

TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN  
KHOA CÔNG NGHỆ THÔNG TIN

NGUYỄN TRẦN HẬU - NGUYỄN CHÍ THỨC

NỀN TẢNG TÍCH HỢP  
DỊCH VỤ THÔNG MINH

KHÓA LUẬN TỐT NGHIỆP CỦ NHÂN  
CHƯƠNG TRÌNH CỦ NHÂN TÀI NĂNG

Thành phố Hồ Chí Minh, 2020

TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN  
KHOA CÔNG NGHỆ THÔNG TIN

NGUYỄN TRẦN HẬU - 1612180  
NGUYỄN CHÍ THỨC - 1612677

NỀN TẢNG TÍCH HỢP  
DỊCH VỤ THÔNG MINH

KHÓA LUẬN TỐT NGHIỆP CỦ NHÂN  
CHƯƠNG TRÌNH CỦ NHÂN TÀI NĂNG

GIÁO VIÊN HƯỚNG DẪN  
PGS.TS TRẦN MINH TRIẾT

Thành phố Hồ Chí Minh, 2020

# Nhận xét của giáo viên hướng dẫn

## Khóa luận đáp ứng yêu cầu của Luận văn cử nhân tin học.

TP.HCM, ngày ..... tháng ..... năm 2020

## Giáo viên hướng dẫn

# Nhận xét của giáo viên phản biện

## Khóa luận đáp ứng yêu cầu của Luận văn cử nhân tin học.

TP.HCM, ngày ..... tháng ..... năm 2020

Giáo viên phản biện

# **Lời cảm ơn**

Chúng em xin chân thành cảm ơn Khoa Công Nghệ Thông Tin, trường Đại Học Khoa Học Tự Nhiên, Tp.HCM đã tạo điều kiện tốt cho chúng em thực hiện đề tài này.

Chúng em xin chân thành cảm ơn Thầy Trần Minh Triết, là người đã truyền đạt những kiến thức, niềm tin, cảm hứng và những lời khuyên bổ ích cho chúng em, đóng góp phần to lớn giúp chúng em có thể hoàn thành đề tài này.

Chúng em xin cảm ơn các anh Võ Huỳnh Anh Huy, anh Đoàn Văn Việt, anh Lê Trọng Nghĩa, anh Lê Thái Phúc Quang là người hướng dẫn chúng em trên công ty đã giúp chúng em biết được nhiều kiến thức hay, truyền đạt cho chúng em về các bài toán thực tế trong doanh nghiệp, dựa vào điều đó đã giúp đỡ chúng em rất nhiều trong quá trình thực hiện đề tài.

Chúng em xin gửi lòng biết ơn đến Ba, Mẹ, các anh chị, bạn bè, đồng nghiệp đã ủng hộ, giúp đỡ và động viên chúng em trong những lúc khó khăn trong quá trình thực hiện đề tài này. Mặc dù đã có cố gắng hoàn thành đề tài một cách tốt nhất, nhưng chắc rằng chúng em sẽ có những thiếu sót, kính mong sự cảm thông và chỉ bảo của Thầy Cô.

**Nhóm thực hiện**

Nguyễn Trần Hậu

Nguyễn Chí Thức

# Đề cương chi tiết

Tên đề tài: Nền tảng tích hợp dịch vụ thông minh.
Giảng viên hướng dẫn: PGS.TS. Trần Minh Triết
Thời gian thực hiện: 02/2020 - 07/2020
Sinh viên thực hiện: Nguyễn Trần Hậu (1612180) - Nguyễn Chí Thức (1612677)
Loại đề tài: Xây dựng và ứng dụng công nghệ vào kiến trúc phần mềm.
Nội dung đề tài (mô tả chi tiết nội dung đề tài, yêu cầu, phương pháp thực hiện, kết quả đạt được, ...):
<ul style="list-style-type: none"><li>• Nghiên cứu vấn đề thực tại mà các lập trình viên thường gặp và các hướng giải quyết.</li><li>• Khảo sát các công nghệ được các công ty liên quan, có kinh nghiệm về phát triển hệ thống nhiều người dùng đã và đang phát triển thành công.</li><li>• Tìm hiểu các công nghệ và ứng dụng vào hệ thống.</li><li>• Thiết kế kiến trúc đáp ứng nhu cầu có thể tích hợp các service vào hệ thống, đảm bảo hiệu năng cho các service tham gia vào kiến trúc.</li></ul>

### **Kế hoạch thực hiện:**

- 01/02 - 01/03: Tìm hiểu các vấn đề cần giải quyết của đề tài và giải pháp của đề tài.
- 01/03 - 01/04: Thiết kế hệ thống và kiến trúc toàn bộ hệ thống.
- 01/04 - 15/05: Tìm hiểu các công nghệ giải quyết.
- 15/05 - 15/07: Xây dựng các services trong hệ thống Platform.
- 15/07 - 01/08: Tích hợp các services trong kiến trúc lại với nhau.
- 01/08 - 15/08: Nâng cấp, chỉnh sửa và hoàn thiện các ứng dụng đã xây dựng.

<b>Xác nhận của GVHD</b>	<b>Ngày 15 tháng 08 năm 2020</b> <b>Nhóm SV thực hiện</b>
	Ngyễn Trần Hậu - Nguyễn Chí Thức

# Mục lục

Danh sách hình	viii
Danh sách bảng	xii
<b>1 Mở đầu</b>	<b>1</b>
1.1 Giới thiệu chung . . . . .	1
1.2 Lý do thực hiện đề tài . . . . .	3
1.3 Mục tiêu đề tài . . . . .	4
1.4 Nội dung đề tài . . . . .	5
<b>2 Các vấn đề và giải pháp khi thiết kế</b>	<b>6</b>
2.1 Các services trong kiến trúc . . . . .	6
2.1.1 Vấn đề . . . . .	6
2.1.2 Kiến trúc Microservice là gì? . . . . .	7
2.1.3 Các tính chất cần lưu ý của các service khi sử dụng kiến trúc Microservice . . . . .	18
2.1.4 Ưu điểm và hạn chế của kiến trúc Microservice . . . . .	19
2.2 Các phương thức giao tiếp giữa các trong kiến trúc Microservice . . . . .	20
2.2.1 Cơ chế giao tiếp giữa các service . . . . .	21
2.2.2 Cách thực hiện cơ chế Asynchronous . . . . .	23
2.2.3 Sử dụng gRPC để giao tiếp giữa các service . . . . .	25
2.3 Bộ nhớ đệm Cache . . . . .	30
2.3.1 Vấn đề . . . . .	30

2.3.2	Redis là gì? . . . . .	31
2.3.3	Ưu điểm . . . . .	31
2.3.4	Các kiểu nổi bật dữ liệu của Redis . . . . .	34
2.3.5	Cách sử dụng đơn giản với Redis-cli . . . . .	36
2.3.6	Kết luận . . . . .	37
2.4	Message queue . . . . .	37
2.4.1	Vấn đề . . . . .	37
2.4.2	Kafka là gì? . . . . .	38
2.4.3	Cách hoạt động của Kafka . . . . .	40
2.4.4	Tại sao sử dụng Kafka . . . . .	42
2.4.5	Các trường hợp sử dụng . . . . .	43
2.4.6	Kết luận . . . . .	43
2.5	Container . . . . .	43
2.5.1	Vấn đề . . . . .	43
2.5.2	Mô tả . . . . .	44
2.5.3	Kế thừa . . . . .	45
2.5.4	Lợi ích . . . . .	46
2.5.5	Hạn chế . . . . .	46
2.5.6	Docker . . . . .	49
2.6	AI Platform . . . . .	51
2.6.1	Vấn đề . . . . .	51
2.6.2	Giải pháp . . . . .	51
2.7	Lưu trữ . . . . .	53
2.8	Scale . . . . .	58
2.8.1	Vấn đề . . . . .	58
2.8.2	Mô tả . . . . .	58
2.8.3	Kỹ thuật . . . . .	60
<b>3</b>	<b>Kiến trúc AI Platform</b>	<b>63</b>
3.1	Tổng quan . . . . .	63
3.2	Kiến trúc của Backend . . . . .	63

3.2.1	Ưu điểm . . . . .	64
3.2.2	Nhược điểm . . . . .	65
3.2.3	Tích hợp . . . . .	66
3.3	Service User . . . . .	67
3.4	Kiến trúc hệ thống Director . . . . .	68
3.5	Kiến trúc hệ thống Businesss . . . . .	69
3.5.1	Business gateway . . . . .	70
3.5.2	Storage service . . . . .	71
3.5.3	Thiết kế đóng gói cho AI service . . . . .	74
<b>4</b>	<b>Một số kịch bản xây dựng</b>	<b>78</b>
4.1	Kiểm tra chính tả sau khi nhận dạng chữ từ ảnh . . . . .	78
4.2	Kiểm tra mật độ xe của giao thông thành phố Hồ Chí Minh	83
<b>5</b>	<b>Kết luận</b>	<b>91</b>
5.1	Các kết quả đạt được . . . . .	91
5.2	Hướng phát triển của đề tài . . . . .	92
<b>Tài liệu tham khảo</b>		<b>93</b>

# Danh sách hình

1.1	Ảnh chụp màn hình nhận diện con người của hệ thống AI trường Oxford. . . . .	2
1.2	Nhận dạng chữ trong biển báo giao thông bằng phần mềm AI của Google. . . . .	3
1.3	Sơ đồ thực hiện theo tuần tự ở các service . . . . .	4
2.1	Nhược điểm khi phát triển phần mềm ứng dụng AI . . . . .	7
2.2	Kiến trúc của AI platform . . . . .	8
2.3	Chi tiết kiến trúc của Director . . . . .	9
2.4	Kiến trúc gọi trực tiếp . . . . .	10
2.5	Client app gửi yêu cầu lên Gateway. . . . .	13
2.6	Minh họa hacker tấn công. . . . .	14
2.7	Kiến trúc API Gateway . . . . .	16
2.8	Kiến trúc nhiều API Gateway . . . . .	17
2.9	Mô tả message broker . . . . .	23
2.10	kiến trúc Asynchronous cho service . . . . .	24
2.11	Hình ảnh minh họa việc giao tiếp của các service sử dụng gRPC . . . . .	26
2.12	Hình ảnh so sánh cơ chế hoạt động tối ưu hơn HTTP/2 và HTTP/1.1 . . . . .	27
2.13	Hình ảnh mô tả việc sử dụng bộ nhớ đệm trong hệ thống .	30
2.14	Kiến trúc Replication của redis . . . . .	32
2.15	Cơ chế hoạt động Persistence của Redis theo kiểu AOF . .	33

2.16	Hình ảnh mô tả cách hoạt động của Message Queue . . . . .	38
2.17	Hình ảnh mô tả tổng quát cơ chế hoạt động của Kafka . . . . .	39
2.18	Mối quan hệ giữa Producer, cluster và Consumer ở Kafka . . . . .	40
2.19	Cách tổ chức dữ liệu các record trên Broker . . . . .	41
2.20	Cơ chế hoạt động Smart Consumer ở Kafka . . . . .	41
2.21	Các ưu điểm nổi trội của Kafka . . . . .	42
2.22	So sánh giữa máy ảo và container . . . . .	47
2.23	So sánh lợi ích của Container và máy ảo . . . . .	49
2.24	So sánh thiết kế của Container và máy ảo . . . . .	50
2.25	Danh sách các Docker container đang chạy trên máy . . . . .	50
2.26	Giao diện của MinIO . . . . .	54
2.27	Tổng quan kiến trúc AI Platform thuở sơ khai . . . . .	56
2.28	Tổng quan 3 chiều scale . . . . .	59
3.1	Các service trong hệ thống Platform . . . . .	64
3.2	Coordinator quản lý điều phối đến các service . . . . .	65
3.3	Component system của User Service . . . . .	67
3.4	Tổng quan kiến trúc hệ thống Director . . . . .	68
3.5	Tổng quan kiến trúc hệ thống Business . . . . .	69
3.6	Kiến trúc storage service . . . . .	71
3.7	Quá trình xử lý của Storage service . . . . .	73
3.8	Mô tả việc đóng gói của service AI . . . . .	75
3.9	Kiến trúc của AI Service . . . . .	76
3.10	Luồng xử lý của Handler . . . . .	77
3.11	Luồng xử lý của Processor . . . . .	77
4.1	Sơ đồ hệ thống với OCR và Language tool . . . . .	79
4.2	Cơ chế Asynchronous cho Task manager service . . . . .	80
4.3	Hình ảnh mô tả quá trình xử lý trên từng service . . . . .	81
4.4	Hình ảnh cho việc demo trên hệ thống . . . . .	82
4.5	Kết quả xử lý trên giao diện người sử dụng . . . . .	82
4.6	Kết quả xử lý của ảnh 4.4 . . . . .	83

4.7	Kẹt xe ở một đoạn đường tại TP.HCM . . . . .	84
4.8	Hệ thống nhận diện xe hơi tham gia giao thông . . . . .	85
4.9	Service HCMTraffic . . . . .	86
4.10	Chạy service HCMTraffic trên terminal . . . . .	86
4.11	Kết quả đầu ra của service HCMTraffic . . . . .	87
4.12	Hệ thống đếm số lượng xe hơi tham gia giao thông . . . . .	88
4.13	Hệ thống nhận diện người tham gia giao thông . . . . .	88
4.14	Hệ thống đếm số lượng người tham gia giao thông . . . . .	89
4.15	Mô tả min scale và max scale . . . . .	90

# Danh sách bảng

2.1 Bảng so sánh giữa gRPC và HTTP API . . . . .	28
--	----

# Tóm tắt đề tài

Ngày nay, với công nghệ ngày càng phát triển và cuộc cách mạng công nghiệp 4.0 người ta thường nhắc đến việc tích hợp trí tuệ nhân tạo (AI) với các thiết bị như điện thoại, tivi thông minh, tủ lạnh,... Vì vậy nhu cầu phát triển các service AI, triển khai, kết hợp các service với các ứng dụng khác ngày càng cao.

Tuy nhiên, đa phần các ứng dụng AI đều phát triển khá là độc lập, mang tính chuyên môn hóa cao, mỗi service chỉ chuyên biệt phục vụ về một chức năng riêng biệt như phần mềm xử lý nhận diện chữ viết, phần mềm xử lý kiểm tra chính tả của một đoạn văn,... Để kết nối các service này, ví dụ đưa một bức ảnh có ký tự bên trong, cần phân tích xem đoạn văn trong bức ảnh và kiểm tra chính tả của đoạn văn đó. Để làm điều này chúng em cần kết nối các service nhận dạng chữ viết và service kiểm tra chính tả lại với nhau, AI platform sẽ giải quyết các vấn đề này.

