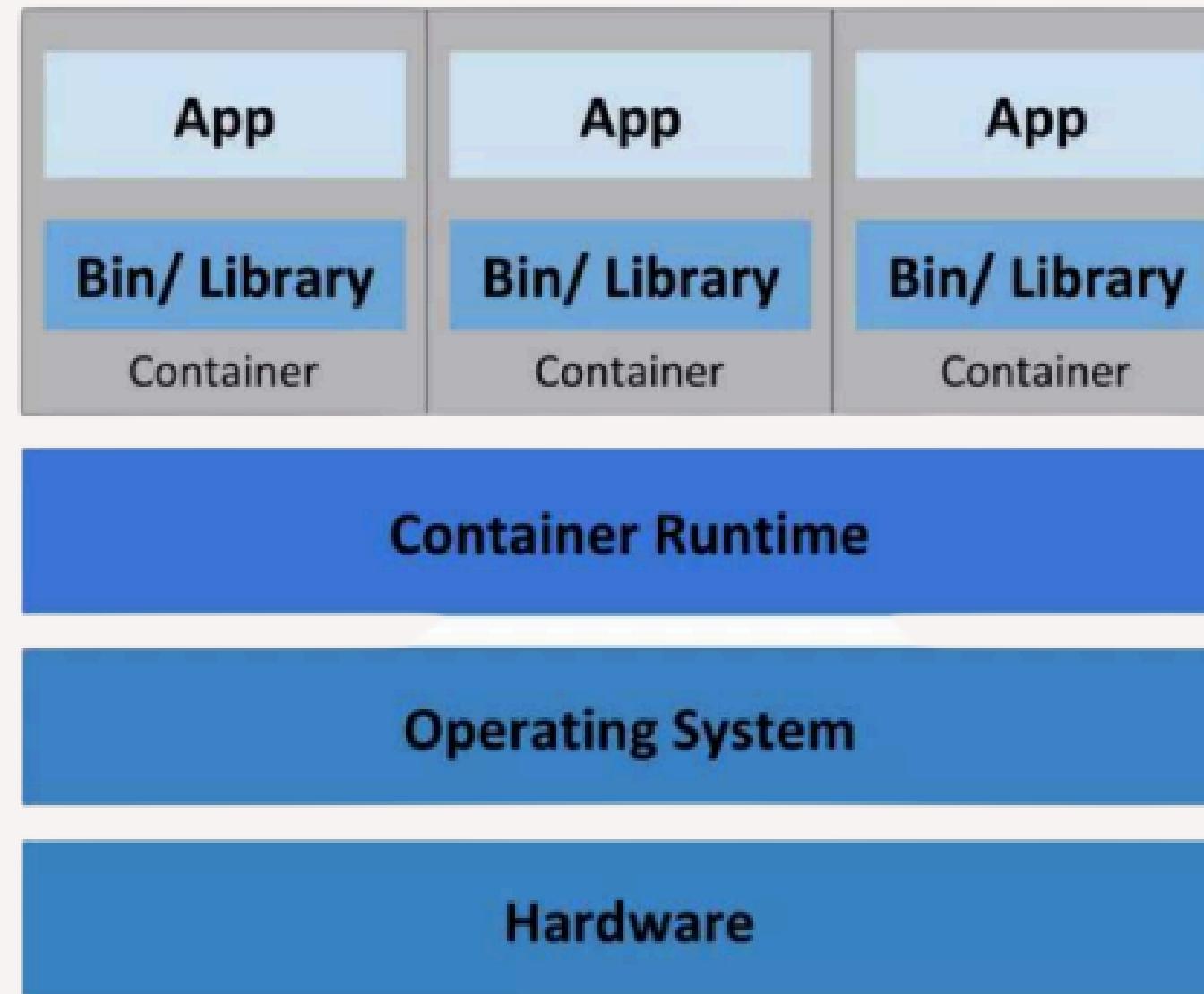
Virtualized Deployment

Virtualized deployment

Dùng máy ảo để **cách ly tài nguyên tốt hơn**, nhưng vẫn cần **cài đặt nhiều dependencies**.



Hình thức deploy

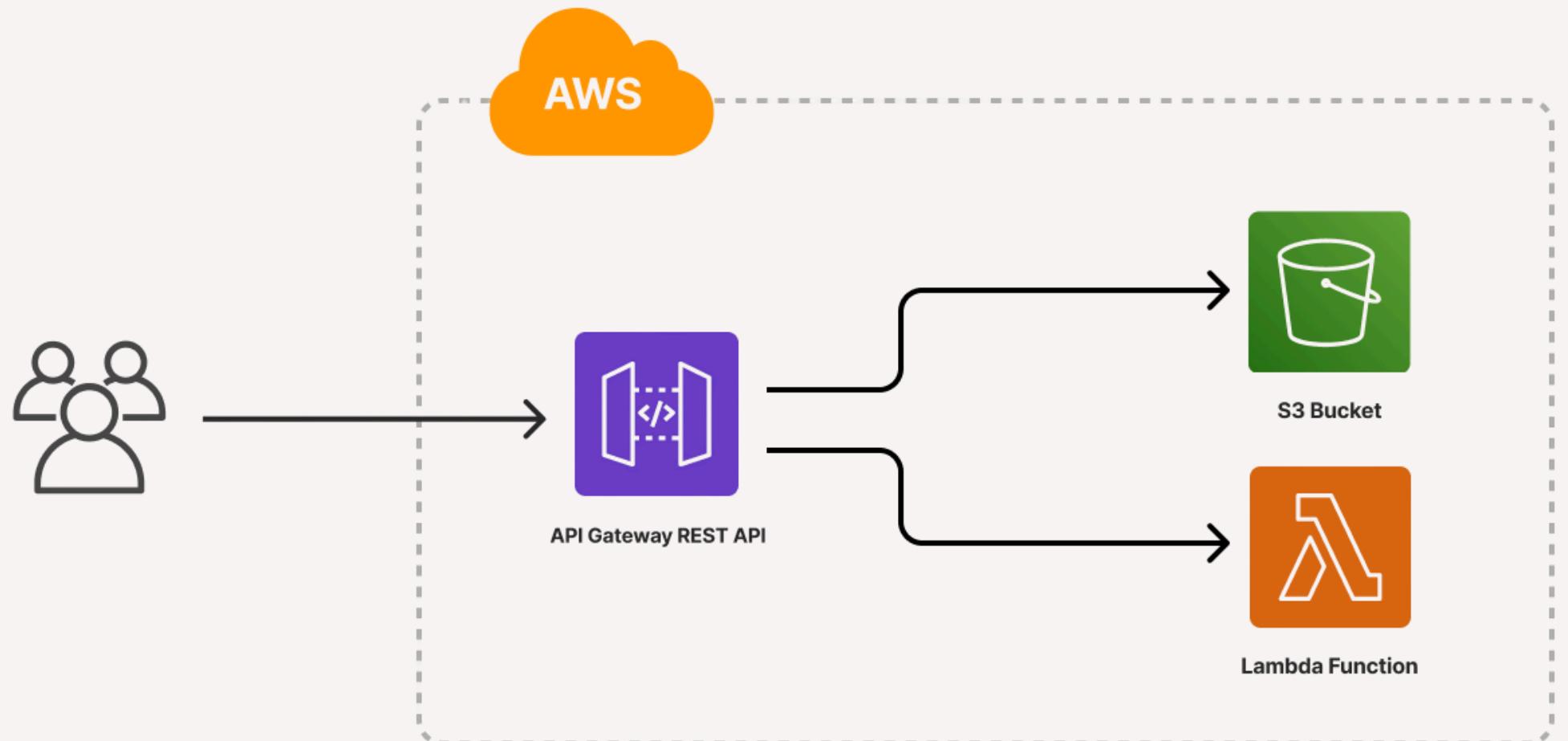


Container deployment

Dùng Docker để **đóng gói app thành image** → không cần cài thêm gì ngoài Docker, dễ quản lý & triển khai.



