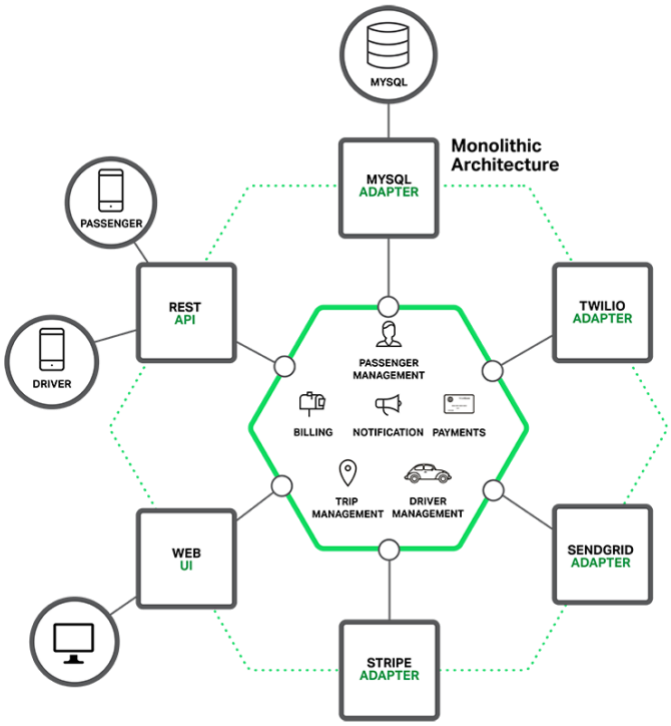
Refactoring a Monolith into Microservices

1/ Introduction to Microservices

Ứng dụng dạng khối (Monolithic Applications)

Ứng dụng dạng khối có kiến trúc gồm 1 khối gắn kết cùng nhau tạo nên 1 ứng dụng duy nhất, bên trong chia thành nhiều thư mục theo chức năng, vai trò.

Monolithic Architecture pattern (microservices.io)



Dù có kiến trúc khá tường minh khi chia thành nhiều modules chức năng và vai trò khác nhau, ứng dụng dạng này vẫn triển khai thành một khối. Mã nguồn xây dựng và chạy ứng dụng có thể phụ thuộc vào một ngôn ngữ lập trình/framework. Ví dụ: C#/ASP.NET và đóng gói thành các tập tin DLL, EXE để triển khai trên server.

Ứng dụng một khối phổ biến khá lâu, từ khi ngành lập trình phần mềm ra đời và dễ xây dựng, thử nghiệm và triển khai.

Góc khuất của kiến trúc khối

Ứng dụng một khối hoạt động rất tốt, cho đến khi quy mô ứng dụng tăng lên, lượng người dùng tăng, các chức năng ngày càng tăng và phức tạp sẽ tạo nên một siêu ứng dụng, một ứng dụng khủng long.

Sau nhiều năm, ứng dụng ban đầu sẽ trở nên cồng kềnh. Không ai thích thay đổi/ bổ sung tính năng cho ứng dụng 1 khối với quy mô lớn. Các thay đổi dù nhỏ cũng ảnh hưởng đến các thành phần khác trong ứng dụng, khi 1 thành phần gây lỗi/ dừng hoạt động sẽ ảnh hưởng đến cả ứng dụng.

Ứng dụng một khối quá lớn, khi viết lại toàn bộ sẽ cần rất nhiều thời gian, nguồn lực, sự dũng cảm và kiên trì. Tình trạng giữ lại thì khổ, xây mới thì khó.

Các thành phần ràng buộc chặt chẽ, sẽ dẫn đến việc phát hiện lỗi tốt thời gian, tiềm ẩn nhiều lỗi ở các góc khuất, tăng thời gian sửa lỗi.

Việc triển khai lên production sẽ tốn thời gian, khởi động chậm, gián đoạn cả ứng dụng đang chạy. Nếu có lỗi trên production sẽ rất khó xác định nguyên nhân, khó cô lập lỗi. Cảm giác xóa đi viết lại mã sẽ nhanh hơn.

Một vài nâng cấp tính năng nhỏ cũng phải triển khai lại toàn bộ ứng dụng.

Khả năng mở rộng chịu tải của các modules chức năng khác nhau trong ứng dụng sẽ không tối ưu. Vd: module xử lý hình ảnh cần tối ưu CPU, module report cần rất nhiều bộ nhớ RAM. Các thành phần này cần triển khai trên các môi trường tối ưu thay vì dùng chung tài nguyên.