Nội dung đề tài của chúng em đưa ra một giải pháp gọi là AI platform mang tính nền tảng, cho phép các nhà phát triển tích hợp các ứng dụng AI với nhau một cách đơn giản và dễ dàng. Khi các service AI mới lên muốn tích hợp vào hệ thống đang được sử dụng bởi platform này, chỉ cần cung cấp các API cần thiết đã được định nghĩa sẵn thì sẽ tích hợp vào và deploy dễ dàng. Do đó AI platform của tụi em giúp lập trình viên focus vào việc lập trình AI, còn việc tích hợp với hệ thống thì để AI platform lo. Một ứng dụng AI thường có quy trình phát triển như sau: phát triển AI của ứng dụng, và những phần còn lại của ứng dụng như các service chứa logic của doanh nghiệp, giao diện,... Sau đó ráp các phần lại và đưa cho người dùng sử dụng. Tuy nhiên, người lập trình chỉ nên focus vào cái cốt lõi của ứng dụng, đó là phần ứng dụng AI, là phần mang lại giá trị lớn nhất cho ứng dụng. AI Platform cung cấp tính năng cho phép ráp các logic của doanh nghiệp và các service AI lại với nhau và giúp chúng chạy theo một flow từ đầu đến cuối theo một thứ tự mà bản thân người phát triển phần mềm định nghĩa.

Tại sao tụi em lại đặt tên là AI platform, mặc dù nền tảng này có thể triển khai cho nhiều loại ứng dụng khác nhau? Tại vì các lập trình viên thuần về AI thường tốn nhiều thời gian để nghiên cứu về thuật toán cũng như những ứng dụng mà nó mang lại. Còn khả năng deploy và đưa các ứng dụng AI cho người dùng thông thường sử dụng về mặt nào đó còn hạn chế. Tại vì mỗi ứng dụng AI đều xử lý tốn kha khá thời gian, nếu hệ thống xử lý không khéo thì trải nghiệm người dùng sẽ tệ đi. Do đó AI platform giảm bớt việc cho lập trình viên AI, lập trình viên AI không cần phải lo lắng quá nhiều về hệ thống 1 người dùng, 100 người dùng, 1 triệu người dùng nữa vì AI platform sẽ handle chuyện đó.

# Chương 1

## Mở đầu

*Chương 1 giới thiệu về kiến trúc Microservices cùng sự phát triển của AI và ứng dụng của AI trong đời sống. Chương này cũng nêu lên lý do, mục tiêu chúng em muốn đạt được khi làm đề tài.*

### 1.1 Giới thiệu chung

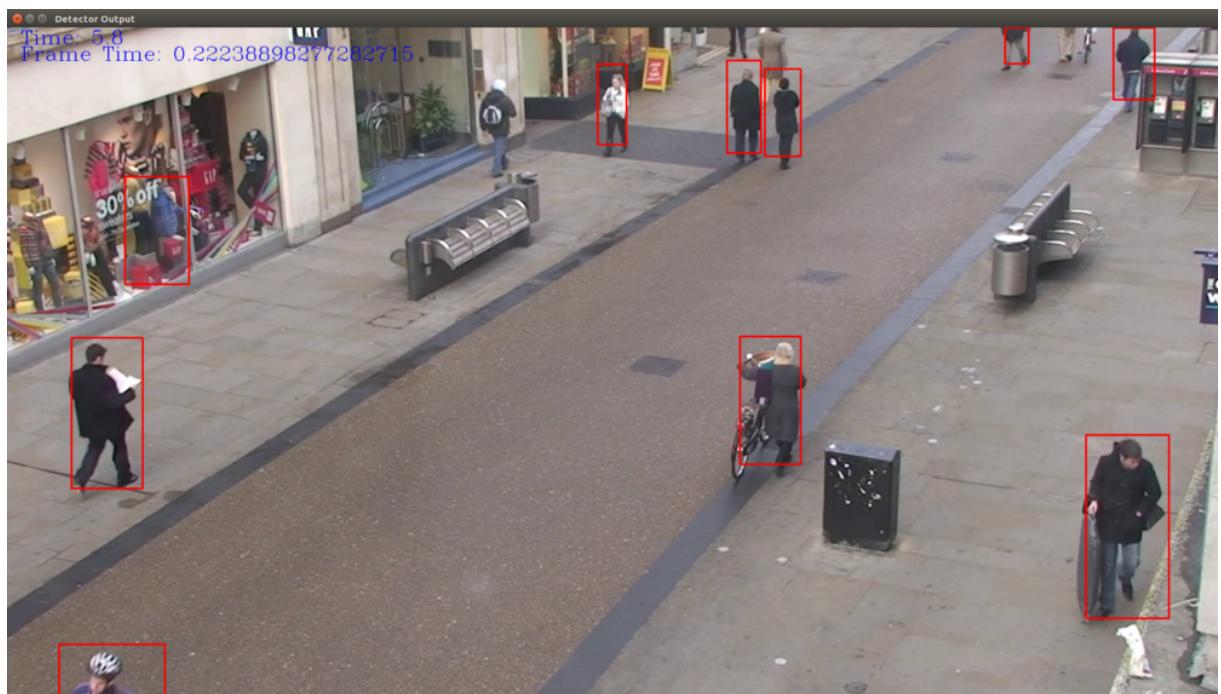
Microservices là một trong những từ khóa nóng nhất của những năm gần đây của ngành công nghệ phần mềm, và xuất hiện liên tục trong những hội nghị về công nghệ phần mềm. Nhưng microservices không chỉ là một trào lưu dễ bị thoái trào, mà nó là kết tinh cũng như là tiến hóa của những ý tưởng từ trước.

Cụm từ Micro web service xuất hiện đầu tiên trong một sự kiện về điện toán đám mây vào năm 2005. Cụm từ này được giới thiệu bởi Peter Rodgers, khi mà kiến trúc áp dụng SOAP SOA đang ở thời kì đỉnh cao, thì ông lại đề xuất kiến trúc áp dụng REST. Ông cho rằng tách một phần mềm lớn ra thành từng component, với mỗi component là một micro web service và kết nối với nhau như pipeline của Unix. Cụm từ Microservices xuất hiện tiếp theo tại một hội nghị về kiến trúc phần mềm vào năm 2011, nơi mà cụm từ này được diễn tả như là một phong cách thiết kế kiến trúc mới, lúc này công ty Netflix và công ty Amazon đang là tiên phong trong

việc áp dụng kiến trúc này.

Kể từ đó, kiến trúc microservices ngày càng được phổ biến rộng rãi bởi vì nó giải quyết được nhiều thách thức trong ngành phần mềm như việc tăng tốc xử lý, khả năng scale ứng dụng, và tăng tốc trong quá trình kiểm thử.

Trong khi đó, trí tuệ nhân tạo đã và đang phát triển một cách tốc độ nhất. Giờ đây ứng dụng của trí tuệ nhân tạo đã được phổ biến vô cùng rộng rãi trong đời sống hằng ngày. Ví dụ như việc nhận diện được khuôn mặt của con người thông qua ảnh và video, nhận diện chữ viết tay để có thể chuyển đổi dữ liệu dạng văn bản ghi chữ thành dữ liệu kĩ thuật số.



Hình 1.1: Ảnh chụp màn hình nhận diện con người của hệ thống AI trường Oxford.

Hình 1.1 là kết quả của hệ thống nhận diện con người trên camera công cộng thuộc dự án của trường đại học Oxford. Ứng dụng của việc này có thể kể đến là đếm số người đang tham gia trên đường từ đó biết được mật độ dân số, hoặc có thể đơn giản là chống trộm.



Hình 1.2: Nhận dạng chữ trong biển báo giao thông bằng phần mềm AI của Google.

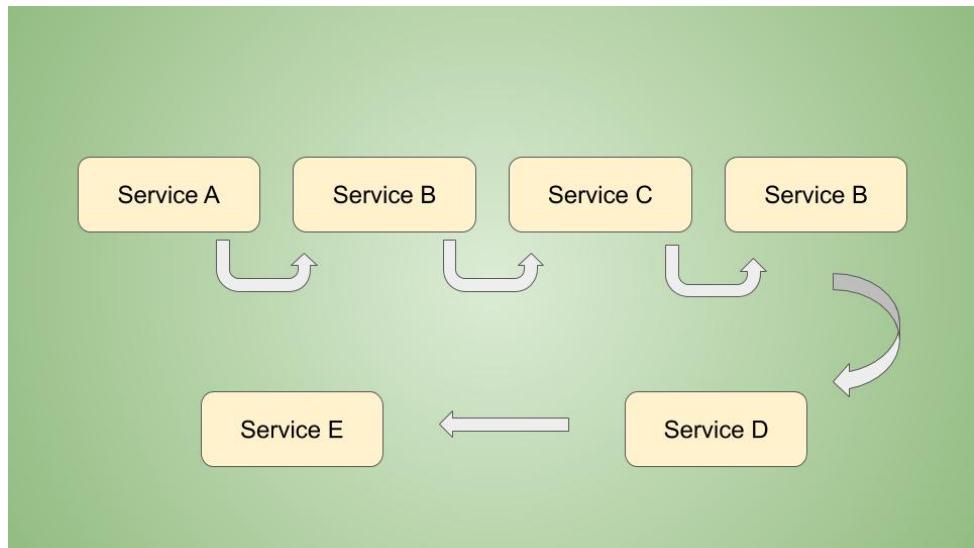
Hình 1.2 nhận dạng chữ viết từ ảnh, lấy ví dụ từ phần mềm nhận dạng chữ của Google. Ứng dụng của việc nhận dạng chữ viết có thể kể đến như khi tham gia giao thông, việc để hệ thống đọc và nhận dạng được các biển báo có gắn chữ và sau đó sẽ hỗ trợ người tham gia giao thông tốt hơn.

## 1.2 Lý do thực hiện đề tài

Với sự phát triển của công nghệ ngày nay thì việc phát triển ứng dụng mới đáp ứng nhu cầu người dùng ngày càng được quan tâm, làm sao cho tốc độ ra sản phẩm một cách nhanh nhất nhưng vẫn đảm bảo hiệu năng của hệ thống, giúp trải nghiệm người dùng một cách hiệu quả.

Các lập trình viên, thông thường chỉ biết đến cách thiết kế hệ thống, phân rã logic ứng dụng thành các module và triển khai ra các module đó thành các service. Họ thường gặp khó khăn trong việc deploy các service của mình lên các server hoặc là có một đội SO (System Operation) chuyên biệt thực hiện hiện.

Bên cạnh đó, các service trong hệ thống thông thường có một luồng đi nhất định, lập trình viên khi code phải cần tự quản lý các service do bản thân quản lý, thiết lập kết nối với server khác trong hệ thống.



Hình 1.3: Sơ đồ thực hiện theo tuần tự ở các service

Với trường hợp này ở service B, thấy rằng khi phát triển, bản thân lập trình viên service B cần phải tự xử lý logic thiết lập kết nối với service C và D. Sau này nếu có sự thay đổi về kết nối, Service B phải lại tự đi thiết lập kết nối với các service khác trong hệ thống, gây khó khăn và phức tạp cho logic của service B thay vì chỉ quan tâm đến logic của mình.

### 1.3 Mục tiêu đề tài

Với những khó khăn chúng em đã đề cập, các lập trình viên thường khó khăn trong việc deploy các service

Chúng em mong muốn AI platform trở thành một kiến trúc mẫu, dành riêng cho những lập trình viên tối ưu thời gian để tập trung vào những giá trị cốt lõi của phần mềm. Lập trình viên chỉ cần dành thời gian để tập trung vào logic của phần mềm, những việc như scale, xử lý downtime có thể để AI platform xử lý.

## 1.4 Nội dung đề tài

Nội dung đề tài gồm 5 chương, nội dung chính của từng chương như sau:

Chương 1: Trình bày tổng quan về sự phát triển của Microservice gắn liền với sự phát triển của các Service AI trong thời đại hiện nay. Tiếp theo chúng em nêu ra lý do thực hiện đề tài dựa trên các vấn đề đó, từ đó đưa ra mục tiêu cần hướng đến của đề tài.

Chương 2: Trình bày, phân tích các vấn đề hiện có mà kiến trúc của chúng em cần giải quyết và đưa ra các giải pháp phù hợp, tối ưu hiệu năng nhất cho hệ thống dựa vào các công nghệ phù hợp ở thời điểm hiện tại.

Chương 3: Mô tả chi tiết các thành phần của kiến trúc platform, chi tiết các services bên trong, phân tích các ưu điểm của kiến trúc và khả năng mở rộng, phát triển cho tương lai.

Chương 4: Trình bày kết quả đạt được và hướng phát triển trong tương lai.

Chương 5: Tổng kết lại quá trình và kết quả thực hiện đề tài của nhóm chúng em.