Hình thức deploy



Serverless deployment

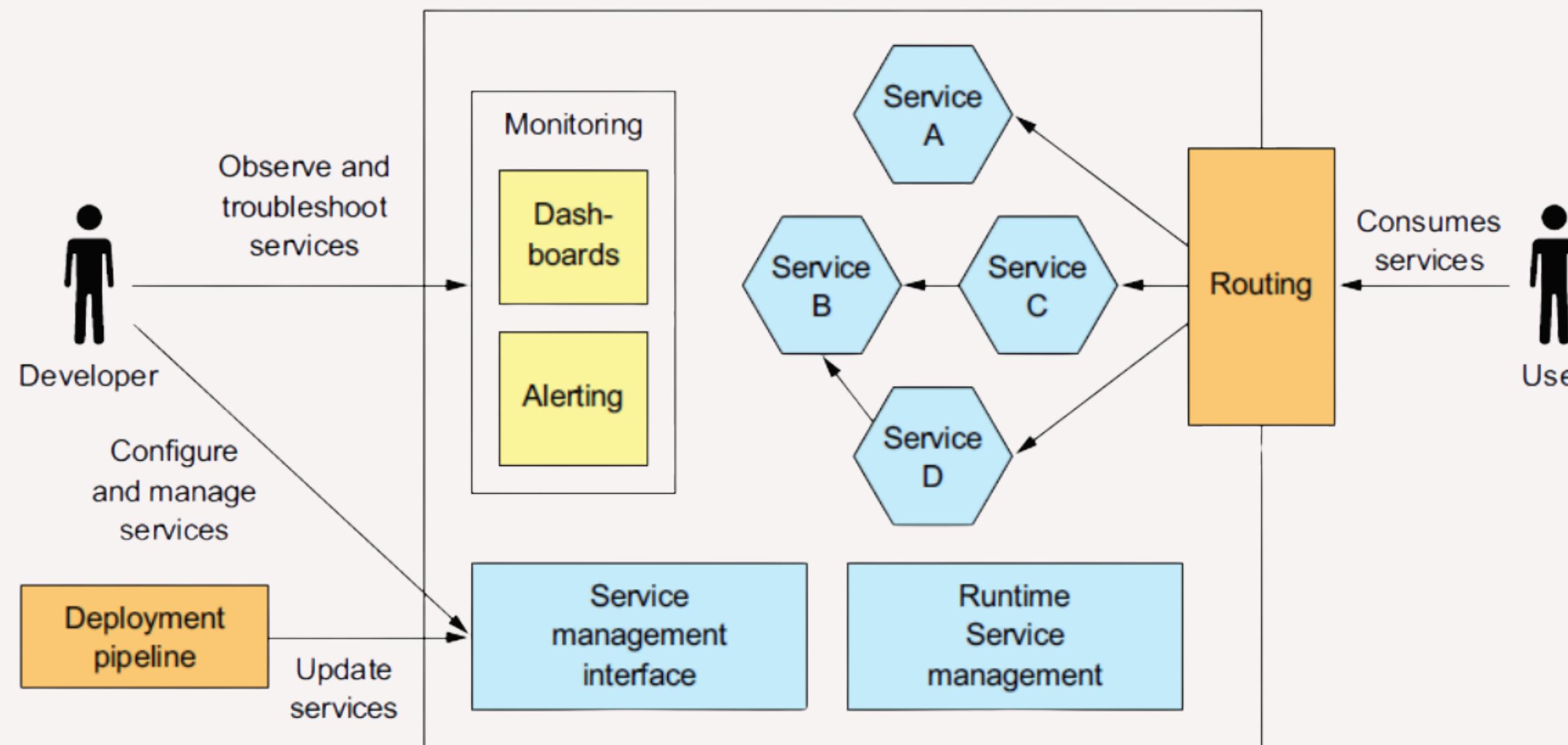
Với sự phát triển của Cloud, AWS đi đầu trong công nghệ serverless khi giúp cho các dự án khi triển khai không cần phải quan tâm đến giá tiền ngay từ đầu mà cho phép '**dùng đến đâu, trả đến đó**' và lo luôn cả về vấn đề scaling



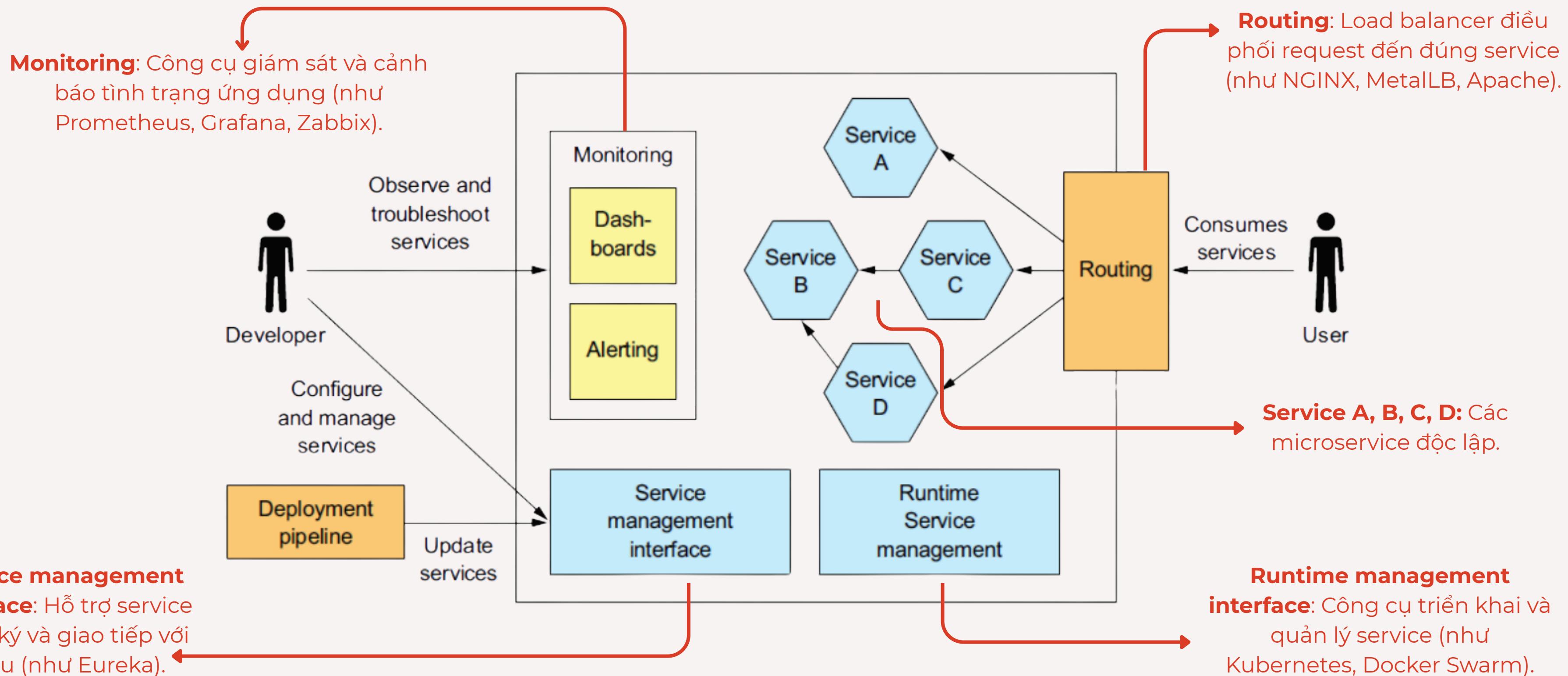


Tự động hóa và hạ tầng

Với kiến trúc microservice, một ứng dụng có thể gồm hàng trăm service khác nhau. **Việc deploy và cấu hình thủ công cho từng service là không khả thi** vì giới hạn nhân lực. Do đó, **tự động hóa deployment và hạ tầng là bắt buộc** để đảm bảo hiệu quả và khả năng mở rộng trong môi trường production.



Tự động hóa và hạ tầng





Phân loại

1

Bằng package ngôn ngữ

Dùng file như JAR/WAR (Java), DLL/EXE (.NET); chỉ cần máy có runtime là có thể chạy ngay.

2

Bằng máy ảo

Mỗi ứng dụng chạy trong máy ảo riêng để tránh xung đột tài nguyên và lỗi tiềm tàng.

3

Bằng container

Giống máy ảo nhẹ, chỉ chứa phần cần thiết để chạy ứng dụng, tiết kiệm tài nguyên và dễ triển khai.

4

Bằng serverless.

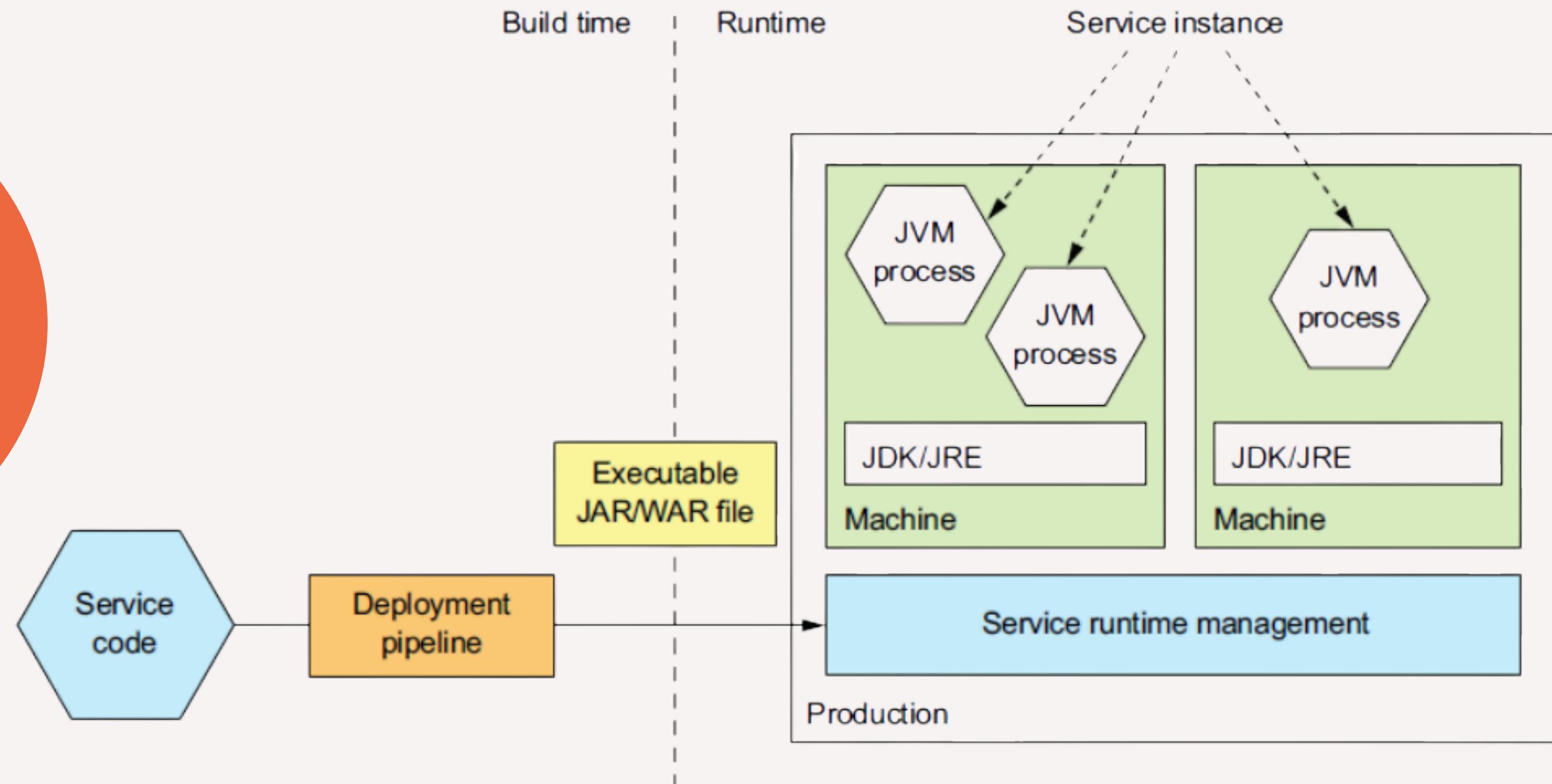
Giao toàn bộ việc deploy và vận hành cho cloud provider, tính phí theo mức sử dụng

Phân loại

1

Bằng package ngôn ngữ

Dùng file như JAR/WAR (Java), DLL/EXE (.NET); chỉ cần máy có runtime là có thể chạy ngay.