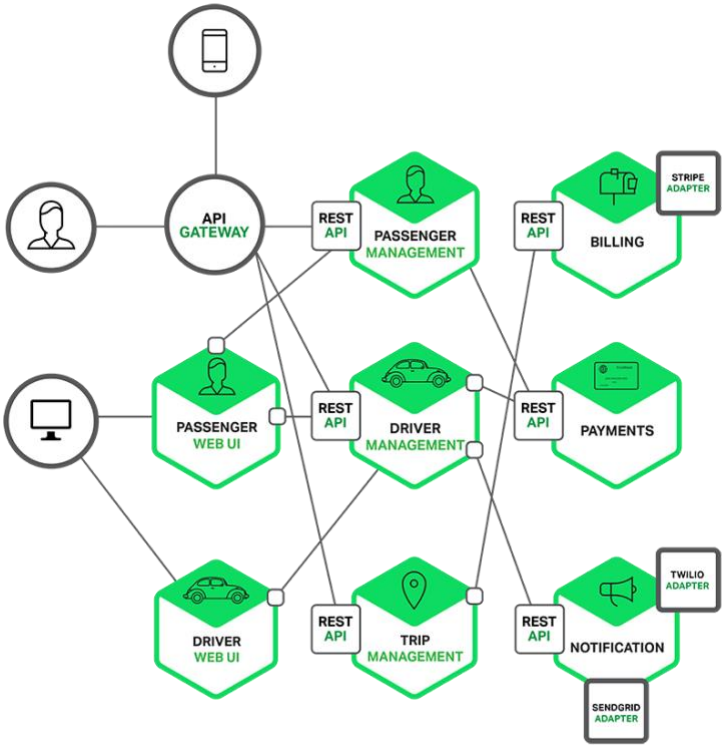
Microservices – chia để trị

Amazon, eBay, Netflix đã giải quyết các vấn đề của ứng dụng một khối bằng kiến trúc microservices. Ý tưởng là chia ứng dụng lớn ra thành các dịch vụ nhỏ được kết nối với nhau.

Mỗi dịch vụ không phải nhỏ quá mức mà là một nhóm chức năng, module cụ thể, độc lập. Vd: quản lý đơn hàng (order), quản lý khách hàng (customer), quản lý tồn kho (inventory).

Các dịch vụ nhỏ sẽ giao tiếp với nhau thông qua APIs hoặc các kỹ thuật trao đổi dữ liệu tiên tiến khác. Khi triển khai, các dịch vụ nhỏ sẽ chạy trong VMs hoặc Docker container, K8S.

Microservice Architecture pattern (microservices.io)



Hình: microservices của ứng dụng gọi xe taxi

Mỗi dịch vụ nhỏ có thể dùng các ngôn ngữ lập trình, công nghệ khác nhau để tối ưu việc tính toán và xử lý tùy theo chức năng và vai trò của dịch vụ.

Để kiến trúc microservices hiệu quả, từng dịch vụ nhỏ sẽ lưu trữ dữ liệu riêng biệt, đảm bảo ít ràng buộc lẫn nhau. Một vài dịch vụ có thể dùng chung 1 CSDL khi cần ưu tiên tính toàn vẹn dữ liệu.

Từng dịch vụ nhỏ có thể lựa chọn công nghệ lưu trữ tối ưu nhất. Vd: dịch vụ điều phối xe cần CSDL hỗ trợ truy vấn theo tọa độ nhanh nhất, dịch vụ cache thì dùng CSDL Redis.

Ưu điểm của microservices

Giảm sự phức tạp khi hệ thống phần mềm ngày càng lớn lên.

Các dịch vụ nhỏ tương đối độc lập, dễ quản lý, bảo trì, nâng cấp. Tự do chọn công nghệ sử dụng, dễ dàng nâng cấp, thay đổi công nghệ mới.

Thúc đẩy tách các khối chức năng ra thành nhiều module rõ ràng hơn. Nếu dùng ứng dụng một khối ta phải dùng design pattern và liên tục tái cấu trúc.

Mỗi dịch vụ nhỏ sẽ do 1 nhóm nhỏ thực hiện, việc xây dựng, kiểm thử độc lập, và triển khai nhanh hơn.

Một số dịch vụ có thể thuê ngoài hoặc giao cho các nhóm khác bên ngoài xây dựng mà vẫn bảo mật hệ thống, mã nguồn cho các dịch vụ còn lại.