# Chương 2

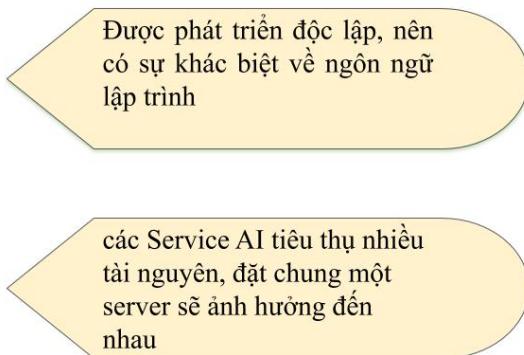
## Các vấn đề và giải pháp khi thiết kế

*Giới thiệu các vấn đề gặp phải và giải pháp cho từng vấn đề khi chúng em bắt đầu thiết kế cho kiến trúc. Bắt đầu từ việc chúng em nghiên cứu kiến trúc Microservices và các công nghệ liên quan, cho đến việc chúng em ứng dụng công nghệ giải quyết những vấn đề cơ bản.*

### 2.1 Các services trong kiến trúc

#### 2.1.1 Vấn đề

Như đã đề cập ở phần giới thiệu, chúng em thấy rằng hiện tại các service AI thường được phát triển độc lập, mang tính chuyên môn hóa cao. Mỗi service phục vụ riêng biệt về một chức năng và được phát triển bởi nhiều team khác nhau. Vì vậy sẽ dẫn đến việc các service AI thường được phát triển bởi một cá nhân hay đội nhóm riêng biệt. Việc gộp các Service AI mong muốn được tích hợp vào cùng một server có nhiều nhược điểm như hình 2.1



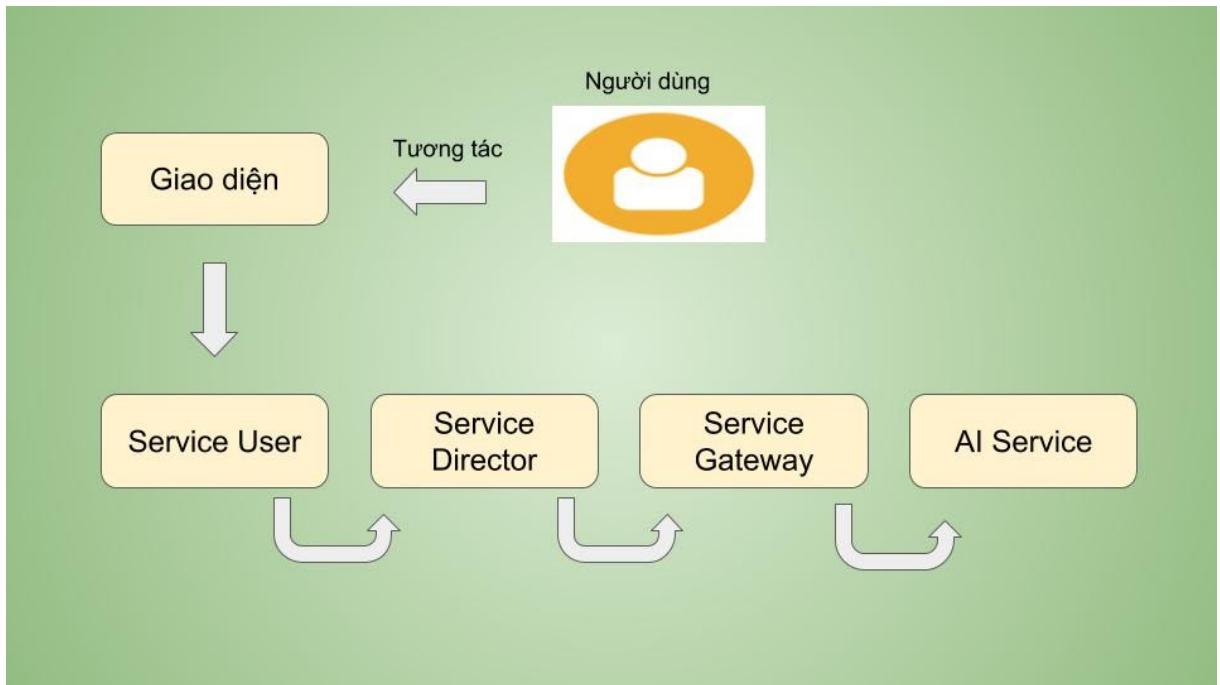
Hình 2.1: Nhược điểm khi phát triển phần mềm ứng dụng AI

Do các đặc tính như vậy, để tích hợp chúng lại với nhau, chúng em cần một kiến trúc các service tách rời với nhau độc lập, đảm bảo không thay đổi logic. Hiện tại với kiến trúc Microservice sẽ giúp chúng em làm điều này.

### 2.1.2 Kiến trúc Microservice là gì?

Microservice là một kiến trúc mà ở đó các module trong một tính năng lớn được chia thành nhiều tính năng nhỏ được phát triển độc lập và riêng biệt, kết nối với nhau thông qua các giao thức như HTTP, gRPC, message, ... Mỗi Module sẽ được đặt trên một server riêng.

Các service AI tương trưng cho các module, mỗi service AI sẽ được đặt trên các server riêng và phát triển độc lập. Bên cạnh đó còn có một số các service khác của hệ thống platform cũng được triển khai theo kiến trúc Microservice, ở mỗi service được sử dụng chuyên biệt về các logic khác nhau. Kiến trúc hiện tại của hệ thống platform thể hiện ở hình 2.2.

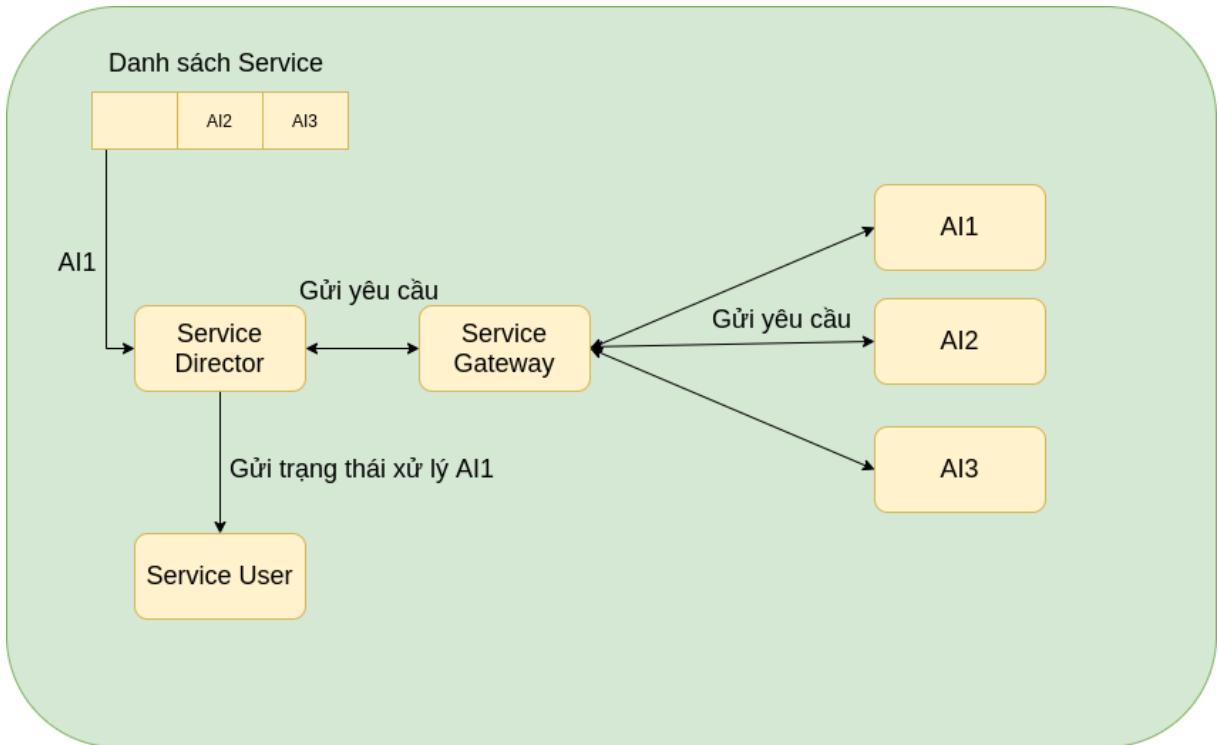


Hình 2.2: Kiến trúc của AI platform

**Giao diện:** Xây dựng giao diện đơn giản cho người dùng tương tác, tạo yêu cầu gửi đến cho hệ thống, và cho phép theo dõi lịch trình của yêu cầu đã đến đâu service nào trong platform.

**Service User:** Quản lý thông tin người dùng và yêu cầu được gửi từ giao diện (front end). Quản lý trạng thái của các yêu cầu đang đi xử lý ở service nào và cung cấp các API cho giao diện người dùng để lấy trạng thái các yêu cầu.

**Service Director:** Khi nhận yêu cầu từ client, với mỗi nhu cầu khác nhau thì sẽ định nghĩa sẵn danh sách các service AI được thực hiện theo một trình tự nhất định. Service Director có chức năng xử lý yêu cầu ở từng AI, lấy trạng thái xử lý, sau khi hoàn thành, chúng sẽ lấy thứ tự tiếp theo trong danh sách và tiếp tục xử lý cho đến khi nào hoàn thành. Ở mỗi lần xử lý thành công, sẽ bắn tin nhắn về cho Service User để cập nhật lại trạng thái hoạt động ở tiến trình đó.

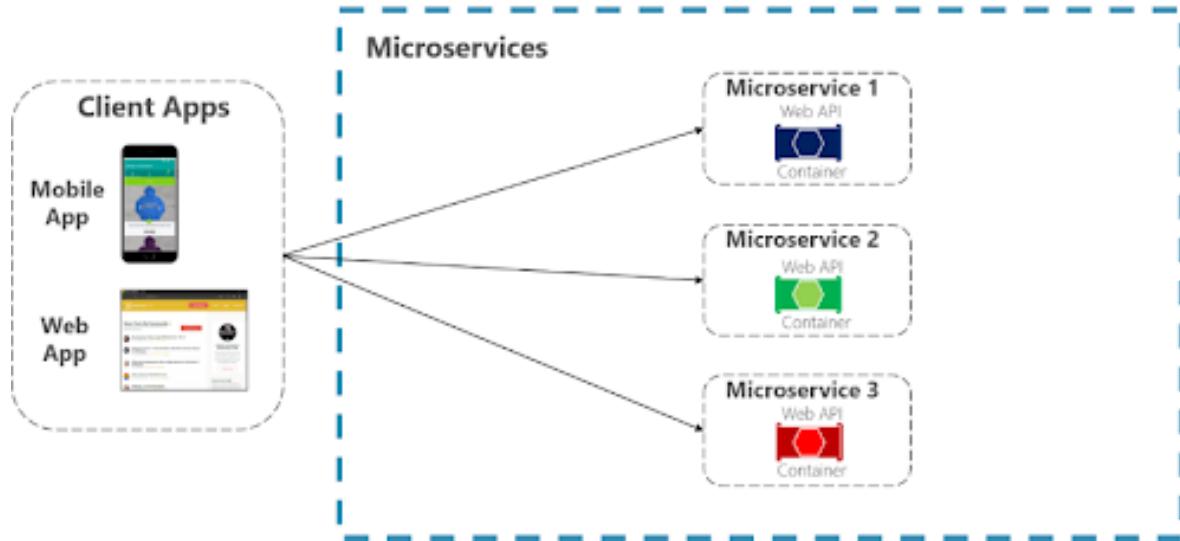


Hình 2.3: Chi tiết kiến trúc của Director

**Service Gateway:** có nhiệm vụ nhận các yêu cầu xử lý từ Director, kiểm tra yêu cầu đến service cụ thể nào nào như AI1, AI2,... và gửi đến service đó. Chúng em sử dụng mẫu thiết kế gateway để thay cho việc client phải gọi trực tiếp đến server (Direct client-to-microservice communication).

## Direct Client-To-Microservice communication

### Architecture



Hình 2.4: Kiến trúc gọi trực tiếp

Hình 2.4 mô tả hướng thiết kế gọi trực tiếp đến từng service từ phía client. Trong cách tiếp cận này, mỗi service trong microservice phải có địa chỉ IP hoặc URL và phải public ra bên ngoài để các app của client gọi được. Thường các service sẽ có chung public IP nhưng khác port, hoặc khác URL để phân biệt với nhau.

Ví dụ kiến trúc microservices bao gồm 3 services:

- Service A có IP là 10.30.83.2 với port là 20000.
- Service B có IP là 10.30.83.2 với port là 20001.
- Service C có IP là 10.30.83.2 với port là 20002.

Thông thường khi một kiến trúc chạy thực tế, mỗi service sẽ lấy một URL. Các URL sẽ giống phần đầu và khác phần đuôi. Xét trường hợp mới nay, bây giờ service A sẽ có URL public là `http://baseurl.com/service/a/`, còn service B sẽ có URL public là `http://baseurl.com/service/b/`.

Bản chất mỗi yêu cầu xử lý gửi từ client API gửi đến IP và port hoặc gửi cho URL của service, thì request sẽ được chuyển đến một load balancer trước tiên. Tại vì một service có thể được scale lên nhiều instance, cho nên load balancer sẽ chọn instance nào để xử lý yêu cầu nhận được này. Load balancer này không đề cập tới trong hình trên vì ngầm hiểu là mỗi service đều có một load balancer ngăn giữa yêu cầu và các instance bên trong.

Kiến trúc gọi trực tiếp từ client đến từng service đủ tốt cho những ứng dụng với số lượng service trong microservices ít và vừa. Đặc biệt là những ứng dụng web đơn giản. Khi số lượng service trong kiến trúc microservices tăng lên từ nhiều đến rất nhiều, kiến trúc gọi trực tiếp dần bộc lộ điểm yếu của nó.

Một trong những điểm yếu của kiến trúc gọi trực tiếp là những công việc cần xử lý chung trước khi gọi yêu cầu từ client app. Cơ bản như việc xác thực tài khoản. Xác thực tài khoản tại vì có những yêu cầu gọi đến service thông qua API là nội bộ, và giới hạn chỉ cho người có tài khoản truy cập chẳng hạn, nếu gọi API mà không gửi nội dung xác thực thì sẽ báo lỗi ngay lập tức. Tuy nhiên với kiến trúc gọi trực tiếp từng service, mỗi service đều phải cài đặt xác thực tài khoản người dùng, dẫn đến việc lặp code không đáng có. Chưa kể, mỗi service có thể có người lập trình viên khác nhau, dễ dẫn đến việc nhiều lời giải cho cùng một bài toán, gây lãng phí tài nguyên không đáng có.

Một điểm yếu khác là client app chỉ giao tiếp với server thông qua giao thức HTTP, trong khi các service giao tiếp với nhau rất đa dạng, có thể là HTTP, RPC, AMQP, ... Do đó nếu client muốn giao tiếp với service không sử dụng HTTP thì phải có một service adapter làm nhiệm vụ chuyển yêu cầu dưới dạng HTTP thành yêu cầu dạng giao thức (protocol) tương ứng. Cách tiếp cận này gọi là man in the middle, nghĩa là có người đàm ông đứng dưới giống như thông dịch viên giữa hai người nói khác thứ tiếng.

Trở lại với kiến trúc microservices, app phía bên client sẽ cần thực hiện một vài tính năng, mà những tính năng này có thể không ở cùng một service, mà sẽ chia ra mỗi service thực hiện một yêu cầu từ client app. Nếu

theo kiến trúc client app gọi trực tiếp từng service, thì client app phải lưu toàn bộ địa chỉ public của toàn bộ service thuộc kiến trúc microservices.