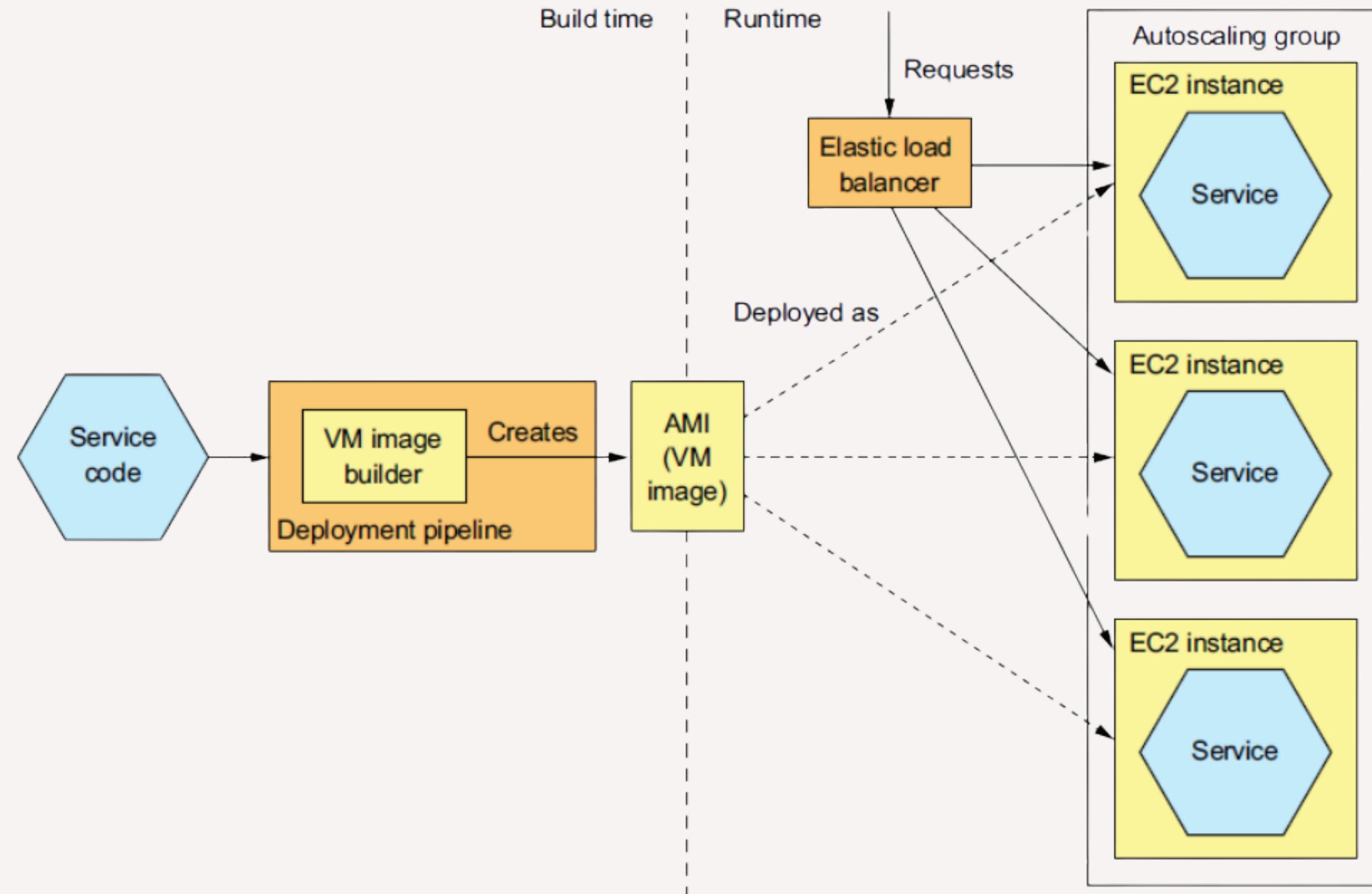


Phân loại

2

Bằng máy ảo

Mỗi ứng dụng chạy trong máy ảo riêng để tránh xung đột tài nguyên và lỗi tiềm tàng.



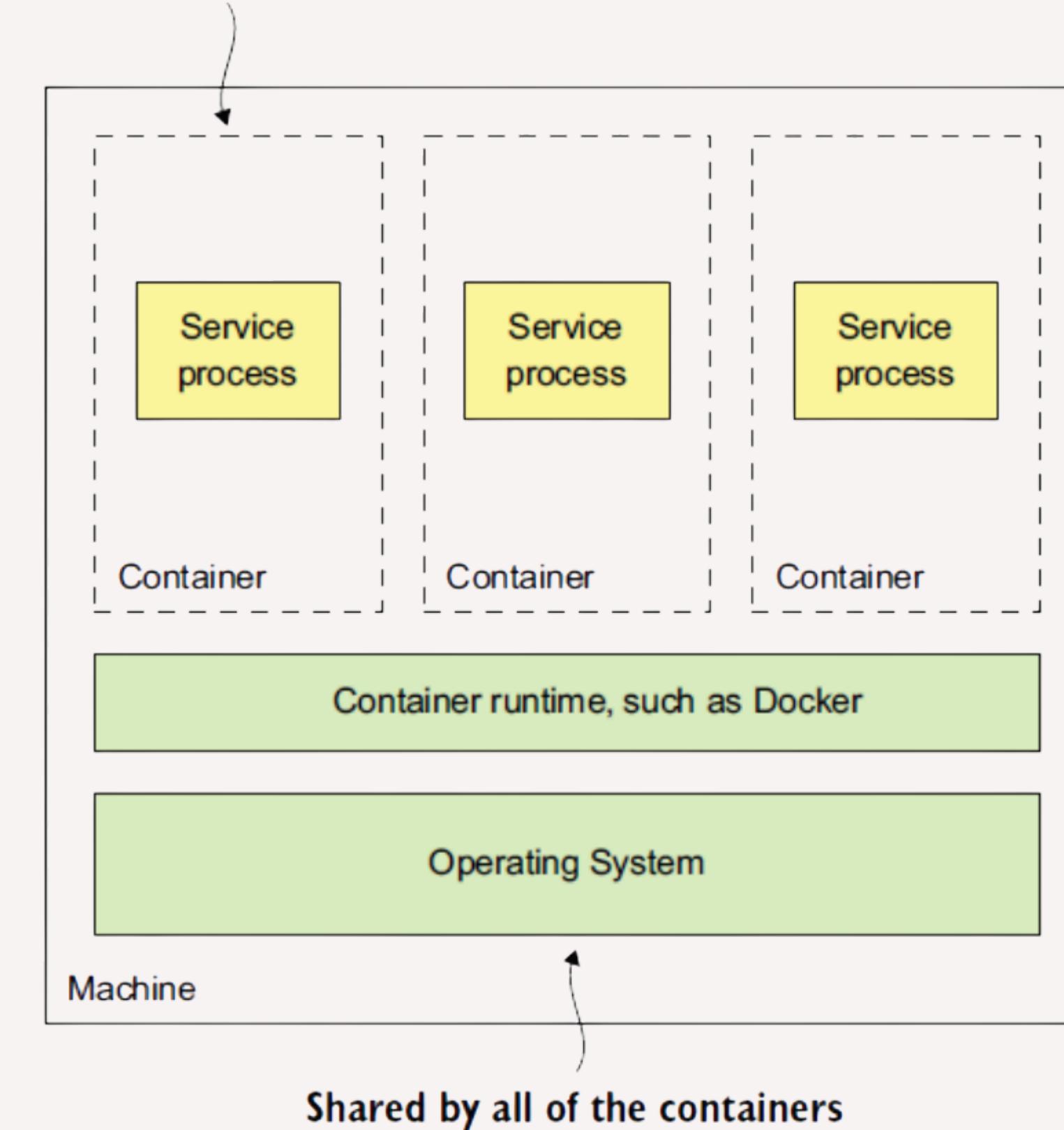
Phân loại

3

Bằng container

Giống máy ảo nhẹ, chỉ chứa phần cần thiết để chạy ứng dụng, tiết kiệm tài nguyên và dễ triển khai.

Each container is a sandbox that isolates the processes.