Cho phép lựa chọn nhiều ngôn ngữ, công nghệ, loại CSDL dùng sao cho tối ưu nhất với dịch vụ.

Các dịch vụ có thể triển khai nhiều phiên bản, triển khai thử nghiêm nhiều phiên bản cùng lúc.

Nhược điểm của microservices

Nếu chia dịch vụ quá nhỏ (siêu nhỏ) sẽ dẫn đến vụn vặt, khó kiểm soát. Có quá nhiều siêu dịch vụ sẽ làm cho các nhóm chức năng và dữ liệu đi theo phân tán quá mức cần thiết. Tốn nhiều công sức phát triển, việc giao tiếp giữa các dịch vụ sẽ siêu phức tạp khi số dịch vụ tăng quá mức.

Mỗi dịch vụ nhỏ có thể cần có yêu cầu triển khai, tài nguyên, việc mở rộng và giám sát khác nhau. May mắn là docker và k8s giúp việc triển, mở rộng, giám sát đơn giản hóa. Nhưng vẫn cần công sức nhiều nhất cho lần triển khai đầu tiên của từng dịch vụ nhỏ.

2/ Tái cấu trúc 1 Monolith thành Microservices

(Refactoring a Monolith into Microservices)

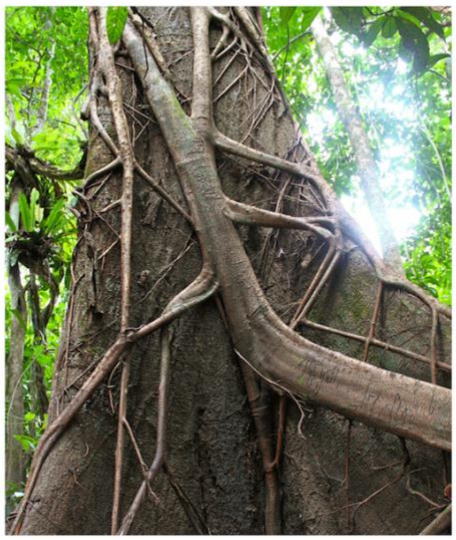
Một chiến lược không nên sử dụng là đập đi xây lại toàn bộ bằng microservices để thay thế – giống như 1 vụ nổ Big Bang. Đó là khi mình tập trung toàn bộ nổ lực để xây dựng lại ứng dụng mới dựa trên microservices từ đầu.

Mặt dù nghe có vẻ hấp dẫn, nhưng lại cực kỳ rủi ro và có khả năng sẽ kết thúc bằng thất bại. Vì thời gian xây dựng mới toàn bộ ứng dụng dựa trên microservices là rất dài và khó theo kịp tốc độ thay đổi của các yêu cầu, tính năng được liên tục bổ sung vào ứng dụng hiện có.

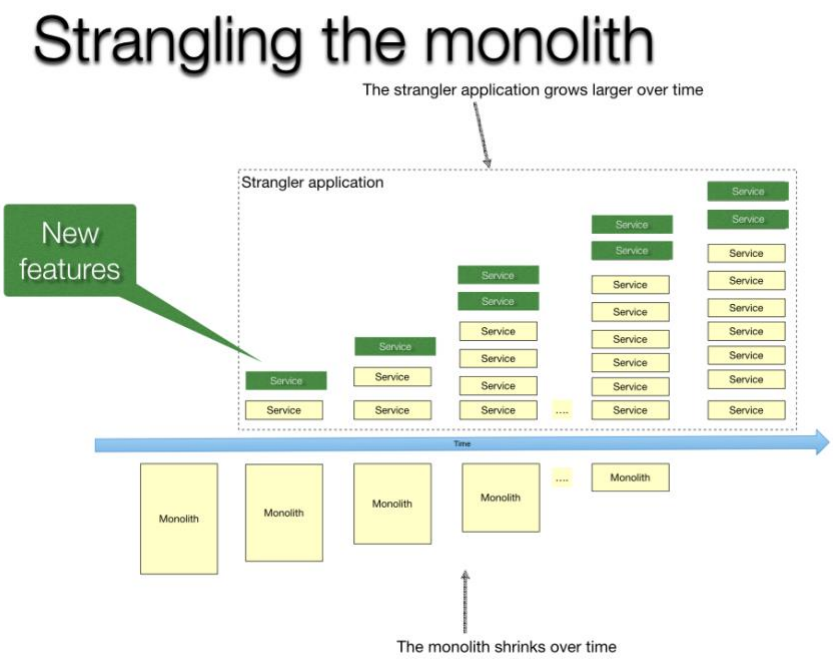
Risks & Responses:

Thay vì viết lại giống như 1 vụ nổ Big Bang, nên chọn giải pháp dần dần tái cấu trúc lại ứng dụng nguyên khối. Dần dần xây dựng các dịch vụ nhỏ và chạy nó cùng với ứng dụng nguyên khối. Theo thời gian, số lượng chức năng dùng trong ứng dụng nguyên khối sẽ giảm dần và chuyển sang thành các dịch vụ nhỏ trong kiến trúc microservices của ứng dụng mới.