Bài toán đặt ra là, cùng với ứng dụng phát triển, service cũ sẽ bị xóa đi, service mới với tính năng mới được thêm vào, thêm cả service được update thay đổi địa chỉ chẳng hạn. Nếu số lượng service mà client app gọi lên chỉ dừng ở mức 3-4, thì mọi chuyện chưa có gì phức tạp cả. Nếu số lượng service tăng từ 3-4 lên 10 lên 20, thì việc phải quản lý 20 địa chỉ của 20 service này trên app thực sự là ác mộng.

Trong thực tế, app client hoặc là app android hoặc là app ios. Để mà đưa ứng dụng đến với người dùng, thì ứng dụng phải ở trên các kho ứng dụng có sẵn mà người dùng có thể truy cập, đối với hệ điều hành android là google play store, đối với hệ điều hành ios là app store. Mỗi lần ứng dụng muốn cập nhập phiên bản mới, thì phải gửi lên cho kho ứng dụng, sau đó chờ kho ứng dụng duyệt. Lý do của việc phải chờ duyệt này là vì, nhiều khi có những lập trình viên xấu tính, thích gài mã độc vào phần mềm để lừa lấy tiền người sử dụng. Cho nên kho ứng dụng sẽ quét xem thử phần mềm gửi lên có mã độc hay không, có vi phạm tiêu chuẩn của kho ứng dụng, nếu mọi thứ ổn thỏa thì mới cho phép ứng dụng được xuất hiện, cũng như lên phiên bản mới trên kho ứng dụng. Thời gian chờ duyệt có thể từ 1 đến 2 ngày.

Lưu ý thời gian chờ duyệt này là một khoảng thời gian tương đối, nếu trong thời gian 1 đến 2 ngày này, service thuộc kiến trúc microservices có thay đổi địa chỉ public, thì toàn bộ quá trình phải dừng lại, để client app cập nhập lại địa chỉ app, xong rồi mới đưa app lên kho ứng dụng trở lại. Trong thời buổi này, mọi thứ diễn ra trong chớp mắt, các đối thủ kinh doanh cạnh tranh nhau gay gắt, thì việc đưa ứng dụng lên kho ứng dụng càng nhanh càng tốt để người dùng tiếp cận là một yếu tố sống còn. Tiêu chí sai sót để còn sửa lỗi đang là kim chỉ nam trong công nghệ phần mềm hiện nay.

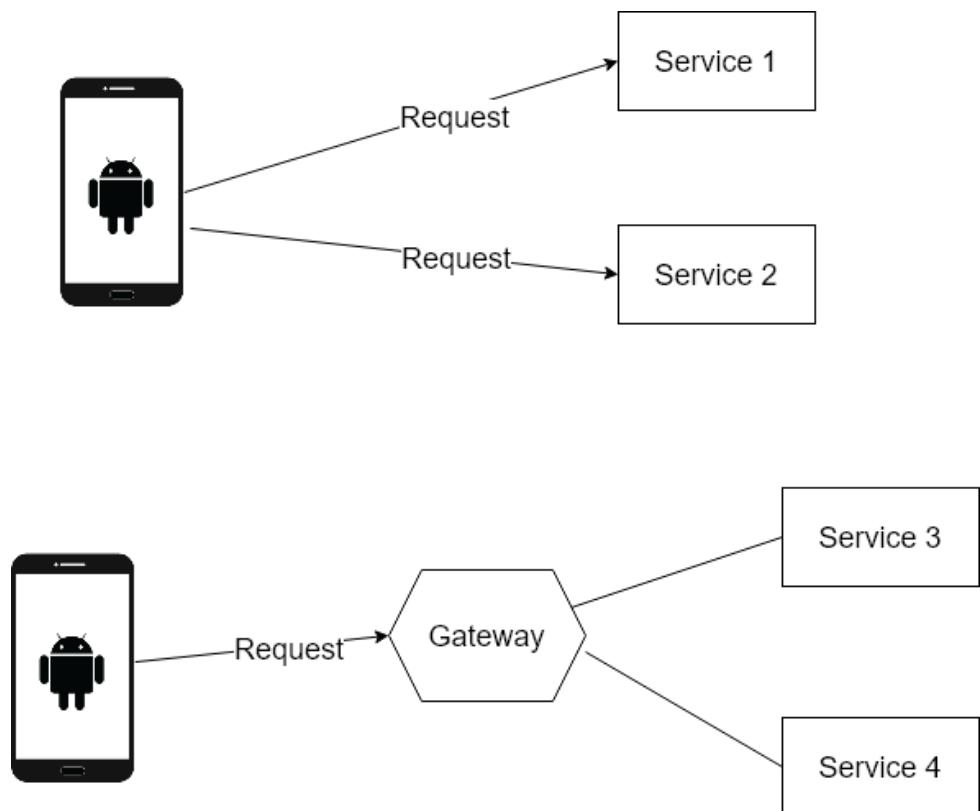
Do đó, có một giải pháp thay thế cho kiến trúc client gọi trực tiếp lên từng service trong microservices, đó là thiết kế API gateway. Gateway làm

đúng như cái tên của nó, là cái cổng đứng giữa ngăn client app và các service trong kiến trúc microservices.

Gateway thể hiện sức mạnh của của mình thông qua việc giải quyết các vấn đề mà kiến trúc gọi trực tiếp gấp phải.

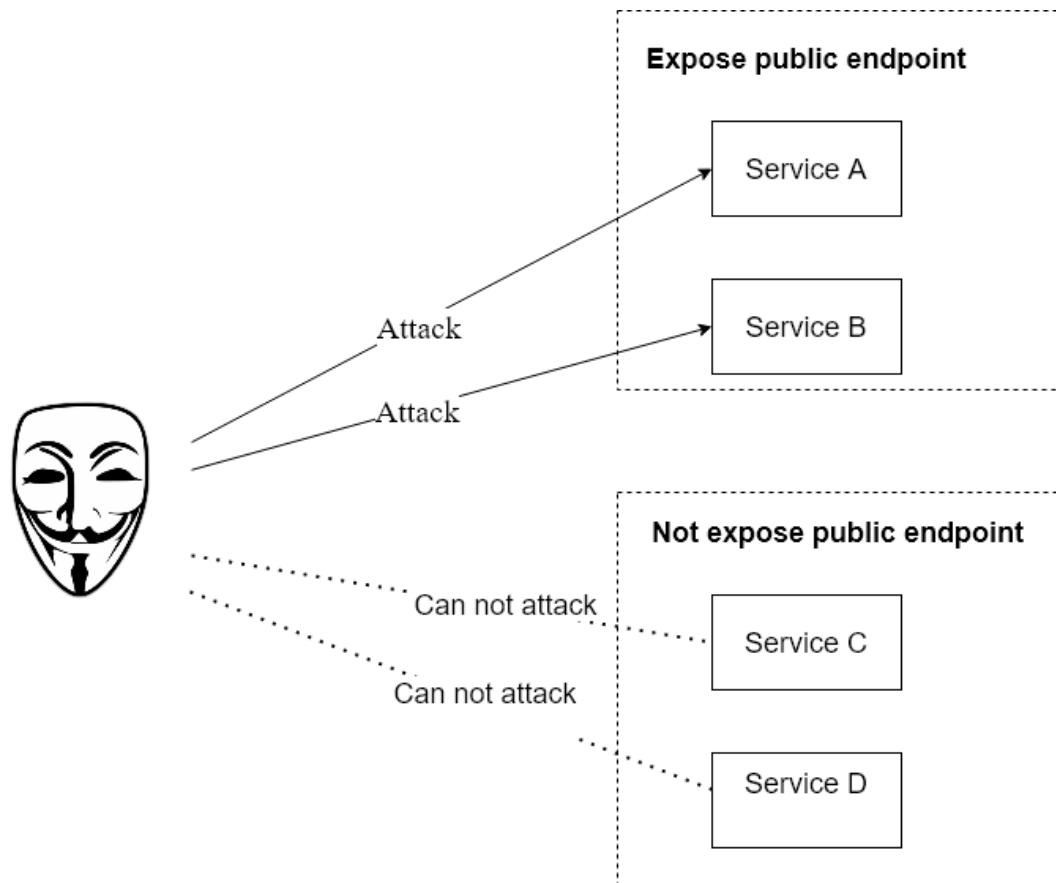
Vấn đề đầu tiên đó là tách sự phụ thuộc của client app vào service trong kiến trúc microservices. Lúc này client app chỉ cần biết đến gateway mà thôi. Giao tiếp với gateway và gateway sẽ giao tiếp với mọi service còn lại trong microservices.

Giả sử client app tạo 2 yêu cầu gọi lên service, yêu cầu thứ nhất lấy danh sách phim, yêu cầu thứ hai lấy địa chỉ rạp phim. Cả hai yêu cầu này nằm ở 2 service khác nhau, client app phải tạo 2 yêu cầu để gọi. Trong khi nếu có gateway, thì gom chung 2 yêu cầu này thành 1 yêu cầu, gateway sẽ tự biết để mà gọi cả 2 service kia, lấy đầy đủ thông tin sau đó trả về cho client app. Giảm từ gọi 2 yêu cầu xuống còn 1 yêu cầu, đơn giản thấy rõ như trong hình 2.5.



Hình 2.5: Client app gửi yêu cầu lên Gateway.

Gateway còn thể hiện tính ưu việt của mình ở chỗ, vì nó thể hiện vai trò là một cái cổng ngăn, cho nên toàn bộ các service của microservices không cần có địa chỉ public ra bên ngoài, mà có thể sử dụng địa chỉ bằng mạng nội bộ. Chỉ cần gateway mở địa chỉ public để client app gọi là được, và sau đó gateway gọi các service của microservice theo đường mạng nội bộ. Nếu không có gateway, mọi service đều phải có địa chỉ public, điều này mở ra rủi ro vô cùng khi đầy rẫy hacker có ý xấu luôn muốn tấn công vào những hệ thống vì mục đích bất chính. Có rất nhiều cách để tăng cường bảo mật, chống bị hack nhưng cách đơn giản nhất có lẽ là rút dây mạng. Ý tưởng đơn giản nhưng có thể hiện thực hóa bằng cách đặt toàn bộ hệ thống trong một mạng ảo của công ty, như thế thì hacker không thể phá phách được vì đơn giản, địa chỉ không public, không truy cập được được thì làm sao mà phá được.

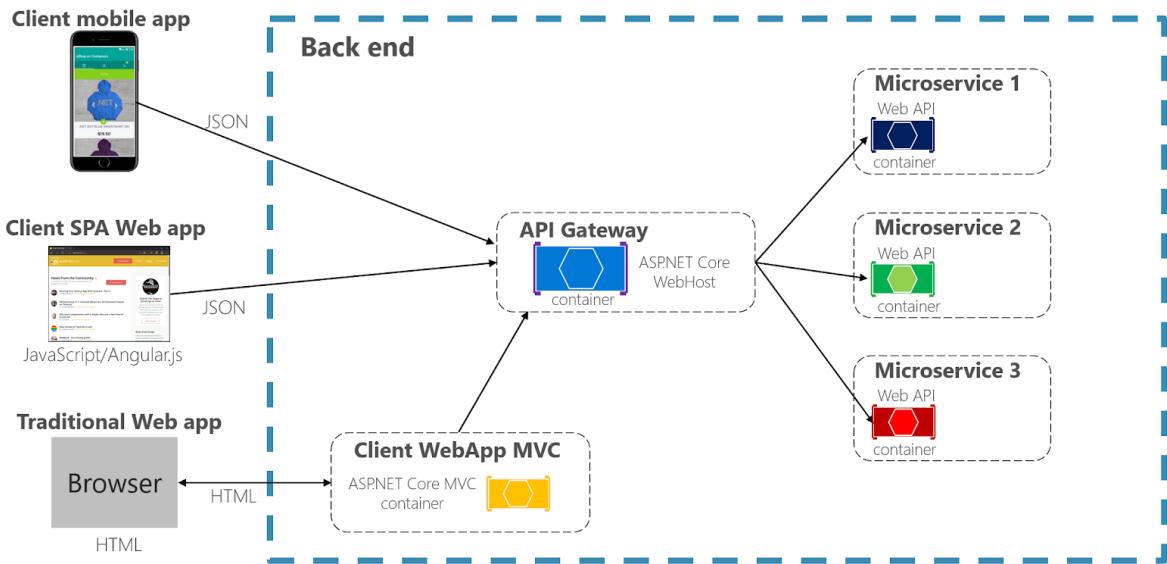


Hình 2.6: Minh họa hacker tấn công.

Bên cạnh đó, gateway còn được sử dụng như là một middleware cho mỗi yêu cầu của client app gửi đến cho các service thuộc microservice. Middleware là gì? Middleware là việc tiền xử lý các yêu cầu của client app trước khi yêu cầu đó được thực sự xử lý. Ví dụ như xác thực yêu cầu xem thử có đúng của người dùng được cấp quyền để gọi yêu cầu hay không, nếu xác thực thành công thì mới cho phép yêu cầu được gửi đến service cụ thể trong microservice để xử lý tiếp, nếu xác thực thất bại, từ chối yêu cầu ngay lập tức, vì có thể đây là tấn công có chủ đích đến từ hacker.

Cho nên mới nói, khi thiết kế và xây dựng một hệ thống microservices phức tạp, với nhiều client app như android app, ios app và cả web app, thì một hướng tiếp cận tốt là sử dụng thiết kế API Gateway. Nói một cách bài bản thì API gateway cung cấp một địa chỉ duy nhất cho toàn bộ hoặc một nhóm các microservices. Nó giống như mẫu thiết kế Facade của lập trình hướng đối tượng nhưng trong trường hợp này, API Gateway là mẫu thiết kế của hệ thống phân tán (distributed system). API Gateway còn được gọi là thiết kế backend của frontend vì các API cung cấp cho client app phải suy nghĩ sao cho việc sử dụng dễ dàng hết mức có thể, chứ không thiên về việc suy nghĩ giao tiếp giữa các service của backend với nhau. Ví dụ API cung cấp cho android app hoặc ios có thể có field ‘platform’, với giá trị có thể là ‘android’ hoặc ‘ios’, nhưng API cung cấp cho web app thì có thể không cần field này tại vì web app có thể chạy trên bất cứ đâu, thậm chí trên windows, mac os, linux. Đôi khi, API cung cấp cho từng client app có thể có định dạng khác nhau, có thể là JSON, có thể là HTML.