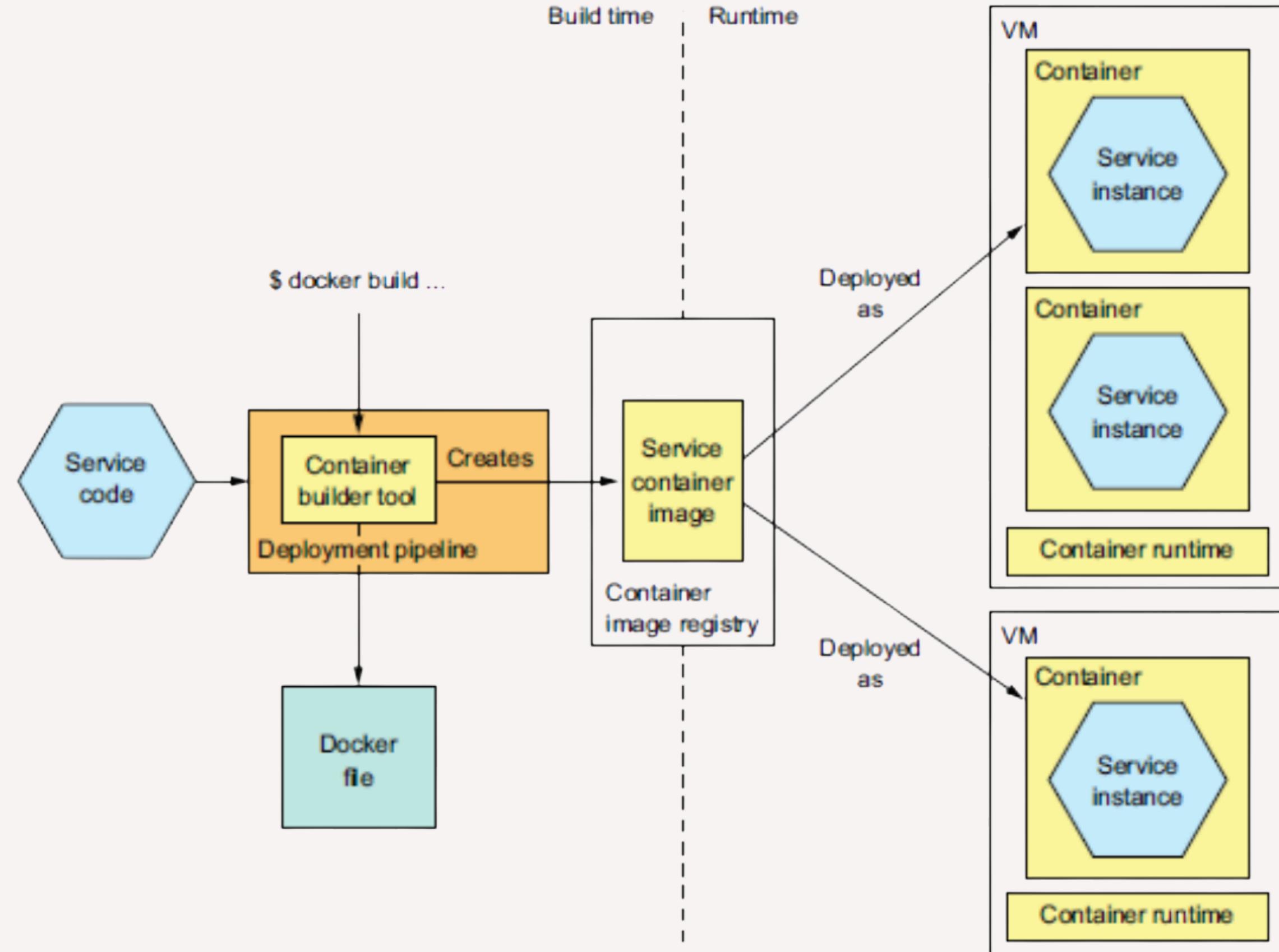


Phân loại

3

Bằng container

Giống máy ảo nhẹ, chỉ chứa phần cần thiết để chạy ứng dụng, tiết kiệm tài nguyên và dễ triển khai.





Phân loại

3



Deploy bằng Docker Container

Deploy bằng Kubernetes

Deploy bằng GitOps

Deploy bằng Docker Container

Bước 1

Bước 2

Bước 3

XÂY DỰNG DOCKER IMAGE

FROM: Chỉ định base image, ví dụ openjdk:8u171-jre-alpine

```
FROM openjdk:8u171-jre-alpine
RUN apk -no-cache add curl
CMD java ${JAVA_OPTS} -jar ftgo-restaurant-service.jar
HEALTHCHECK --start-period=30s --
    interval=5s CMD curl http://localhost:8080/actuator/health || exit 1
```

COPY: Sao chép file thực thi (JAR/WAR) vào image.

HEALTHCHECK: Thiết lập Docker định kỳ kiểm tra tình trạng service (sau 30s, 5s/lần).

CMD hoặc ENTRYPOINT: Chạy lệnh java -jar app.jar khi container khởi động.



Deploy bằng Docker Container

Bước 1

Bước 2

Bước 3

ĐẨY DOCKER IMAGE LÊN REGISTRY

- **Gắn tag cho image:** docker tag ftgo-restaurant-service registry.acme.com/ftgo-restaurant-service:1.0.0.RELEASE
- **Đẩy lên registry:** docker push registry.acme.com/ftgo-restaurant-service:1.0.0.RELEASE





Deploy bằng Docker Container

Bước 1

Bước 2

Bước 3

CHẠY DOCKER CONTAINER

```
docker run \
-d \
--name ftgo-restaurant-service \
-p 8082:8080 \
-e SPRING_DATASOURCE_URL=... -e SPRING_DATASOURCE_USERNAME=...
-e SPRING_DATASOURCE_PASSWORD= \
registry.acme.com/ftgo-restaurant-service:1.0.0.RELEASE
```

Runs it as a background daemon

The name of the container

Binds port 8080 of the container to port 8082 of the host machine

Environment variables

Image to run



Deploy bằng Docker Container

Bước 1

Bước 2

Bước 3

CHẠY DOCKER CONTAINER

```
docker run \
  -d \
  --name ftgo-restaurant-service \
  -p 8082:8080 \
  -e SPRING_DATASOURCE_URL=... -e SPRING_DATASOURCE_USERNAME=... \
  -e SPRING_DATASOURCE_PASSWORD= \
  registry.acme.com/ftgo-restaurant-service:1.0.0.RELEASE
```

Runs it as a background daemon

The name of the container

Binds port 8080 of the container to port 8082 of the host machine

Environment variables

Image to run

Tuy nhiên, docker run có 1 số hạn chế

- Chỉ chạy được trên một máy.
- Không xử lý được lỗi khi máy chủ bị hỏng (mất máy = mất service).
- Docker có cơ chế restart container sau khi crash hoặc reboot, nhưng vẫn là giải pháp đơn lẻ.
- Dịch vụ thường cần chạy cùng các thành phần khác như DB, message broker... => cần deploy đồng bộ.