Martin Fowler gọi chiến lược này là Strangler Application. Cái tên này xuất phát từ cây nho thắt cổ một cái cây khác được tìm thấy trong rừng nhiệt đới. Một cây nho mọc xung quanh 1 cái cây cổ thụ và vươn lên cao để đón ánh sáng mặt trời trên tán rừng. Đôi khi, cây cổ thụ chết để lại một cây nho hình cây. Chúng ta cũng sẽ xây dựng một ứng dụng mới bao gồm các microservices xung quanh ứng dụng hiện tại, ứng dụng một khối này cuối cùng sẽ chết.



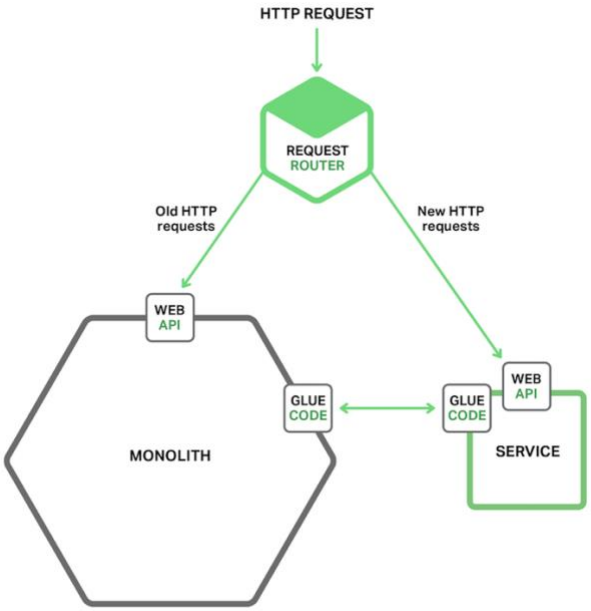
[Strangler application (microservices.io)]



Hãy xem các chiến lược khác nhau để làm điều này

Chiến lược 1 – Ngừng xây mới các chức năng

Chuyển các chức năng mới sang dịch vụ



Một dịch vụ hiếm khi tồn tại biệt lập và thường cần truy cập dữ liệu thuộc sở hữu của nguyên khối.

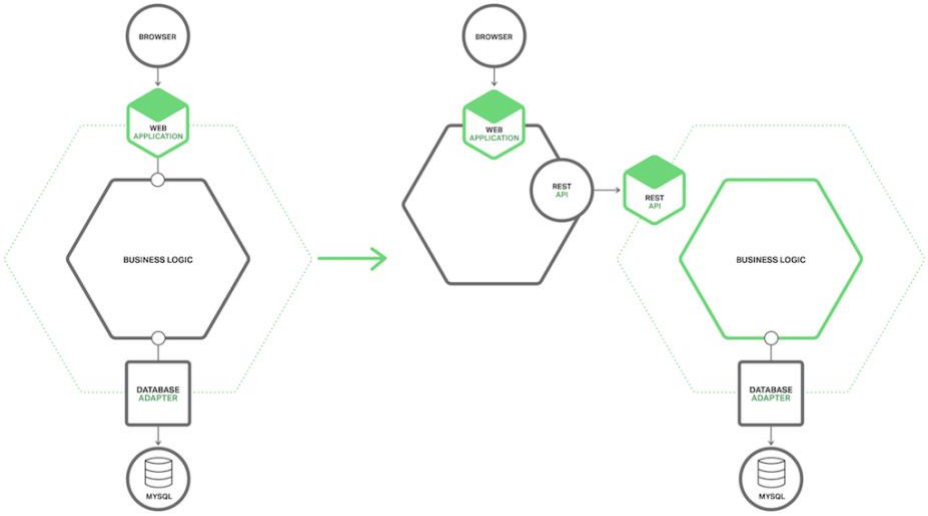
Có ba chiến lược mà một dịch vụ có thể sử dụng để truy cập dữ liệu của nguyên khối:

- Gọi một API từ xa được cung cấp bởi nguyên khối
- Truy cập trực tiếp vào cơ sở dữ liệu của monolith
- Duy trì bản sao dữ liệu của chính nó, được đồng bộ hóa với cơ sở dữ liệu của monolith

Việc triển khai chức năng mới như một dịch vụ sẽ ngăn không cho ứng dụng nguyên khối trở nên khó quản lý. Dịch vụ có thể được xây dựng, triển khai và mở rộng độc lập với nguyên khối.