## Using a single custom API Gateway service

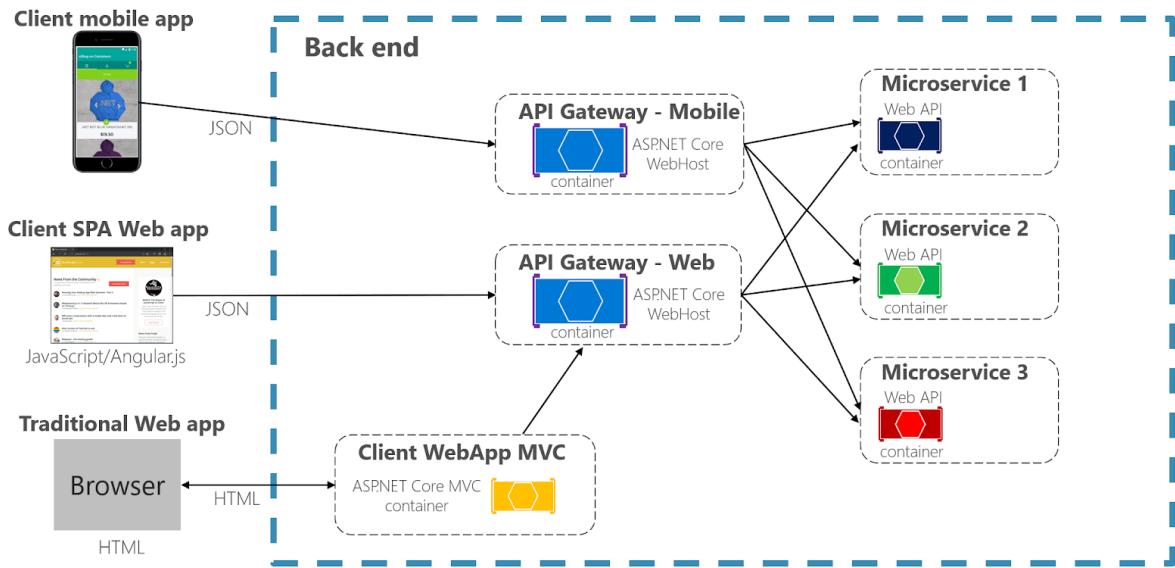


Hình 2.7: Kiến trúc API Gateway

Như hình 2.7, API Gateway đứng giữa làm nhiệm vụ nhận yêu cầu từ client mobile app và client web app để gửi yêu cầu đến cho các service thuộc kiến trúc microservice đứng đằng sau nó.

Tuy nhiên kiến trúc này cũng không hẳn tốt hoàn toàn, cùng với sự phát triển của phần mềm ngày càng nhanh. Càng nhiều client app muốn kết nối đến với hệ thống microservices, đó có thể là client app của cùng công ty với hệ thống, cũng có thể là một client app khác của bên thứ 3 muốn sử dụng hệ thống. Dần dần API Gateway trở nên bự hơn, nặng nề hơn vì để đáp ứng toàn bộ các nhu cầu của client app. Và nhu cầu của client app cũng chính là nhu cầu trực tiếp của người sử dụng. Do đó một hệ thống microservice lớn không nên chỉ có một API Gateway, mà có thể có nhiều hơn một, mỗi API Gateway sẽ ứng với một group các service trong microservice. Việc phân chia, gom nhóm thế nào cũng phải cho hợp lý. Thường thì gom nhóm tùy theo nhiệm vụ, như nhóm service làm chung một nhiệm vụ hoặc xử lý chung trên một đối tượng nào đó thì sẽ có một API Gateway.

## Using multiple API Gateways / BFF



Hình 2.8: Kiến trúc nhiều API Gateway

Hình 2.8 là một ví dụ cho việc có nhiều API Gateway, một API Gateway chuyên để xử lý client app là mobile app, ví dụ như android app hoặc ios app. Một API Gateway chuyên xử lý những yêu cầu đến từ web app.

Như ở trên đã nói một trong những điểm yếu của kiến trúc gọi trực tiếp đó là mỗi service đều phải cài đặt các nội dung như xác thực tài khoản, và sẽ đến chuyện trùng lặp tính năng, trùng lặp code gây dư thừa và lãng phí. Nếu có API Gateway, thì API Gateway sẽ xử lý những chuyện không liên quan, để cho những service trong microservice chỉ xử lý những yêu cầu chính, để không gây lãng phí tài nguyên.

API Gateway gánh vác giùm rất nhiều chuyện cho các service thuộc kiến trúc microservices. Như nãy chúng em có nói đến vấn đề xác thực tài khoản. Mỗi yêu cầu xử lý đến từ client app hay web app đều phải là yêu cầu được xác thực, được định danh. Nếu không xác thực định danh thì hacker có thể dựa vào đó để tấn công hệ thống một cách có chủ đích. Bên cạnh xác thực định danh, API Gateway còn có thể tỏ ra hữu ích bằng cách xử lý những việc như: tích hợp hệ thống tự phát hiện service (service discovery integration), sử dụng cache cho kết quả các yêu cầu, hạn chế số

lần gọi yêu cầu trong một giây, cơ chế tự động khắc phục nếu yêu cầu gấp trực tiếp, cơ chế chia tải (load balancing), blacklist hoặc whitelist cho yêu cầu, và có thể nhiều hơn nữa. Thủ tướng tượng nếu từng service trong kiến trúc microservice đều phải code và cài đặt toàn bộ những thứ vừa liệt kê, thì sẽ là một ác mộng.

### 2.1.3 Các tính chất cần lưu ý của các service khi sử dụng kiến trúc Microservice

**Resiliency:** Trong hệ thống có thể có nhiều hoặc rất nhiều service trong kiến trúc Microservice. Ở đó các service có thể dễ dàng bị lỗi với rất nhiều lý do như đường mạng, quá tải do số lượng request lớn, phần cứng, ... Khi giao tiếp với các Service bị lỗi tạm thời như vậy, cần có cơ chế thử lại để khắc phục, có hai mẫu thiết kế như nhau

**Retry:** Đường mạng thông thường có thể dễ dàng bị lỗi, thay vì ngừng giao dịch ngay lập tức, người gọi (server) nên thử lại hành động đó với số lần nhất định, hoặc đến khi request timeout. Ở đây cần lưu ý rằng có thể có trường hợp là người gọi có thể đã gọi thành công, nhưng phản hồi từ service được gọi lại bị đứt. Lúc này khi retry có thể khiến cho server thực thi hành động đó hai lần, đặc biệt là đối với các phương thức POST.

**Circuit Breaker:** Quá nhiều yêu cầu lỗi có thể dẫn đến "nghẽn cổ chai", thay vào đó có thể đưa các yêu cầu vào hàng đợi, các service xử lý sẽ lấy các message từ queue này và xử lý theo thứ tự được đưa vào, tránh việc trường hợp thất bại lặp lại liên tục chiếm tài nguyên của hệ thống như bộ nhớ, threads, database, ... Dẫn đến Server ngừng hoàn toàn, Circuit Breaker có thể ngăn chặn điều này. Các hàng đợi có thể chứa lượng message lớn và có các cơ chế để persistent xuống bộ nhớ dưới đĩa, đảm bảo các yêu cầu từ client không bị mất.

**Load Balancing:** Khi một yêu cầu gọi từ service A đến service B. Thông thường trong hệ thống Microservice, mỗi service thường có nhiều instance để đảm bảo khả năng chịu lỗi, do đó mỗi service cần có gateway để

cho các client gọi vào, ở Gateway có thuật toán phù hợp để chọn instance xử lý yêu cầu từ bên ngoài vào, có thể sử dụng các thuật toán phổ biến như Random, Round Robin, Hash, ...

**Distributed tracing:** một giao dịch có thể đi qua nhiều service trong kiến trúc, do đó khó có thể monitor chi tiết một giao dịch đã đi đến được đâu để có thể kiểm tra lại giao dịch với trường hợp bị lỗi mặc dù ở từng Service đã có hệ thống ghi log riêng biệt, việc kiểm tra từng service trong hệ thống sẽ mất rất nhiều thời gian. Do đó cần có hệ thống theo dõi tình trạng của một giao dịch đã đi đến đâu, đây là yêu cầu rất quan trọng trong các ứng dụng, giúp nhanh chóng phát hiện vấn đề ở Service nào để khắc phục sớm nhất.

**TLS encryption và TLS authentication:** để đảm bảo bảo mật cho các service trong hệ thống, cần phải mã hóa các request và response giữa các service với TLS.

#### 2.1.4 Ưu điểm và hạn chế của kiến trúc Microservice

Việc tổ chức thành các service độc lập như vậy có những ưu điểm nổi bật sau:

Mỗi service được chia nhỏ tập trung vào một chức năng cụ thể, giúp việc phát triển độc lập không cần phụ thuộc vào service khác, tránh trường với kiến trúc Monolith một service bị lỗi khiến cả hệ thống dừng hoạt động. Hơn nữa việc chia nhỏ chức năng ra thành các service giúp cho việc quản lý logic của từng dễ dàng hơn

Microservice, giúp quá trình deploy và phát triển một cách độc lập. Các lập trình viên không cần đợi các tính năng khác hoàn thành trước mới tiếp tục phát triển tính năng của mình, thay vào đó, họ lập trình song song với nhau và đến ngày hoàn thành rồi tích hợp, ở đây các Service AI, Director, Gateway,... được phát triển độc lập hoàn toàn với nhau, bởi nhóm lập trình khác nhau và ngôn ngữ khác nhau, nhưng vẫn có thể tích hợp vào hệ thống dễ dàng thông qua các giao thức protocol.

Mỗi service do chỉ phục vụ một tính năng cụ thể trong logic của toàn bộ hệ thống, giúp các thành viên khác của team dễ dàng hiểu và tham gia vào dự án sau này. Nâng cao được tốc độ thêm tính năng cho ứng dụng sau này.

Các service riêng biệt với nhau, do đó có thể sử dụng nhiều ngôn ngữ lập trình riêng biệt, dễ dàng thay đổi ngôn ngữ, dễ dàng thay đổi các công nghệ được sử dụng trong từng service, mà không ảnh hưởng đến toàn bộ hệ thống.

Bên cạnh các ưu điểm của mình, Microservice lại có những hạn chế nhất định . Vì các service giao tiếp với nhau thông qua đường mạng, dẫn tốc độ không cao bằng kiến trúc Monolith và dễ xảy ra lỗi hơn thông thường, cần có thêm cơ chế xử lý khi xảy ra lỗi. Bên cạnh đó, hệ thống phân tán gây khó khăn trong việc đảm bảo tính đồng nhất của dữ liệu, các vấn đề liên quan đến Distributed Transaction. Khi có quá nhiều service, sẽ gây khó khăn cho công tác quản lý, vận hành và tốn nhiều tài nguyên hơn khi cần deploy các service đó trên các server khác nhau.

## 2.2 Các phương thức giao tiếp giữa các trong kiến trúc Microservice

Với kiến trúc Microservice, chúng em đã chia nhỏ logic của platform thành các server riêng biệt, tiếp theo cần phải xác định giao thức liên lạc giữa các server với nhau, đảm bảo khi các server liên lạc với nhau thông qua đường mạng cho đến khi hoàn thành một transaction mà vẫn đảm bảo hiệu năng của hệ thống được tối ưu hóa tốt nhất. Sau đây chúng em xin trình bày về các phương thức giao tiếp giữa các service, các vấn đề về hiệu năng và cách tối ưu hóa.

## 2.2.1 Cơ chế giao tiếp giữa các service

Hiện tại có hai kiểu giao tiếp cơ bản mà các cụm service có thể áp dụng để giao tiếp với nhau:

**Synchronous Communication:** Ở pattern này các client gọi API được cung cấp bởi các server khác bằng cách sử dụng protocol như HTTP hoặc gRPC. Lựa chọn này được gọi là synchronous vì thread xử lý ở client phải đợi service thực thi và trả về kết quả sau đó mới thực hiện tiếp.

**Asynchronous Message Passing:** Ở pattern này Service gửi message đến service mà không cần đợi đến lúc thực thi xong, các message này được nhận ở service và đưa vào hàng đợi chờ xử lý. Sau khi có kết quả thì có hai lựa chọn để client có thể lấy được kết quả. Một là client tự chủ động gọi server để lấy thông tin process đang quản lý, hoặc sau khi xử lý xong thì server sẽ bắn một tín hiệu để báo cho client đã xử lý xong để client cập nhật lại trạng thái của yêu cầu đã gửi đến server.

Hai kiểu giao tiếp này hiện nay được sử dụng phổ biến, tùy vào nhu cầu của service và logic để có thể chọn cách giao tiếp phù hợp. Trong kiến trúc của AI Platform kết hợp cả hai kiểu giao tiếp này, mỗi loại đều có ưu và nhược điểm nhất định.

Đối với Synchronous:

- Ưu điểm:** Đặc điểm dễ nhận diện nhất là tiện dụng và quen thuộc với lập trình viên, xử lý theo nguyên tắc Synchronous được áp dụng phổ biến, không cần nhiều kinh nghiệm để xử lý, thiết kế hệ thống dễ dàng hơn. Với Synchronous, sau khi kết thúc tiến trình thì sẽ biết kết quả ngay lập tức, khi giao tiếp với server nếu kết quả thì dễ dàng xử lý các bước tiếp theo. Với các trường hợp cần xử lý real-time, có kết quả mới tiếp tục xử lý được thì Synchronous là lựa chọn tốt hơn, bởi vì nội tại Synchronous đã có ràng buộc là hoàn thành bước này thì mới thực hiện bước tiếp theo.
- Nhược điểm:** Chương trình chạy theo thứ tự đồng bộ nên sẽ sinh ra trạng thái chờ và là không cần thiết trong một số trường hợp, lúc

này bộ nhớ sẽ dễ bị tràn vì phải lưu trữ các trạng thái trước đó. Nếu thời gian chờ quá lâu mới thực hiện được tiến trình tiếp theo sẽ làm giảm trải nghiệm người dùng. Hơn thế là vấn đề Blocking I/O, thread đang được sử dụng ở luồng hiện tại phải đứng chờ tín hiệu từ server làm giảm hiệu suất của hệ thống.