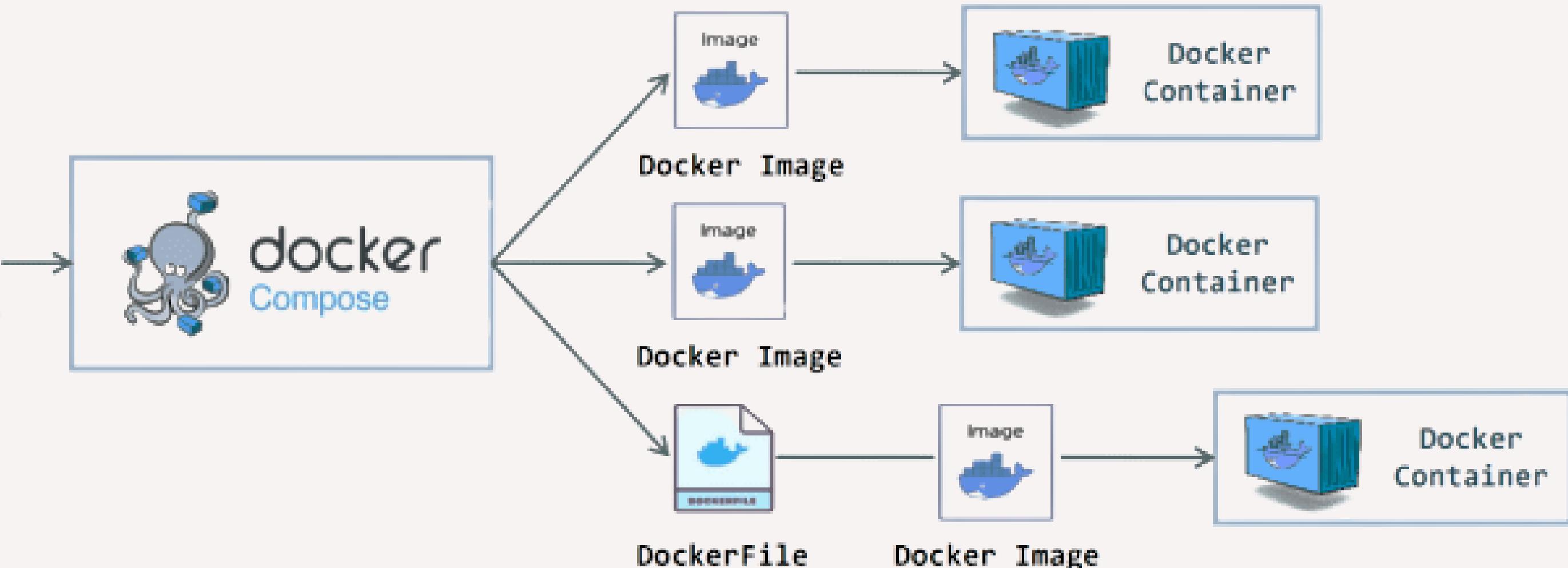
Docker Compose

Deploy bằng Docker Container

DOCKER COMPOSE

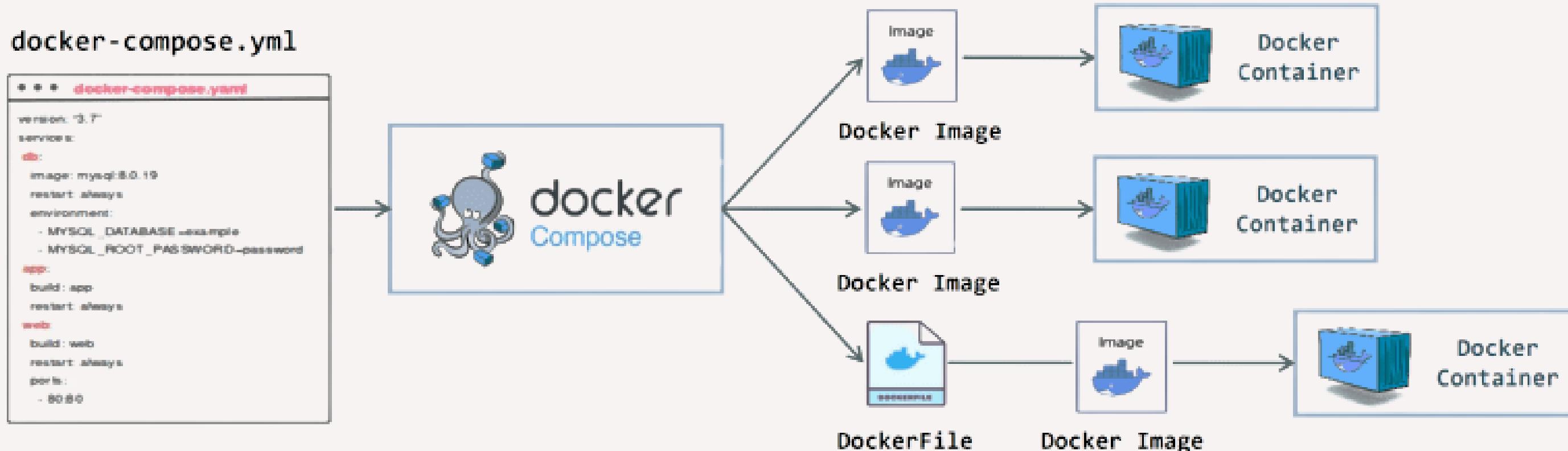
docker-compose.yml

```
* * * docker-compose.yml
version: '3.7'
services:
  db:
    image: mysql:8.0.19
    restart: always
    environment:
      - MYSQL_DATABASE=example
      - MYSQL_ROOT_PASSWORD=password
  app:
    build: app
    restart: always
  web:
    build: web
    restart: always
    ports:
      - 80:80
```



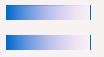
Deploy bằng Docker Container

DOCKER COMPOSE



Chỉ hoạt động trên một máy, không phù hợp môi trường production lớn.

=> **docker orchestration như Kubernetes**



Deploy bằng Kubernetes

Tổng quan

Kiến trúc

Chiến lược

Kubernetes (K8s) là **nền tảng điều phối container**, giúp triển khai, quản lý và mở rộng các dịch vụ một cách tự động, hiệu quả và ổn định.

1

Quản lý tài nguyên

- Xem toàn bộ các máy trong cụm như một khối tài nguyên duy nhất (CPU, RAM, ổ đĩa).
- Biến nhiều máy thành một “máy ảo logic” lớn để dễ phân phối container.

2

Lập lịch (Scheduling)

- Tự động chọn máy thích hợp để chạy container.
- Cân nhắc dựa trên: yêu cầu tài nguyên của container và khả năng sẵn có của từng máy trong cụm.

3

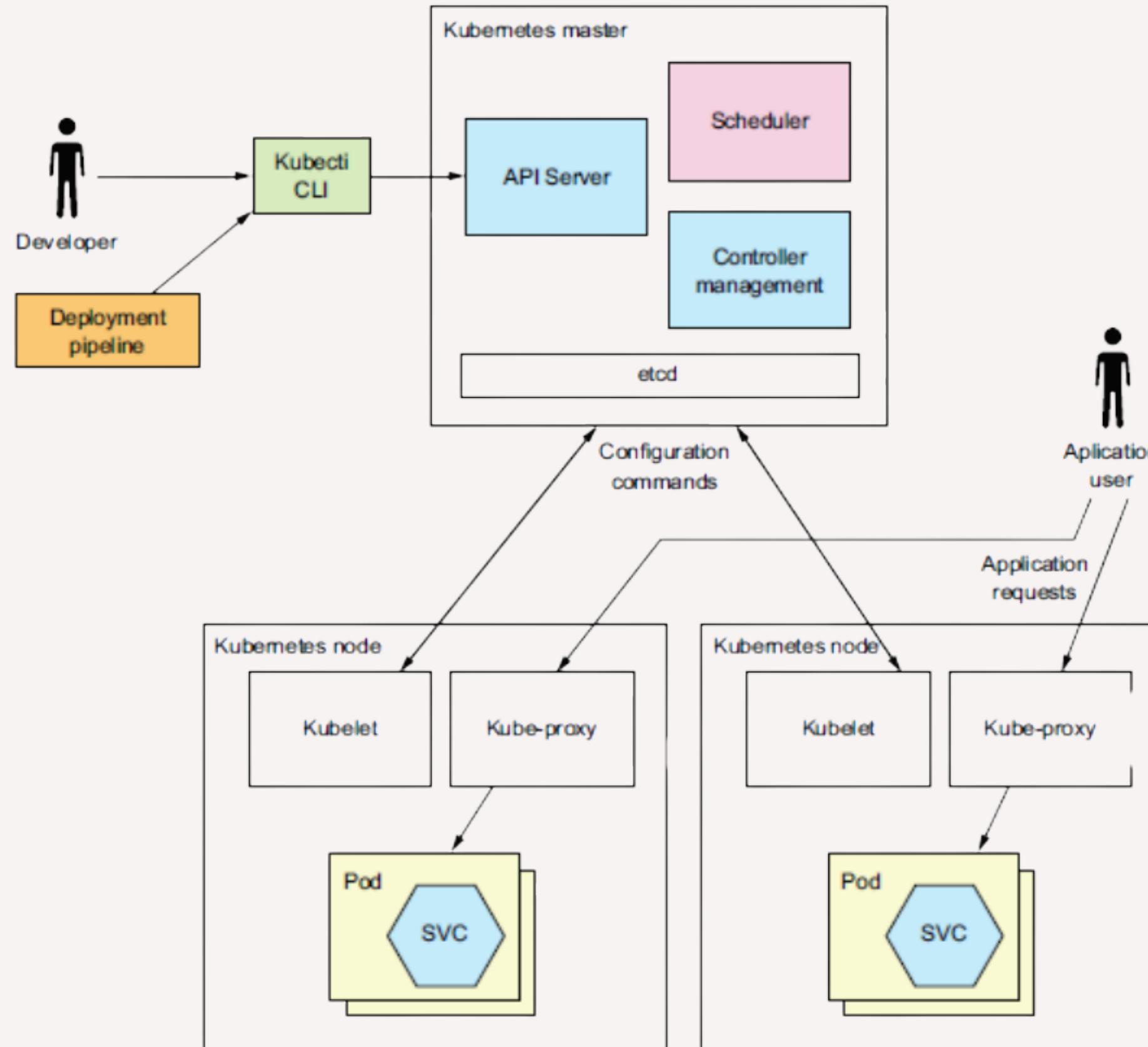
Quản lý dịch vụ

- Đặt tên, version, và điều phối các service.
- Đảm bảo luôn có đủ số lượng instance hoạt động.
- Tự động cân bằng tải, định tuyến request đến container phù hợp.
- Hỗ trợ rolling update và rollback nếu có lỗi trong phiên bản mới.