Tuy nhiên, cách này mới chỉ là khởi đầu, chưa giải quyết được vấn đề với nguyên khối. Mình cần phải phá vỡ nguyên khối.

Chiến lược 2 – Chia Front-end và Back-end



Hai ứng dụng Front-end và Back-end vẫn là 1 nguyên khối.

Chiến lược 3 – Phân tách/giải nén dịch vụ

Chuyển các modules hiện có trong nguyên khối thành các dịch vụ nhỏ tương ứng có quy mô tương đối độc lập. Mỗi lần bạn phân tách được 1 module và biến nó thành 1 dịch vụ nhỏ, nguyên khối ban đầu sẽ thu nhỏ lại. Khi chuyển đủ hết các modules thì nguyên khối ban đầu sẽ hết vai trò sử dụng và sẽ được loại bỏ.

Ưu tiên chuyển đổi module nào thành dịch vụ

Một ứng dụng nguyên khối lớn có hàng chục hoặc hàng trăm modules, tất cả đều là các ứng cử viên để chuyển thành dịch vụ. Tìm ra module nào để chuyển đổi đầu tiên là 1 thách thức.

Một cách tiếp cận là hãy bắt đầu phân tách 1 vài modules tương đối độc lập và rõ ràng bất kỳ. Điều này sẽ cung cấp kinh nghiệm với microservices và quá trình phân tách để từ đó bạn sẽ có chiến lược và sự lựa chọn các modules tiếp theo cần phân tách.

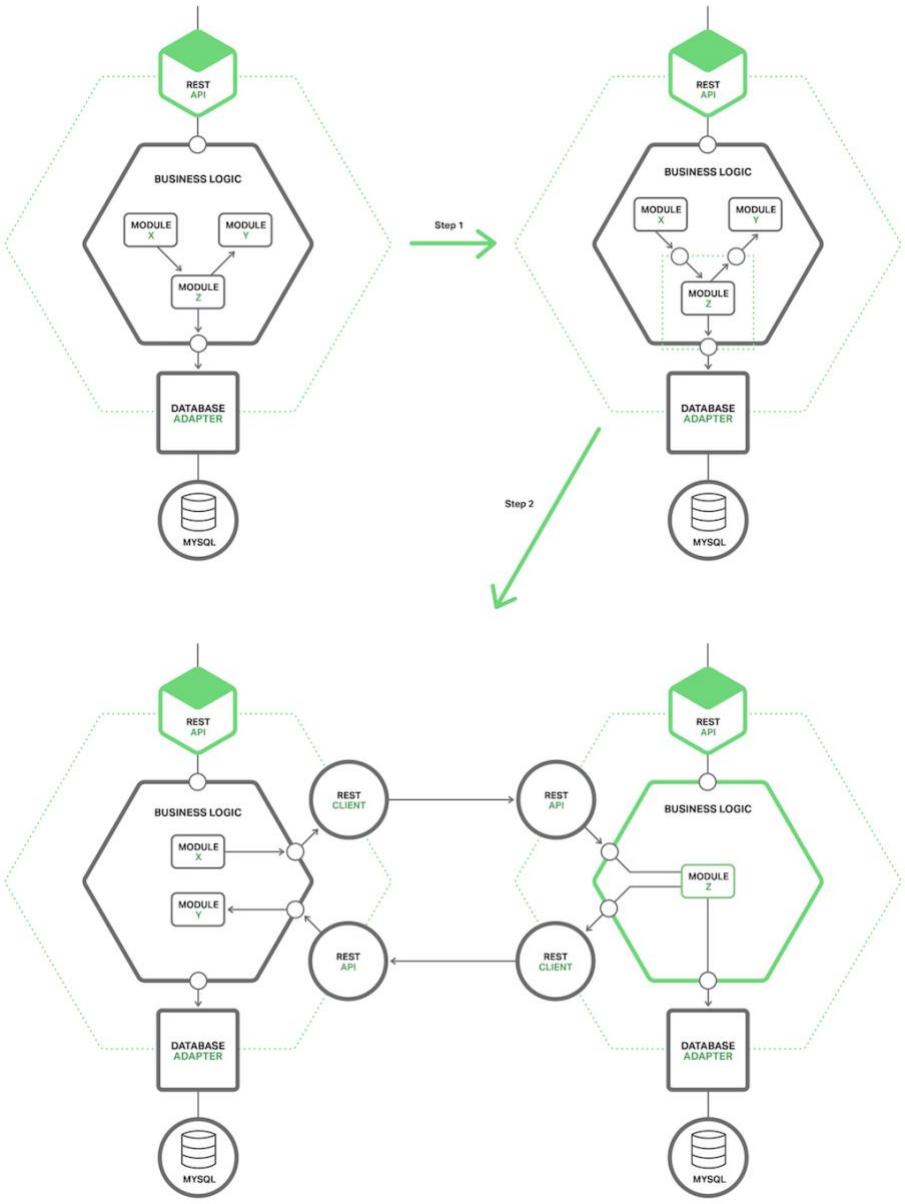
Việc chuyển đổi 1 module thành 1 dịch vụ cũng tốt khá nhiều thời gian. Hãy xếp hạng các module theo lợi ích nhận được và độ lớn của chúng. Thường sẽ có lợi khi phân tách các module thay đổi thường xuyên. Khi đã chuyển 1 module thành 1 service, việc xây dựng, bổ sung tính năng và triển khai sẽ độc lập với nguyên khối, điều này làm giảm độ phức tạp của nguyên khối và đẩy nhanh quá trình chuyển đổi.