Đối với Asynchronous: nghĩa là ở Client, Thread của luồng hiện tại không cần đợi tiến trình được thực thi hoàn tất, thay vào đó được sử dụng bởi các tiến trình khác cho các yêu cầu khác, điều này ảnh hưởng rất lớn, giúp tăng performance của hệ thống. ngoài ra Asynchronous còn có một số điểm nổi bật trong kiến trúc Microservice như:

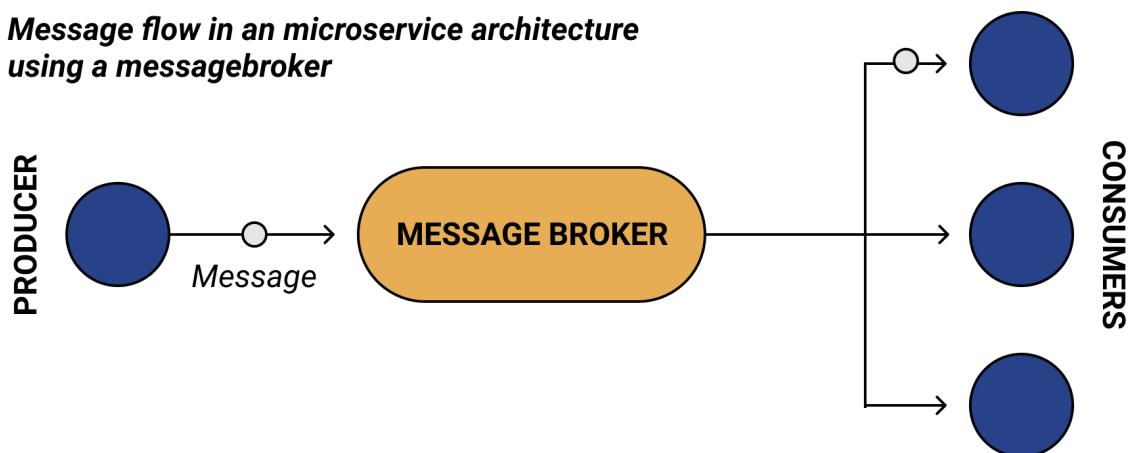
- **Reduced coupling:** các yêu cầu được gửi không cần biết cụ thể thông tin kết quả cuối cùng.
- **Multiple subscribers:** có thể sử dụng mô hình pub/sub, nhiều consumer có thể cùng đăng ký vào một topic, một request có thể được xử lý bởi nhiều service khác nhau.
- **Failure isolation:** nếu consumer bị lỗi thì ở các client vẫn có thể gửi các yêu cầu đến cho service. Các yêu cầu nằm ở hàng đợi và khi consumer được phục hồi, nó sẽ lấy các yêu cầu tồn động nằm trong hàng đợi. Đây là một tính năng rất hữu ích trong kiến trúc Microservice. Vẫn có thể xử lý yêu cầu trong thời gian vô hiệu ngắn hạn. Ngược lại Synchronous APIs, tất cả các yêu cầu đều sẽ bị lỗi.
- **Responsiveness:** Các yêu cầu đến các service sẽ có thời gian phản hồi ngắn hơn rất nhiều vì không cần đợi đến khi tiến trình được hoàn thành hoàn toàn, nó sẽ được xử lý ở background. Điều này rất hữu dụng cho kiến trúc Microservice, làm cho các service không phụ thuộc vào nhau. Ví dụ giả sử có tiến trình A gọi B, B gọi C, giả sử tất cả đều là Synchronous sẽ khiến tiến trình ở Service A bị timeout hoặc thread đợi rất lâu.

Như vậy việc giao tiếp giữa các service trong kiến trúc Microservice có nhiều lựa chọn. Trong đề tài này, do hệ thống xử lý của các service AI cần nhiều thời gian để xử lý, không thể để các service Client gửi yêu cầu và chờ đợi thời gian lên đến chục giây, xem xét với các đặc điểm của Asynchronous Message và Synchronous API thì chúng em sẽ chọn kiến trúc Asynchronous Message giao tiếp giữa các service để tăng hiệu năng của hệ thống. Bên cạnh đó trong project này, chúng em có sử dụng giao diện tương tác trên web, việc giao tiếp giữa front end và backend đặc thù nên sẽ chọn cơ chế synchronous API.

### 2.2.2 Cách thực hiện cơ chế Asynchronous

Có nhiều cách để thực hiện cơ chế Asynchronous trong kiến trúc Microservice, trong đó có hai cách phổ biến như sau

Cách thứ I: Giữa server và client giao tiếp với nhau thông qua Message Broker, client gửi các yêu cầu vào hàng đợi, ở server sẽ nhận các yêu cầu bằng cách lấy ra lần lượt các yêu cầu từ hàng đợi và xử lý. Ở trường hợp này, Client không biết trạng thái xử lý của các yêu cầu mình đã gửi đi, thường được sử dụng cho các luồng phụ của hệ thống ghi log.

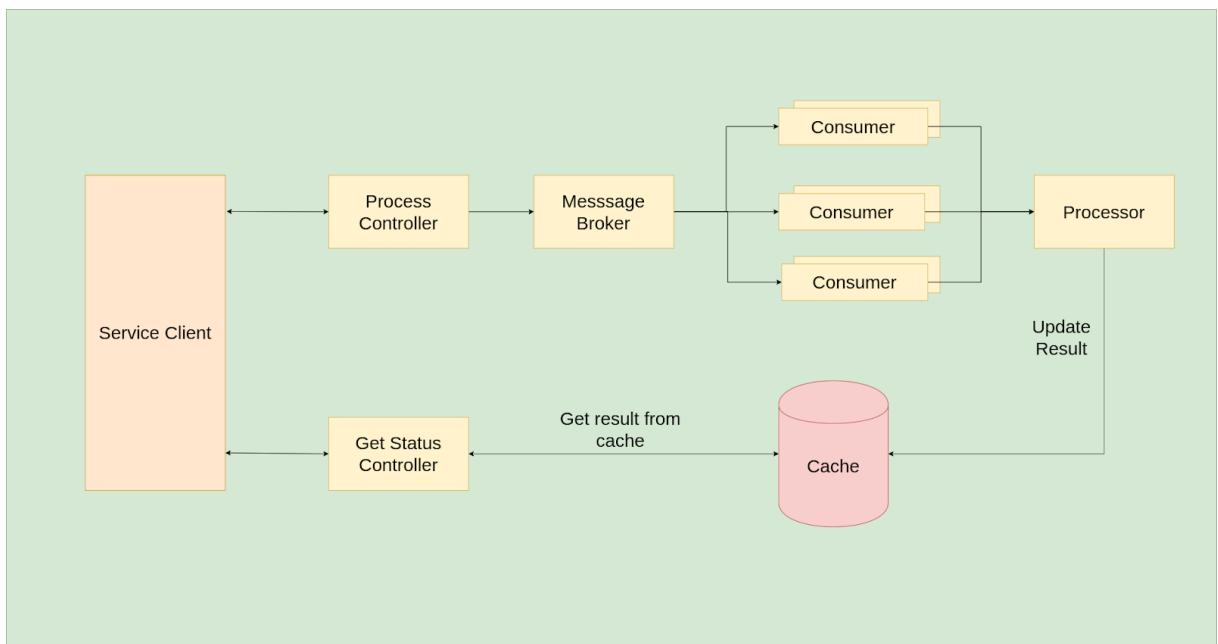


Hình 2.9: Mô tả message broker

Cách thứ II: Kết hợp việc sử dụng protocol như HTTP hoặc Grpc, Cache và Message Broker. Cụ thể là ở Server sẽ cung cấp hai phương thức cho client.

**Process:** client gửi yêu cầu gửi lý vào API process, trong khi đó ở server sẽ nhận yêu cầu, cập nhật trạng thái đang xử lý lưu vào bộ nhớ đệm và trả về trạng thái là đang xử lý cùng với một ID, client sẽ dùng ID để truy vấn trạng thái của yêu cầu. Đồng thời gửi yêu cầu vừa nhận được vào hàng đợi để chờ được xử lý. Ở nội tại của server sẽ có các consumer lấy yêu cầu theo thứ tự được đẩy vào và xử lý, sau khi có kết quả sẽ lưu lại trong bộ nhớ đệm (Cache).

**GetStatus:** ở Client sẽ gọi nhưng thức này cùng với ID được cung cấp ở phương thức trước để lấy trạng thái của quá trình xử lý, lấy kết quả từ Cache ra trả về cho client. Nếu là đang xử lý, client sẽ gọi lại sau một khoảng thời gian. Nếu vượt quá số lần mà vẫn không thành công coi như trạng thái xử lý yêu cầu đó là thất bại. Ở phía Service sẽ lấy kết quả từ bộ nhớ đệm của yêu cầu đó và trả về cho client.



Hình 2.10: kiến trúc Asynchronous cho service

Với cách thứ hai, ở Client có thể biết được trạng thái xử lý yêu cầu

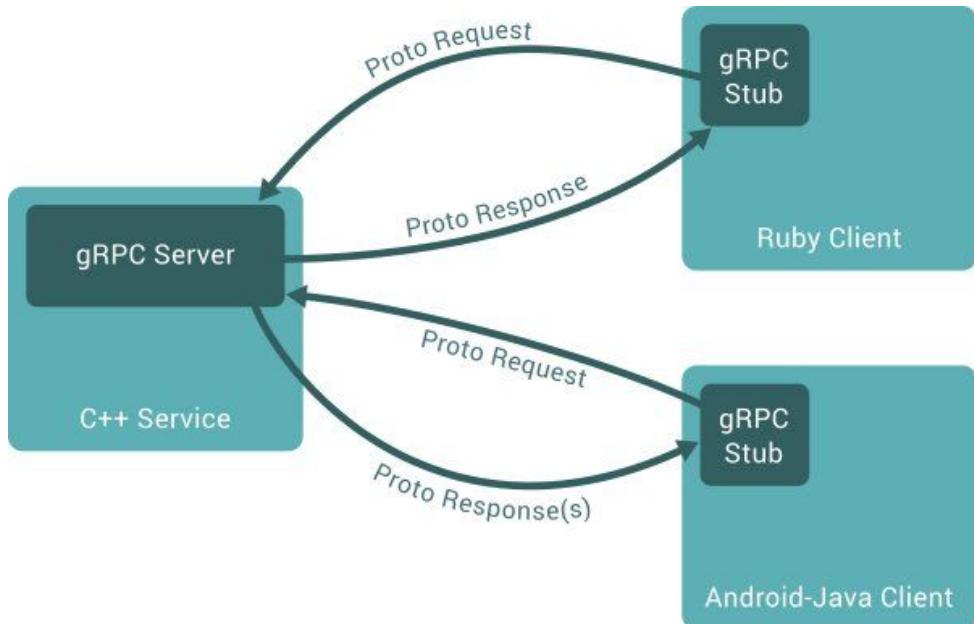
của bản thân, dựa vào đó để tiếp tục tiến trình của mình mà vẫn đảm bảo hiệu suất cao do thread vẫn không cần chờ đến khi tiến trình của server hoàn thành, tránh việc Blocking thread quá lâu ở client side (các service AI thường có thời gian xử lý lên đến vài chục giây), giúp tăng hiệu năng của hệ thống. Điều này phù hợp với đồ án hiện tại đang làm của chúng em. Trong hệ thống cần đảm bảo hiệu suất của các service tốt, vừa cần phải biết trạng thái xử lý của server mà client gọi đến, dựa vào kết quả để thực thi tiếp tiến trình của mình.

### 2.2.3 Sử dụng gRPC để giao tiếp giữa các service

Ở thời điểm hiện tại JSON REST API là protocol được sử dụng phổ biến để giao tiếp giữa client và server hoặc giữa các service trong kiến trúc Microservice vì tính đơn giản, gần gũi và cộng đồng rộng lớn. Nhưng gần đây thì có công nghệ mới được phát triển dành riêng cho việc giao tiếp giữa server và server là gRPC.

Trước tiên cần phải biết về RPC là gì? Đó là phương pháp gọi hàm từ một máy tính ở nơi khác để lấy về kết quả, xem như lời gọi hàm trong kiến trúc monolithic. Trong lịch sử phát triển mạnh mẽ của công nghệ ngày nay thì RPC là một cơ sở hạ tầng không thể thiếu trong việc liên lạc giữa các service giống như IPC (Inter Process Communication).

gRPC là một framework RPC Open source đa ngôn ngữ được phát triển bởi Google dựa trên nền tảng Protobuf và giao thức HTTP/2, giúp kết nối giữa các service trong hệ thống cùng với các tiện ích như load balancing, tracing, health checking và authentication, thường dùng giữa các server trong hệ thống. Sau đây là một số so sánh sự ưu việt khi sử dụng Protobuf và HTTP/2 khi giao tiếp giữa các service.