Deploy bằng Kubernetes

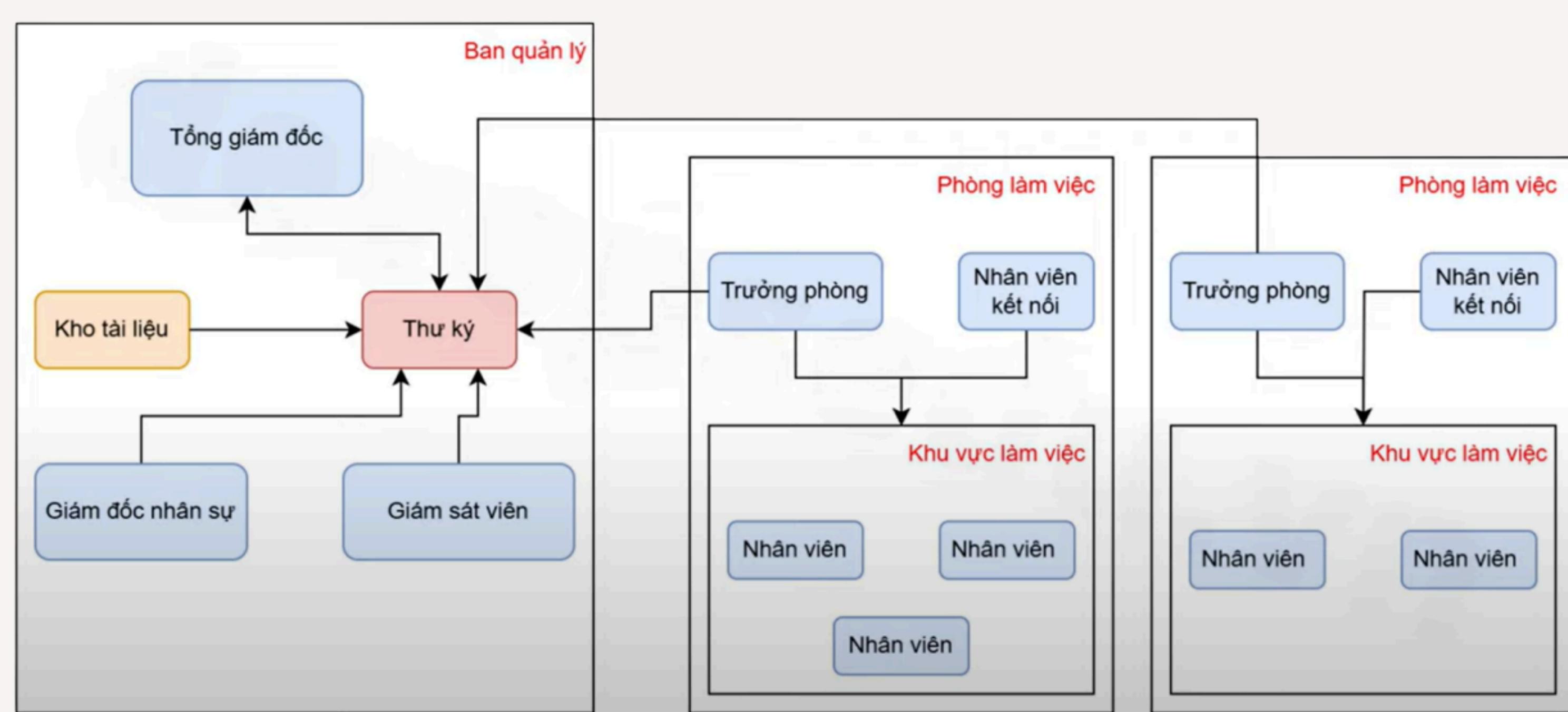
Tổng quan
Kiến trúc
Chiến lược





Deploy bằng Kubernetes

Tổng quan
Kiến trúc
Chiến lược



Deploy bằng Kubernetes

Tổng quan
Kiến trúc
Chiến lược

Basic deployment



- Cập nhật **toàn bộ instance** hiện có.
- **Phải deploy lại tất cả** để có hiệu lực.
- Đơn giản nhưng **dễ gây downtime**.

Deploy bằng Kubernetes

Tổng quan
Kiến trúc
Chiến lược

Multiservice Deployment

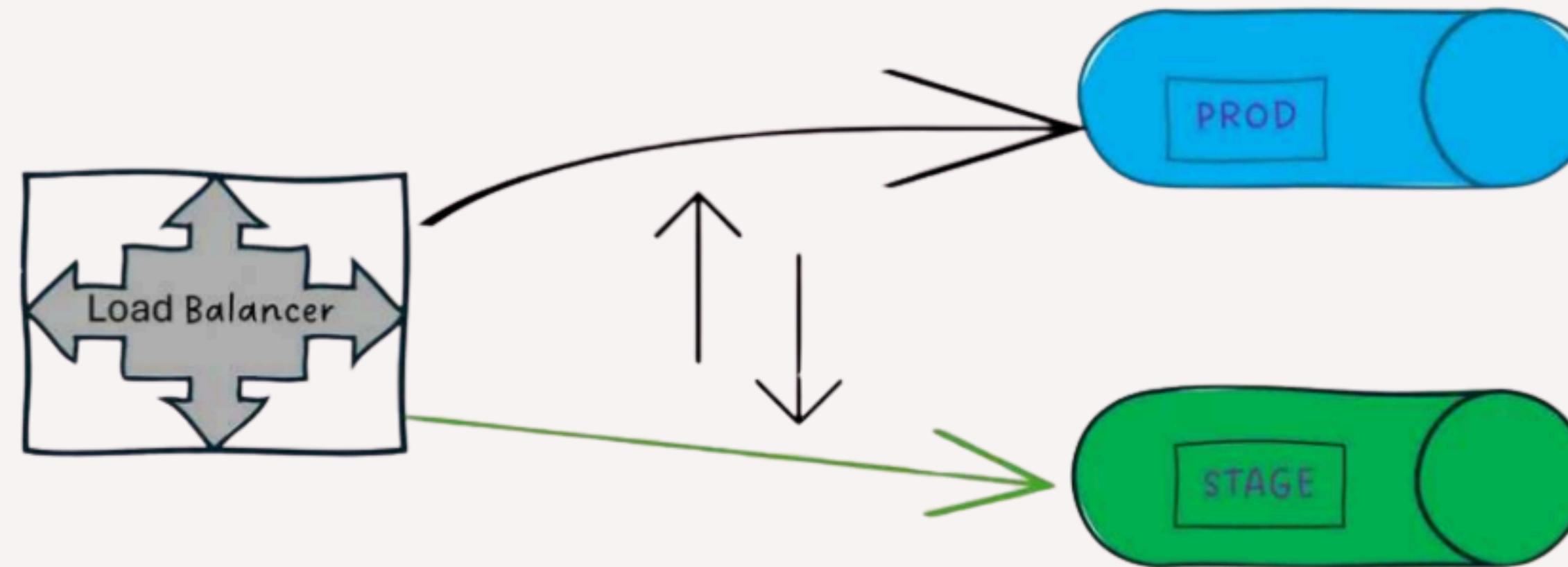


- **Thay thế tất cả instance cùng lúc** bằng phiên bản mới.
- **Rủi ro cao** nếu bản mới lỗi, rollback chậm.

Deploy bằng Kubernetes

Tổng quan
Kiến trúc
Chiến lược

Blue green deployment

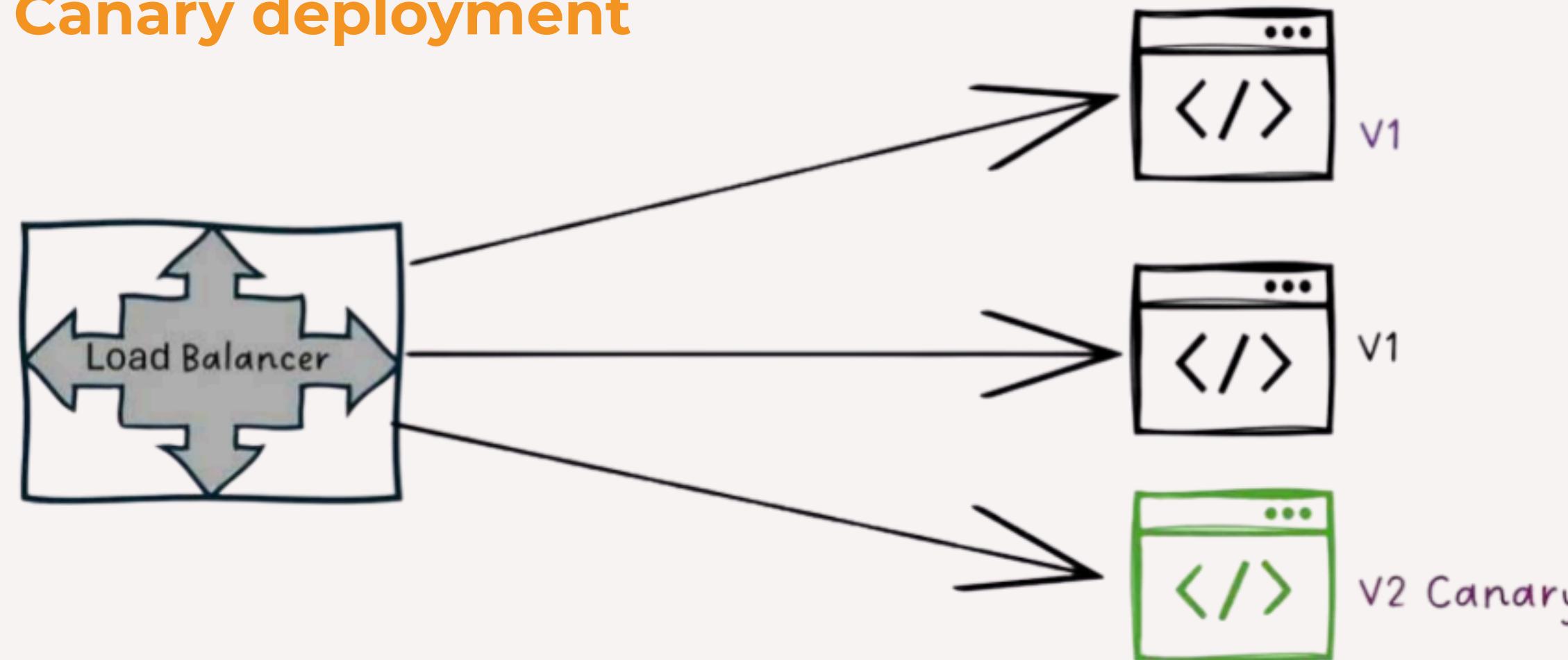


- Có 2 môi trường: **staging (green)** và **production (blue)**.
- Chuyển hướng traffic khi bản mới đã sẵn sàng.
- An toàn nhưng **tốn tài nguyên**.