Một số hướng dẫn khác về việc tách nguyên khối thành services:

- Decompose by business capability - define services corresponding to business capabilities
- Decompose by subdomain - define services corresponding to DDD subdomains

- Self-contained Service - design services to handle synchronous requests without waiting for other services to respondnew
- Service per teamnew

Sơ đồ cho thấy kiến trúc trước, trong và sau khi phân tách 1 module.



Trong ví dụ, module Z là ứng cử viên. Các thành phần của nó được dùng bởi module X và Y.

Bước đầu tiên là xác định ranh giới của module Z và các module X, Y và các APIs giao tiếp giữa các modules này.

Bước thứ 2 là biến module thành 1 service độc lập. Các tương tác đến và đi của service Z được thực hiện dùng APIs.

Challenges

1/ Đáp ứng tốc độ thay đổi yêu cầu, chức năng lên ứng dụng nguyên khối trong khi xây dựng microservices
=> đã được giải quyết như trên

2/ Các yêu cầu, chức năng mới liên quan nhiều dịch vụ
=> các lựa chọn chia services và có thể phải chấp nhận

3/ Mỗi dịch vụ có CSDL riêng, tuy nhiên có 1 số giao dịch kéo dài trên nhiều dịch vụ. transaction trên nhiều dịch vụ (distributed transaction).
=> có framework để xử lý (Dapr - Microsoft) và dùng message broker/ message bus
[The world is distributed | Microsoft Learn]