Hình 2.11: Hình ảnh minh họa việc giao tiếp của các service sử dụng gRPC

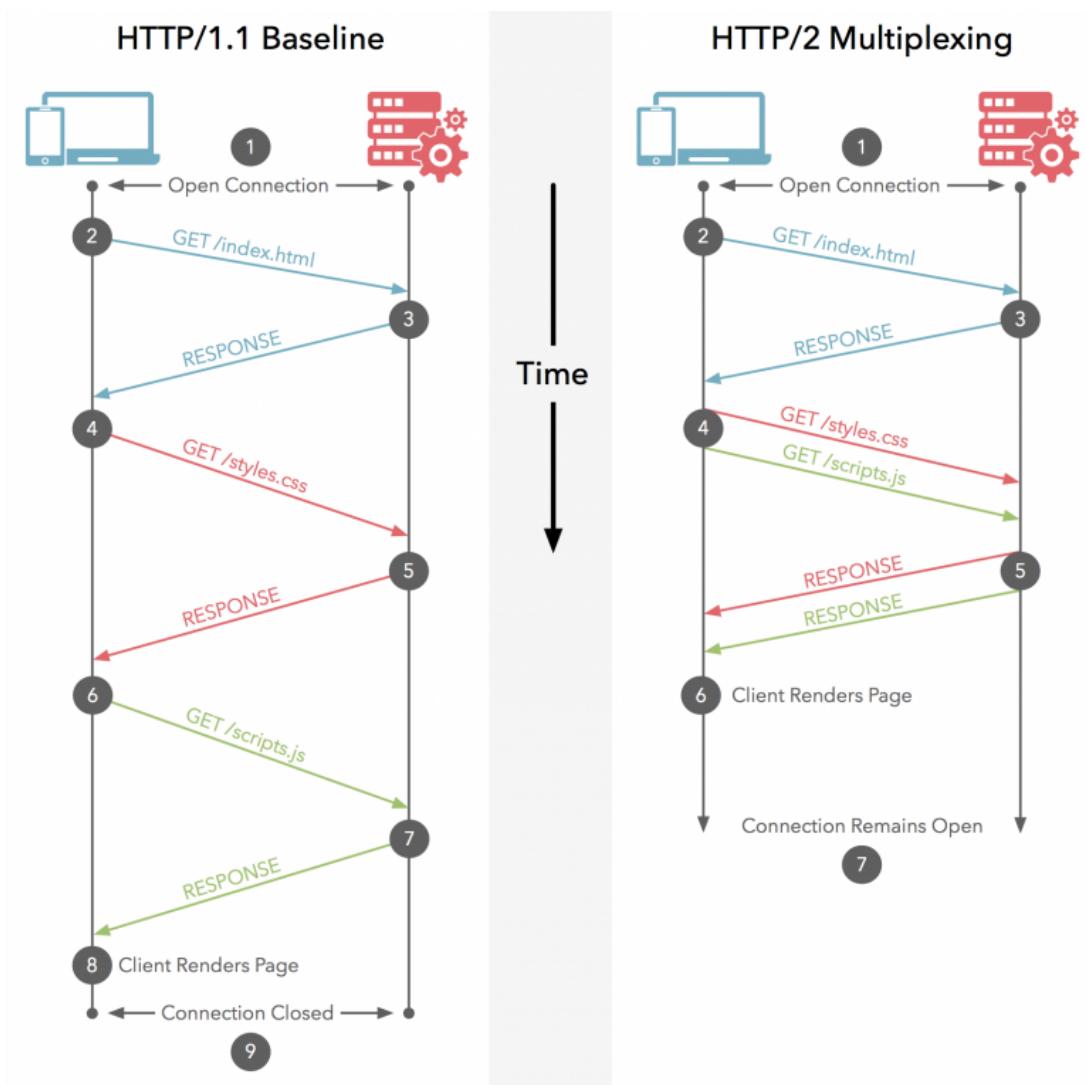
Protocol buffer được phát triển bởi google, chúng nhẹ hơn, hiệu quả hơn, dễ dàng tùy biến và có cơ chế tự động serializing dữ liệu giống như XML một cách dễ dàng, nhanh và có độ lớn nhỏ hơn. Cho phép định nghĩa cấu trúc của data dưới dạng file protoc và nó tự động tạo ra file sử dụng để giao tiếp với ngôn ngữ muốn sử dụng. gRPC hiện tại cũng đã hỗ trợ khá đầy đủ các ngôn ngữ như C++, Java, Python, Go.

HTTP/2 là phiên bản chính thức thứ hai của giao thức truyền tải siêu văn bản HTTP, dần được thay thế chuẩn HTTP/1.1 trong những thập kỷ gần đây với những ưu điểm nổi bật của mình, nổi bật như

- HTTP/2 sử dụng dữ liệu nhị phân trong khi HTTP/1.1 dùng dữ liệu dạng text, hiệu năng được cải thiện hơn so với phiên bản cũ.
- Sử dụng các Headers với dữ liệu đã được nén nhỏ đi trước khi được gửi. Khiến cho thông tin của một request gửi lên server giảm độ lớn, tăng hiệu năng.
- Sử dụng cơ chế bất đồng bộ, thay vì server phải gửi nhận theo một trật tự nhất định thì trong giao thức HTTP/2 cho phép các gói tin

nhỏ hơn được xử lý nhanh và trả về sớm hơn thay vì đi theo một trật tự. Hơn nữa sau khi đó thì HTTP/2 vẫn giữ connection giữa server và client để sử dụng cho các request kế tiếp thay vì đóng connection.

- HTTP/2 cho phép xử lý nhiều truy vấn dữ liệu giữa các service trên cùng một kết nối TCP duy nhất và giữ kết nối liên tục thay vì mỗi câu truy vấn là một kết nối TCP, giúp tiết kiệm tài nguyên của hệ thống.



Hình 2.12: Hình ảnh so sánh cơ chế hoạt động tối ưu hơn HTTP/2 và HTTP/1.1

Với hàng loạt ưu điểm như thế thì việc sử dụng HTTP/2 thực sự nhanh hơn HTTP/1.1 rất nhiều lần. Vì vậy ngày nay HTTP/2 ngày càng được sử dụng rộng rãi và phổ biến hơn, thay thế dần cho HTTP/1.1 điển hình là các ông lớn như Google, Facebook, Youtube,... đã sử dụng giao thức này.

Trước đó, chúng em đã nói về các ưu điểm mà gRPC hiện nay đang sử dụng so với các công nghệ sử dụng của RESTFUL API. sau đây là bảng so sánh tổng thể giữa gRPC và HTTP API with JSON.

Feature	gRPC	HTTP APIs with JSON
Contract	Required (.proto)	Optional (OpenAPI)
Protocol	HTTP/2	HTTP
Payload	Protobuf (small, binary)	JSON (large, human readable)
Prescriptiveness	Strict specification	Loose. Any HTTP is valid
Streaming	Client, server, bi-directional	Client, server
Browser support	No (requires grpc-web)	Yes
Security	Transport (TLS)	Transport (TLS)
Client code-generation	Yes	OpenAPI + third-party tooling

Bảng 2.1: Bảng so sánh giữa gRPC và HTTP API

Có thể thấy rằng việc giao tiếp giữa các service trong kiến trúc Microservice dùng gRPC sẽ tạo ra hiệu quả và hiệu suất cao hơn so với HTTP APIs JSON. Vì vậy trong hệ thống của chúng em sẽ sử dụng gRPC làm protocol chính để giao tiếp giữa các service.

Để sử dụng gRPC, cần phải định nghĩa gRPC service, tên phương thức, kiểu dữ liệu request và response bằng cách sử dụng protocol buffer. Đối với ngôn ngữ java, cần định nghĩa package mà các class được tạo ra khi biên dịch chương trình bằng option `java_package`.

```
1 option java_package = "io.grpc.examples.routeguide";
```

Sau đó tiếp tục định nghĩa Service và loại request response:

```
1 service RouteGuide {  
2     rpc GetFeature(Point) returns (Feature);  
3 }  
4  
5 message Point {  
6     int32 latitude = 1;  
7     int32 longitude = 2;  
8 }  
9  
10 message Feature {  
11     int32 latitude = 1;  
12     int32 longitude = 2;  
13 }
```

Ở đây ta định nghĩa một service là RouteGuide có phương thức là GetFeature. Với kiểu dữ liệu ở đầu vào là Feature, kiểu dữ liệu trả về là Point. Kiểu Point Và Feature có 2 thuộc tính kiểu integer là latitude và longitude.

Sau đó bằng cách sử dụng Gradle hoặc Maven của java và biên dịch chương trình sẽ có các class được tạo ra

Feature.java, Point.java các kiểu dữ liệu (message) được định nghĩa sẽ tạo thành các class có các thuộc tính tương ứng

RouteGuideGrpc.java chứa base class như RouteGuideGrpc.RouteGuideImplBase với các hàm đã được định nghĩa trong service, chúng giống như là một interface và cần được override lại các phương thức đó với logic riêng của từng hàm. Ở đây là Phương thức getFeature, các logic xử lý của hệ thống được thực thi tại đây, sau đó với onCompleted() sẽ kết thúc hàm xử lý và trả về phản hồi cho caller

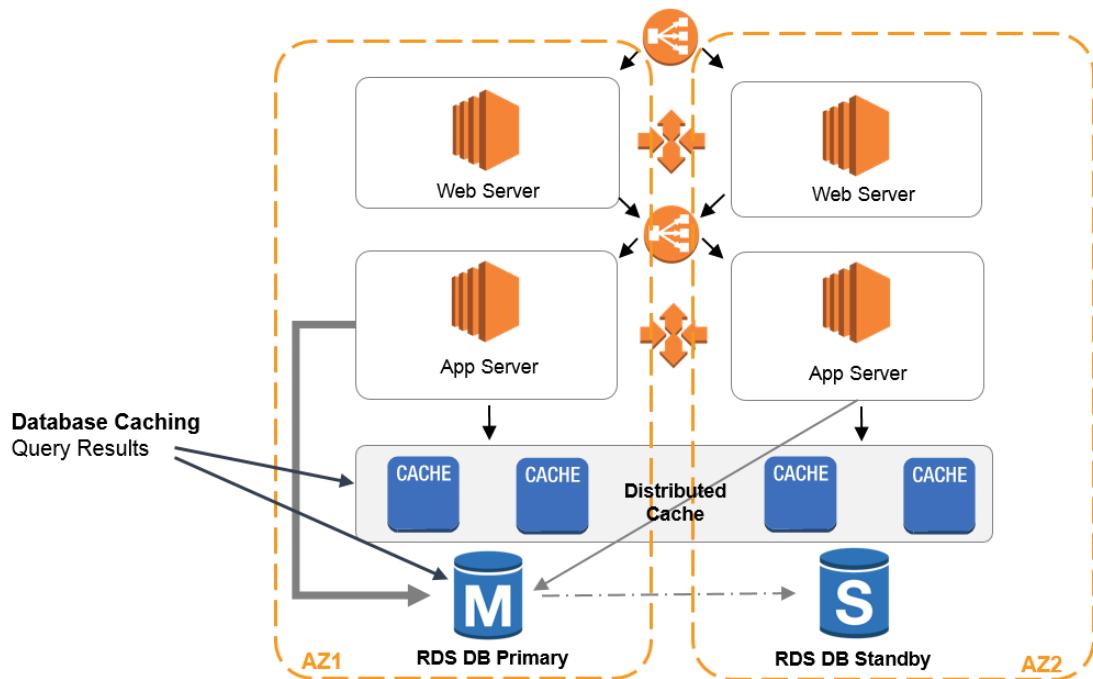
```
1 @Override  
2 public void getFeature(Point request, StreamObserver<Feature>  
3                         responseObserver) {  
4     responseObserver.onNext(checkFeature(request));  
5     responseObserver.onCompleted();  
6 }
```

## 2.3 Bộ nhớ đệm Cache

### 2.3.1 Vấn đề

Với các ứng dụng và kiến trúc ngày nay, thì hiệu năng của hệ thống luôn là vấn đề được quan tâm, ảnh hưởng đến thành công của các doanh nghiệp, tốc độ xử lý cao giúp tăng trải nghiệm của người dùng. Vì thế trong AI Platform, chúng em luôn chú ý nâng cao hiệu năng, làm cho tốc độ thống xử lý nhanh hơn. Để làm được điều đó thì chúng em sử dụng Caching. Trong các service, chúng em sử dụng bộ nhớ đệm để lưu lại các trạng thái xử lý của từng giao dịch được sinh ra là đang xử lý, thất bại hoặc đã xử lý thành công. Việc sử dụng Cache giúp tăng hiệu năng và tốc độ khi đọc và ghi các trạng thái của một giao dịch.

Caching là quá trình lưu lại một số dữ liệu trong tạm thời trong Cache. Cache là một nơi để lưu trữ dữ liệu tạm thời, nơi mà có tốc độ truy xuất nhanh hơn nhiều khi lấy ra từ đĩa hoặc database.



Hình 2.13: Hình ảnh mô tả việc sử dụng bộ nhớ đệm trong hệ thống

## 2.3.2 Redis là gì?

Trong đề tài này chúng em sử dụng cache với công nghệ hiện tại là Redis. là một mã nguồn mở dùng để lưu các dữ liệu có cấu trúc, có thể sử dụng cho việc lưu trữ database, cache hoặc message broker, lưu trữ dưới dạng key-value và được sử dụng rộng rãi nhất hiện nay. Redis hỗ trợ nhiều cấu trúc dữ liệu khác nhau như: hash, string, set,... Tất cả các dữ liệu đó đều được lưu trên ram giúp tốc độ đọc ghi rất nhanh với các cơ chế hỗ trợ khác nhau persistence, high availability và replication.

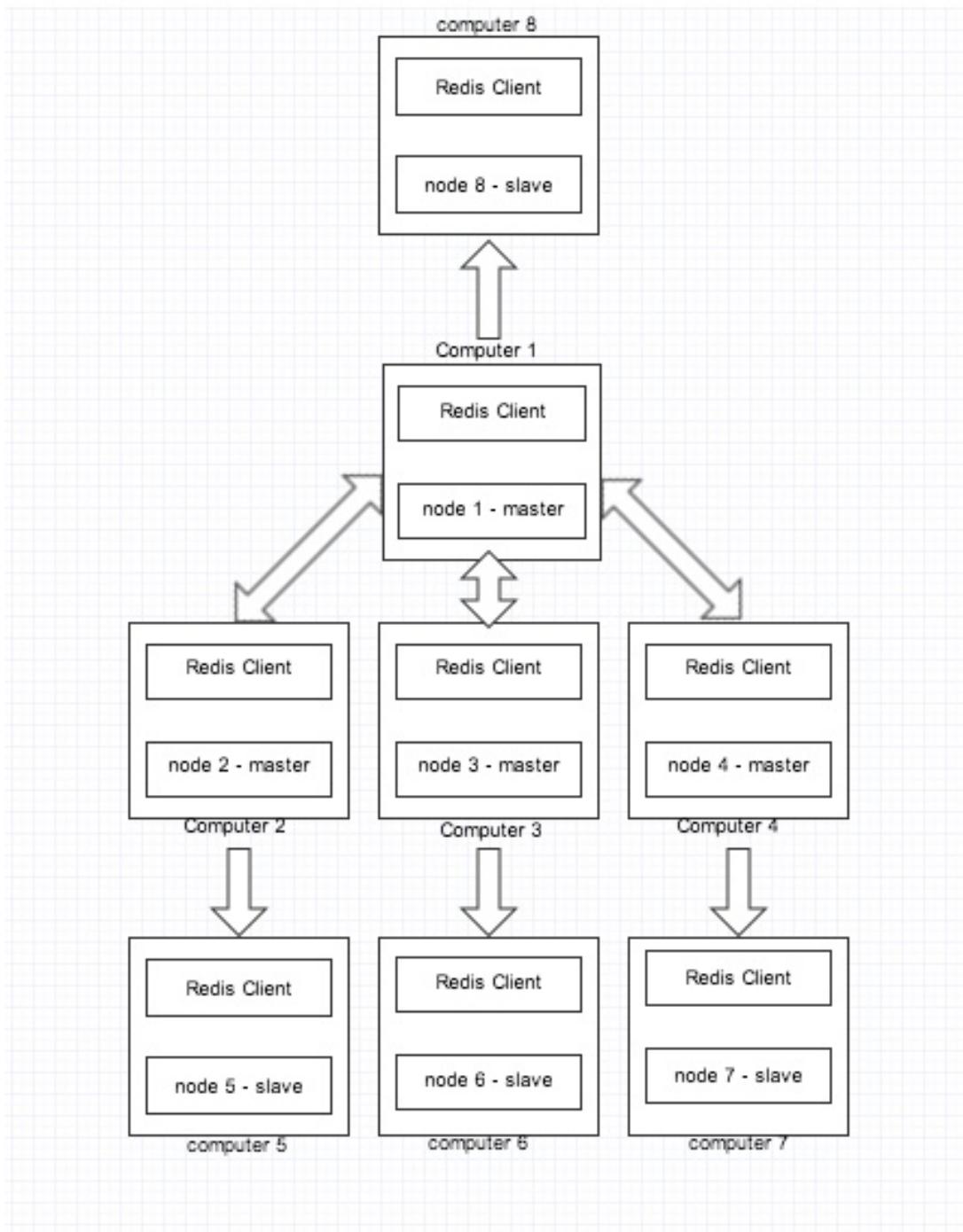
## 2.3.3 Ưu điểm

Redis hiện nay được phổ biến rộng rãi trong cộng đồng, do có một số ưu điểm nổi trội như

**In memory database** Tất cả dữ liệu trên redis server đều được lưu dưới bộ nhớ ram, ngược lại với các cơ sở dữ liệu truyền thống như PostgreSQL, Cassandra, MongoDB hầu hết chúng đều lưu dưới đĩa. Khi so với kiểu cơ sở dữ liệu truyền thông thì Redis nhanh hơn rất nhiều do các thao tác với dữ liệu đều trên ram thay vì với ổ đĩa. Redis có thể hỗ trợ nhiều thao tác hơn với thời gian phản hồi nhanh hơn trong cùng đơn vị thời gian với hàng triệu thao tác đọc ghi trên giây.