Deploy bằng Kubernetes

Tổng quan
Kiến trúc
Chiến lược

Canary deployment

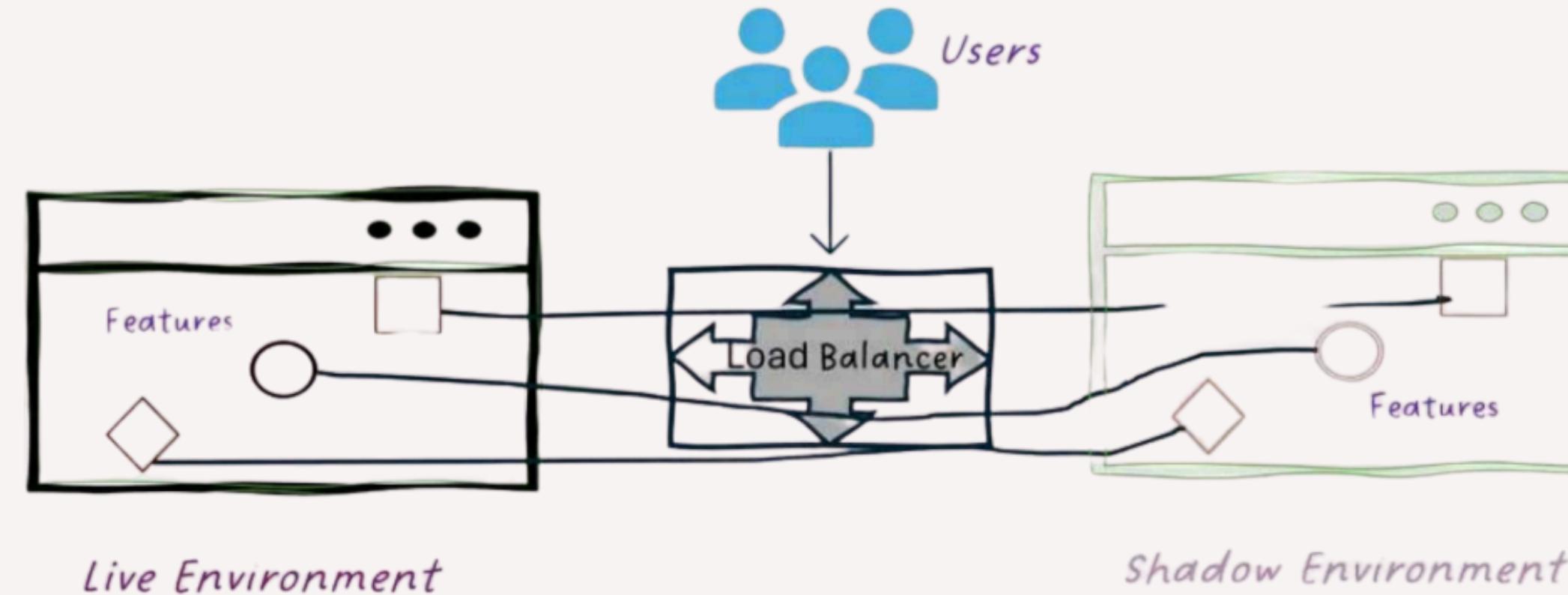


- Chỉ triển khai cho **một nhóm người dùng nhỏ trước**.
- Mở rộng dần nếu ổn định.
- **Dễ rollback**, ít tổn kém hơn blue-green.

Deploy bằng Kubernetes

Tổng quan
Kiến trúc
Chiến lược

Shadow deployment

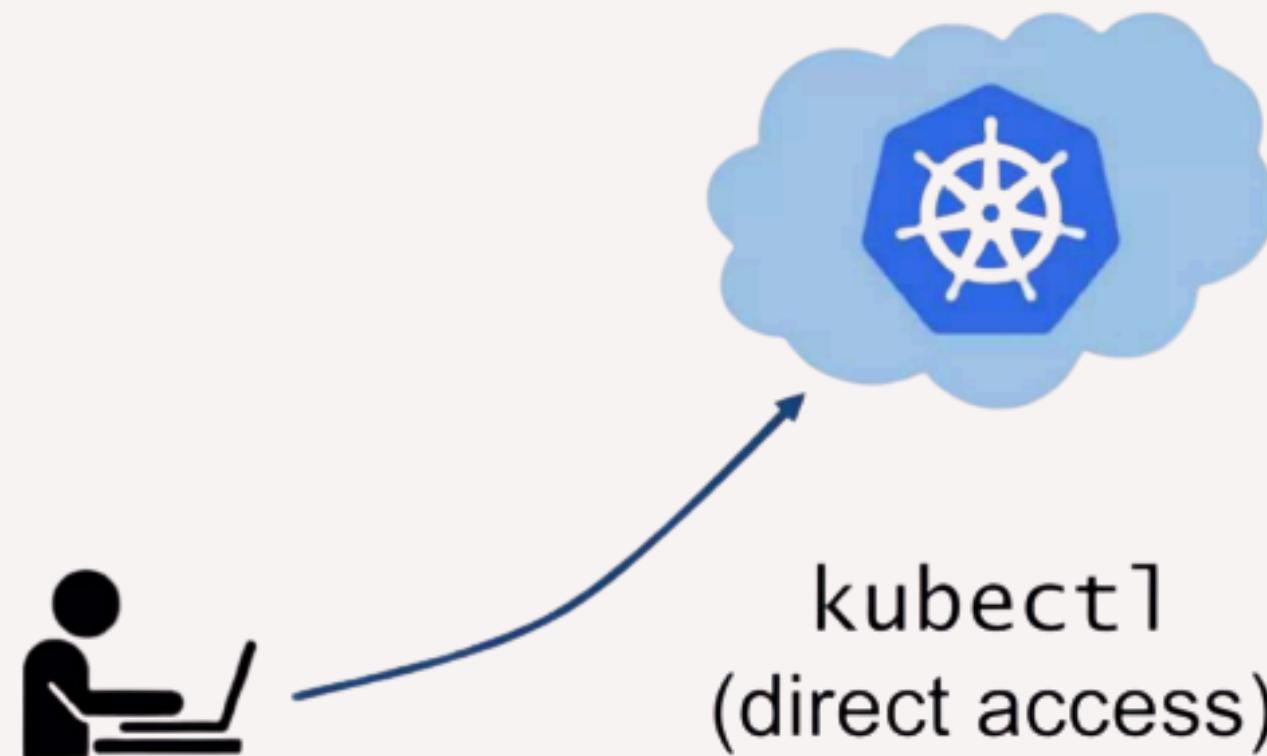


- **Triển khai bản mới nhận toàn bộ traffic** nhưng không trả kết quả cho người dùng.
- Ghi nhận để kiểm thử và phân tích hiệu năng.
- Vận hành song song, **không gây ảnh hưởng thực tế**.

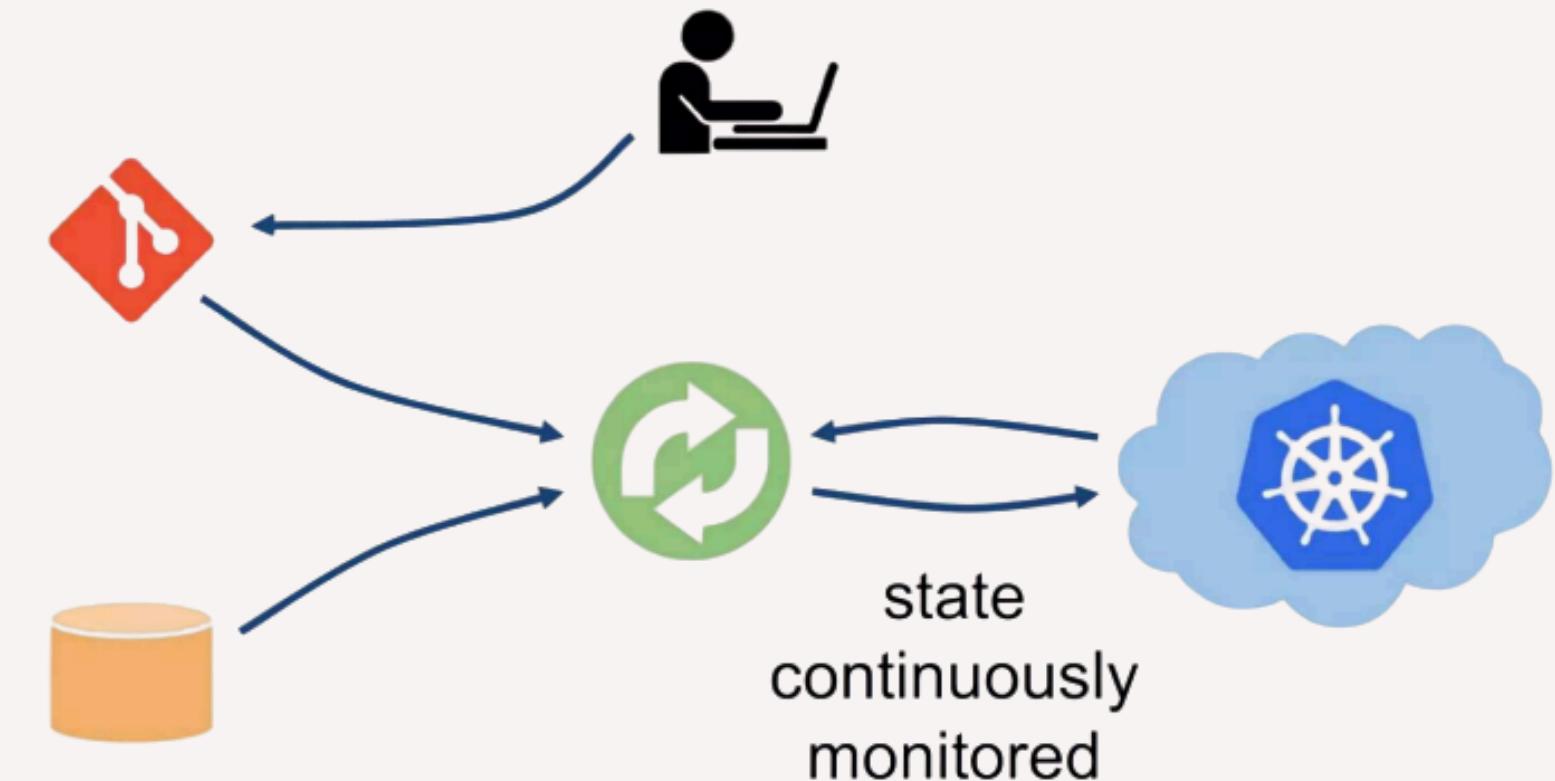
Deploy bằng GitOps

GitOps là một **phương pháp quản lý phiên bản và triển khai hạ tầng** dựa trên nguyên tắc “**mọi thứ là mã**” (everything-as-code), sử dụng Git làm “nguồn sự thật” duy nhất.

Traditional Operation Model



GitOps Operation Model



- Mọi thay đổi đều phải thông qua Git: **Không ai** SSH vào server, không ai chạy thủ công **kubectl apply** hay **terraform apply** ngoài Git.