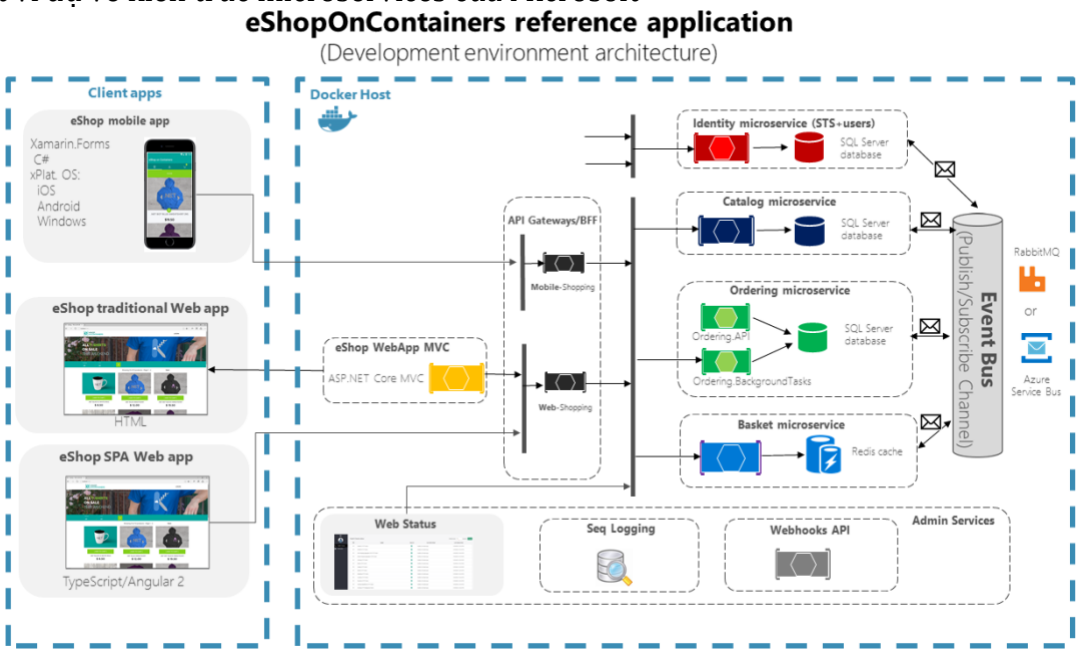
4/

3/ Xây dựng microservices theo chuẩn

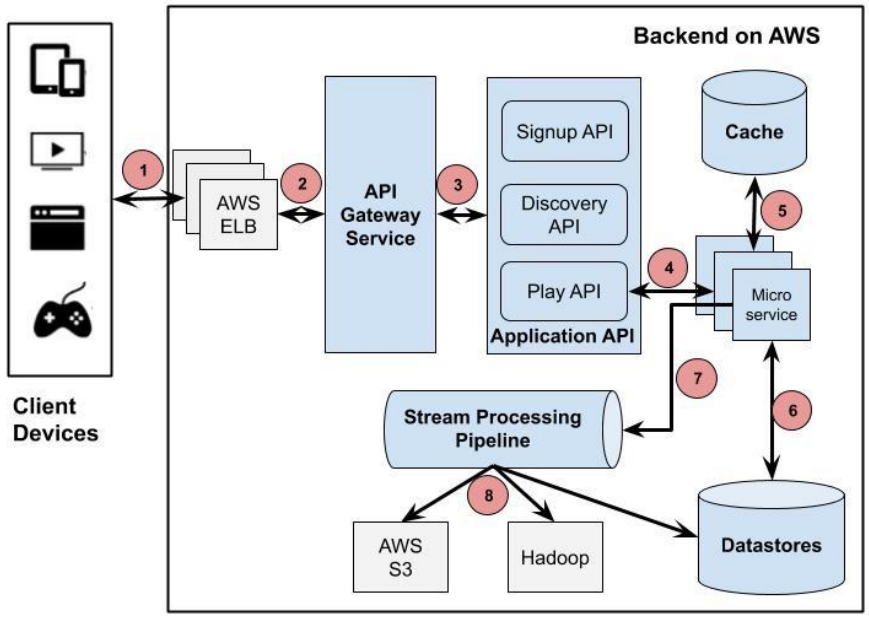
Chọn các chuẩn, các hướng dẫn tin cậy, đã được sử dụng thực tiễn.

Microservices Architecture

Một ví dụ về kiến trúc microservices của Microsoft



Microservices tại Netflix:



[Understanding Design of Microservices Architecture at Netflix (techaheadcorp.com)]
[A Design Analysis of Cloud-based Microservices Architecture at Netflix | by Cao Duc Nguyen | The Startup | Medium]

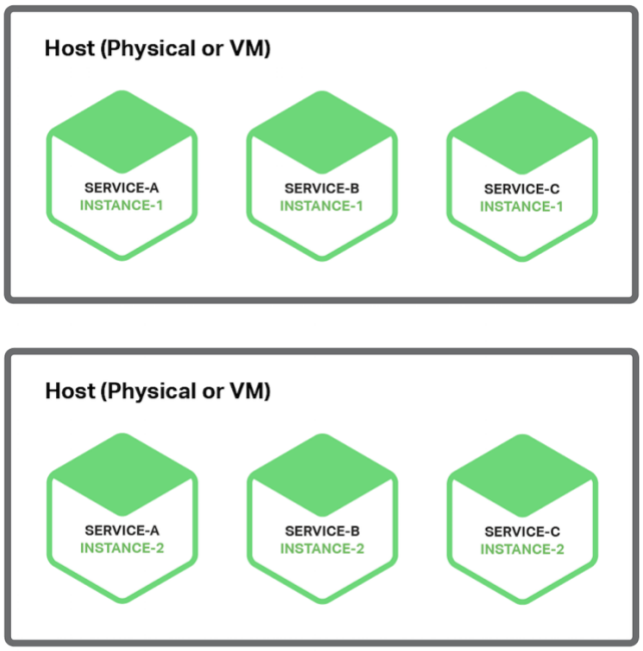
Xây dựng microservices software framework