Deploy bằng GitOps

ArgoCD - GitOps agent

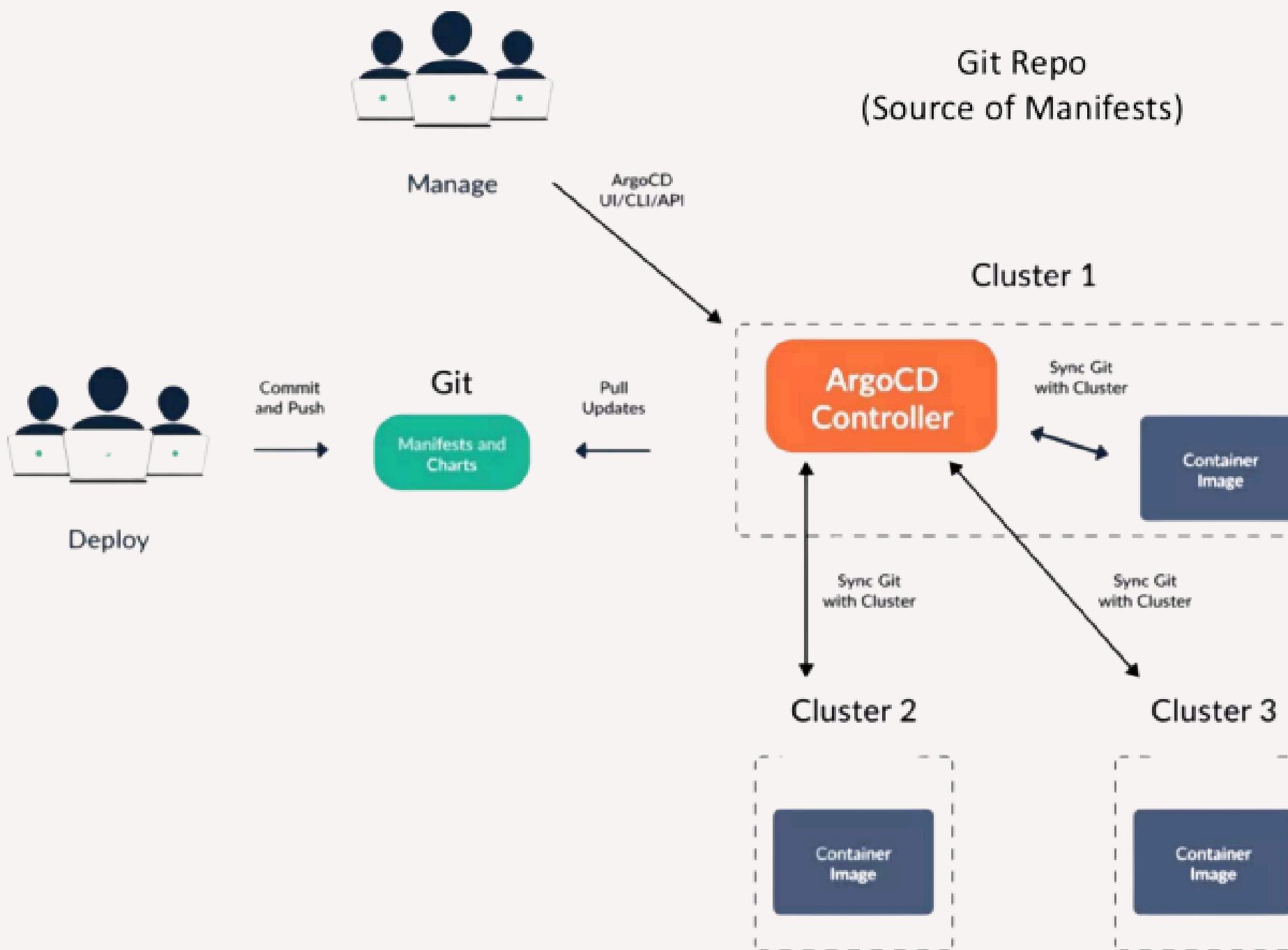
Traditional Model





Deploy bằng GitOps

ArgoCD - GitOps agent

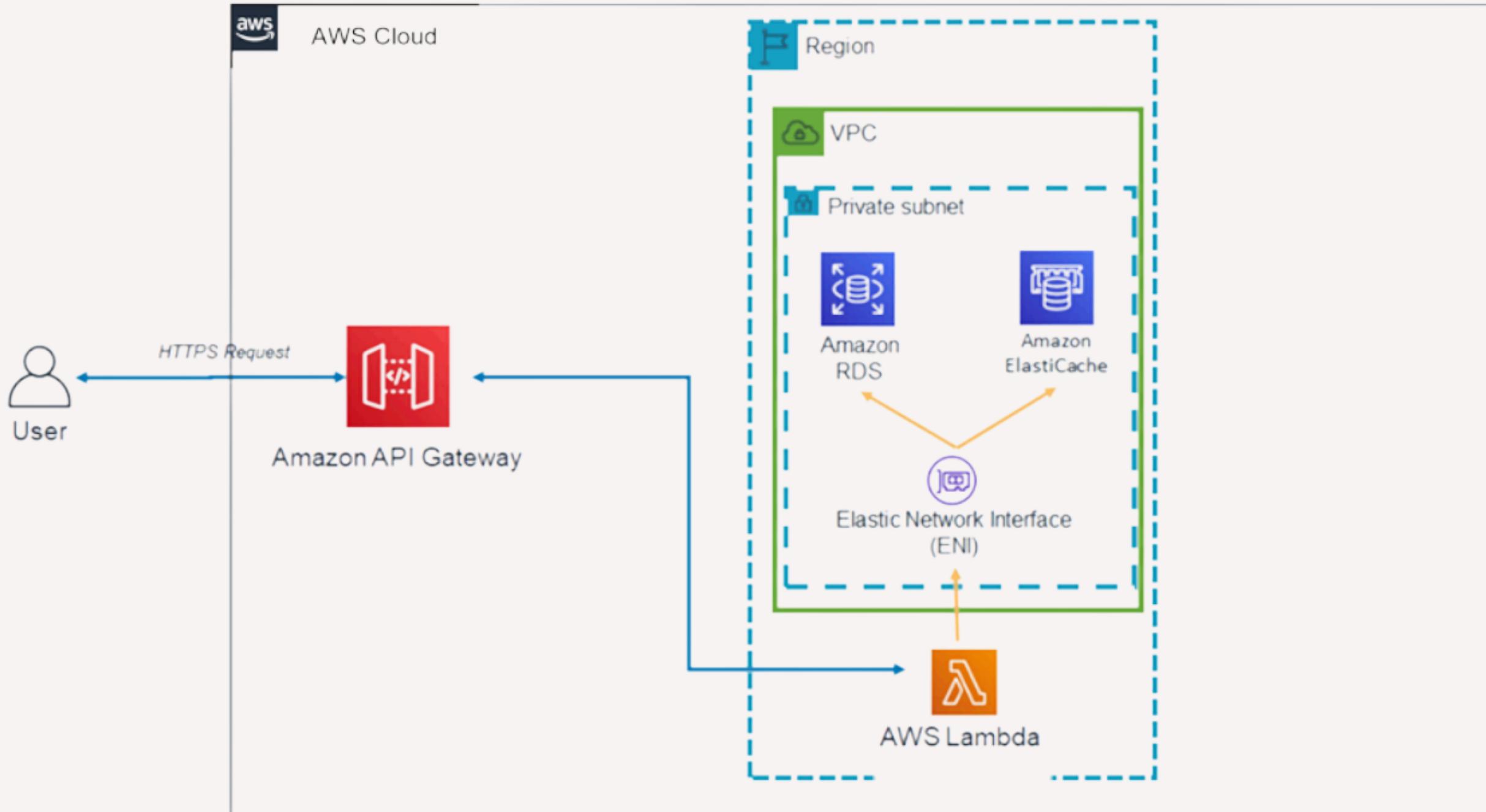




4

Bằng serverless.

Giao toàn bộ việc deploy và vận hành cho cloud provider, tính phí theo mức sử dụng



- Serverless không có nghĩa là "không có máy chủ" – vẫn có máy chủ, nhưng bạn không cần quản lý chúng.
- **Cloud provider (như AWS Lambda, Azure Functions, Google Cloud Functions...)** sẽ lo mọi việc: cấp phát, mở rộng, cập nhật bảo mật.
- **Triển khai theo đơn vị nhỏ:** chỉ cần viết một hàm hoặc microservice nhỏ, định nghĩa trigger (HTTP, queue, cron...) rồi deploy lên nền tảng.

Tài liệu tham khảo

- **Microservices Patterns: With examples in Java First Edition by Chris Richardson (Author):**

https://www.amazon.com/Microservices-Patterns-examples-Chris-Richardson/dp/1617294543/ref=sr_1_15?dib=eyJ2IjoiMSJ9.ACNGezS5dr_lXx94__yxHr7QJE-kBwDXAGWi2NOM1AXq9uSKYlubIdSxnLpIDSt6nMx4rAQM025BLv4FnTt4MI0Ngi8AdWaX2YgikfBZmmOdvQNWVubi8GZPMYRAuif9jNJR1xxPHuHEKACDX2rINuFTnBitOFkwKpqMRXaTmkGHtiy5Eux8ivzGj_OUraW3h5efWRuvma_uZQ2YkeRjPwt81Y50-1w62ZxW3M-5mfwCLgL6m-uNG_hDOgelbnjKxhdDIOM3UoG2s2-FODa76B2Gxp76f6vJLrOfHp-zg.4OJDviQtJyhgCMHwZZpvjwP8X3YqYemiq9bWbaGr3k&dib_tag=se&keywords=Microservices+Architecture&qid=1744520609&sr=8-15

- **GitOps:** https://youtu.be/MeU5_k9ssrs?si=8A8ZGxdK2KvCe7UL

**THANK
YOU**