Các giải pháp, công cụ quản lý, giám sát services

4/ Chiến lược triển khai các services

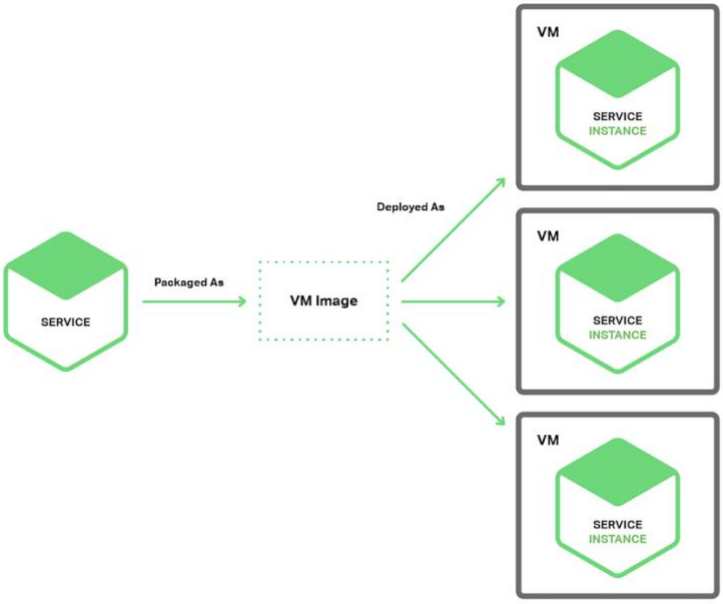
1/ Nhiều instance dịch vụ cho mỗi máy chủ
(Multiple service instances per host pattern)

Đây là cách tiếp cận truyền thống, dễ triển khai. Mỗi instance dịch vụ chạy trên 1 hoặc nhiều host.

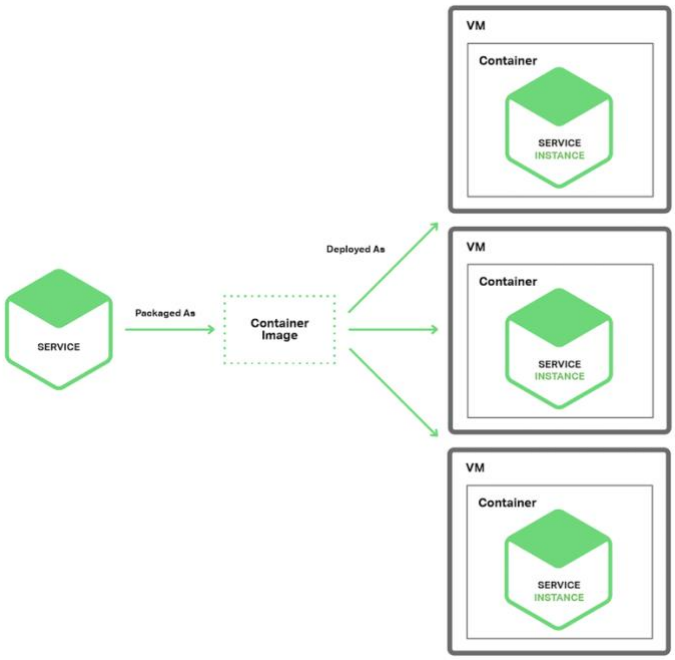


2/ Nhiều instance
(Service instance per host pattern)

Service instance per virtual machine pattern



Service instance per container pattern



5/ Roadmap

Phân tích hiện trạng legacy OSS/BSS

Xác định các thành phần và mô hình của morden/new OSS/BSS

Chuẩn bị kỹ thuật, công nghệ

- Microservice architecture (theo chuẩn nào, tùy biến sao cho phù hợp)
- Xây dựng khung phần mềm (microservice software framework)
- Các công cụ và phương pháp quản trị các thành phần (microservice management platform)

Xác định phạm vi và cách thức chuyển đổi microservices

- Chiến lược chuyển đổi
- Cách chọn các services chuyển đổi
- Rủi ro và phương án dự phòng

Thực hiện chuyển đổi

....