**Flexible data structures** Redis cung cấp nhiều loại cấu trúc dữ liệu khác nhau mà hầu hết các ứng dụng hiện nay sử dụng, cụ thể như Strings, Lists, Sets, Sortedset, Hashes, Bitmaps, HyperLogLog. Đặc biệt là HyperLogLog là một cấu trúc dữ liệu xác suất dùng để đếm các số lượng phần tử phân biệt trong tập data.

**Replication, High availability** Redis cung cấp kiến trúc Clustering và Replication đảm bảo redis high availability. Khi có một node trong cluster bị lỗi, thì nó sẽ có cơ chế thay thế, đưa slave lên thành master. Bên cạnh đó các dữ liệu được chia ra nằm ở các node trong cụm cluster để tăng hiệu suất đọc ghi. Ví dụ dữ liệu có 64GB mà trong cụm cluster có 10 node, mỗi node sẽ chứa 6.4 GB dữ liệu.

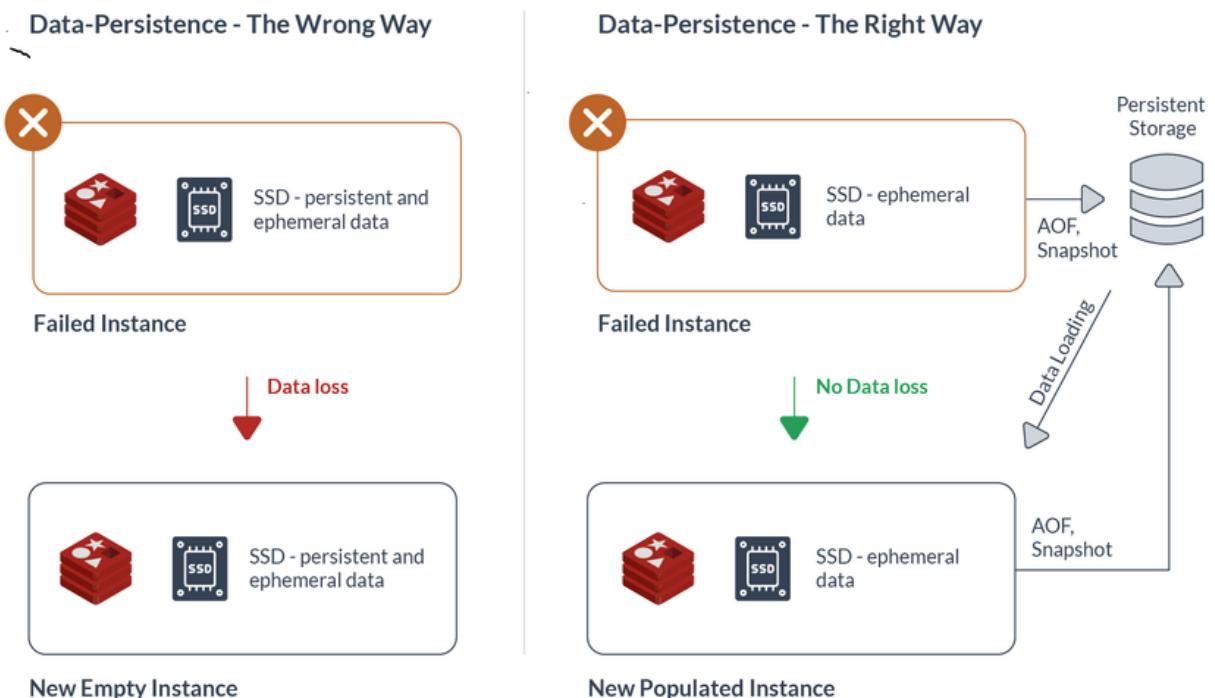


Hình 2.14: Kiến trúc Replication của redis

**Persistent** Do dữ liệu trên redis được lưu trên Ram nếu khởi động lại thì toàn bộ dữ liệu sẽ mất, vì vậy cần cơ chế Persistent. Cơ chế Persistent đảm bảo cho các trường hợp không mong đợi đột然 Redis server bị dừng và phải khởi động lại, có hai cách khác nhau

**RDB Mechanism** RDB sẽ sao chép tất cả dữ liệu hiện tại của Redis server vào một nơi lưu trữ khác, điều này diễn ra cố định sau một khoảng thời gian. Với RDB thì sẽ có khả năng bị mất dữ liệu khi Redis bị lỗi sau lần lưu trữ gần nhất

**AOF** sẽ ghi log lại các thao tác của client với Redis server, vì thế tất cả dữ liệu sẽ đảm bảo không bị mất khi Redis server bị lỗi, nhưng có vấn đề việc ghi log như vậy tốn tài nguyên và chi phí khi ghi lên đĩa.



Hình 2.15: Cơ chế hoạt động Persistence của Redis theo kiểu AOF

Simplicity and ease-of-use Bên cạnh các ưu điểm về hiệu suất thì lý do đơn giản mà Redis được phổ biến rộng rãi là dễ dàng sử dụng. Hiện nay ở hầu hết các ngôn ngữ đều cung cấp thư viện redis client để kết nối đến Redis server như C++, Java, Golang,...

## 2.3.4 Các kiểu nổi bật dữ liệu của Redis

### a) List

Redis list là danh sách các chuỗi được sắp xếp theo thứ tự truyền vào, có thể thêm vào đầu bằng LPUSH hoặc vào cuối bằng RPUSH. tương tự như vậy ta cũng có thể lấy các phần tử hoặc cập nhật các phần tử trong List. Redis Lists là danh sách của các chuỗi, Sắp xếp theo thứ tự chèn vào. Bạn có thể thêm element tới một List Redis vào đầu hoặc vào cuối. Chiều dài tối đa của một list là hơn 4 tỉ của các element/list, các thao tác thêm hoặc xóa các element cuối hoặc đầu list với độ phức tạp hằng số. Các thao tác ở giữa list (với list nhiều phần tử) thì chi phí tương đương O(n). Ví dụ sử dụng với kiểu dữ liệu List của Redis

```
1 127.0.0.1:6379 > LPUSH fresher chithuc
2 (integer) 1
3 127.0.0.1:6379 > LPUSH fresher 21
4 (integer) 2
5 127.0.0.1:6379 > LPUSH fresher VNG
6 (integer) 3
7 127.0.0.1:6379 > LRANGE fresher 0 2
8 1) "VNG"
9 2) "21"
10 3) "chithuc"
```

### b) Sets

Redis Sets là một tập có thứ tự của các chuỗi, bạn có thể thêm, xóa, kiểm tra sự tồn tại của chuỗi trong sets với độ phức tạp O(1). Các giá trị trong set không có trùng lặp. Số lượng lớn nhất của member trong set là hơn 4 tỉ phần tử/set. Dùng lệnh sadd để thêm giá trị vào Sets, và smembers để kiểm tra tất cả các giá trị có trong Sets đó. Ví dụ sử dụng với kiểu dữ liệu Sets

```
1 127.0.0.1:6379 > SADD fresher thuc
2 (integer) 1
3 127.0.0.1:6379 > SADD fresher 21 vng
4 (integer) 2
```

```
5 127.0.0.1:6379 > SMEMBERS fresher
6 1) "21"
7 2) "thuc"
8 3) "vng"
```

### c) Hashes

Redis hash là lệnh sử dụng để quản lý các key/value trong đó value có giá trị là hash. vì thế chúng là kiểu dữ liệu hoàn hảo đối với các object. Trong redis mỗi hash có thể lưu trữ tới hơn 4 tỷ cặp field-value. Ví dụ sử dụng với kiểu dữ liệu Hash HGET key field: lấy giá trị field trong hash.

```
1 127.0.0.1:6379 > HMSET user:1 ten "thuc" tuoi "21"
2 OK
3 127.0.0.1:6379 > HGET user:1 ten
4 "thuc"
5 127.0.0.1:6379 > HGET user:1 tuoi
6 "21"
```

### d) Sorted set

Kiểu dữ liệu tương tự như Redis Sets, không lặp lại giá trị. Điểm khác biệt ở đây là Sorted Set mỗi giá trị liên kết với một số được xem là độ ưu tiên của số đó (có thể lặp lại), điểm số này là cơ sở cho việc tạo ra Sorted Sets có thứ tự. Với Sorted Set ta có thể thêm, xóa, kiểm tra tồn tại các phần tử trong set với tốc độ rất nhanh, kể cả phần tử giữa Set.

ZRANGE: lấy các phần tử trong tập hợp từ start đến stop theo giá trị score của chúng. Ví dụ sử dụng

```
1 127.0.0.1:6379 > ZADD fresher 1 value1
2 (integer) 1
3 127.0.0.1:6379 > ZADD fresher 2 value2
4 (integer) 1
5 127.0.0.1:6379 > ZADD fresher 5 value5
6 (integer) 1
7 127.0.0.1:6379 > ZADD fresher 3 value3
8 (integer) 1
9 127.0.0.1:6379 > ZRANGE fresher 0 3 WITHSCORES
10 1) "value1"
```

```
11 2) "1"  
12 3) "value2"  
13 4) "2"  
14 5) "value3"  
15 6) "3"  
16 7) "value5"  
17 8) "5"
```

### e) HyperLogLogs

Là cấu trúc dữ liệu để đếm các phần tử phân biệt trong tập dữ liệu lớn, với hyperloglog kể cả cho số lượng phần tử ngày càng lớn đi nữa thì ta vẫn không cần tăng lượng bộ nhớ sử dụng, đánh đổi lại thì tần suất xác của hyperloglog chỉ đạt 99

Dưới đây là các câu lệnh được sử dụng với hyperloglog

pfadd: thêm phần tử mới vào hyperloglog pfcount: đếm các phần tử phân biệt có trong tập hyperloglog

```
1 > pfadd hll a b c d  
2 (integer) 1  
3 > pfcount hll  
4 (integer) 4
```

### 2.3.5 Cách sử dụng đơn giản với Redis-cli

Đầu tiên, phải cài đặt redis server thông qua câu lệnh

```
1 sudo apt-get install redis-server
```

Để khởi động service lên, dùng câu lệnh

```
1 redis-server
```

Để tương tác với server thì ta dùng redis-cli, mặc định sau khi cài đặt thì redis-cli sẽ kết nối với máy chủ redis nằm ở localhost:6379. Tiếp theo chúng em sẽ thử tương tác với redis server thông qua ví dụ minh họa dưới đây.

```
1 set foo Hello World  
2 get foo
```

Lệnh SET thực hiện một phép gán. Lưu ý, SET sẽ thay thế bất kỳ giá trị hiện có nào đã được lưu trữ trong key, trong trường hợp key đó đã tồn tại, ngay cả khi key được liên kết với giá trị không phải là chuỗi.

Lệnh GET lấy giá trị của một key ra và in ra màn hình console

### 2.3.6 Kết luận

Tóm lại Redis rất hiệu quả cho việc lưu trữ Cache, tốc độ đọc ghi nhanh phù hợp cho các ứng dụng yêu cầu hiệu suất tốt. Hiện nay hầu hết các ông lớn trong ngành như Twitter, GitHub, Weibo,.. đều sử dụng Redis.

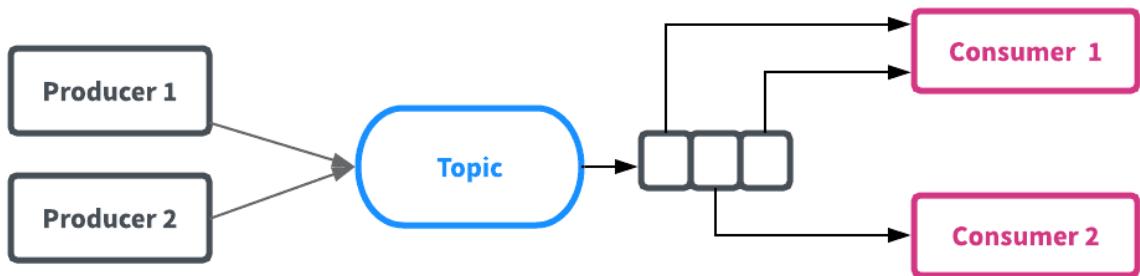
## 2.4 Message queue

### 2.4.1 Vấn đề

Để đáp ứng nhu cầu kiến trúc microserver với cơ chế giao tiếp asynchronous giữa các service trong hệ thống của chúng em thì hệ thống message queue không thể thiếu. Với nhiều công nghệ message queue hiện tại thì Kafka Apache đang được ưa chuộng và sử dụng phổ biến trong giới lập trình với các ưu điểm nổi bật và hiệu suất cao trong hệ thống distributed system

Dầu tiên, cần phải hiểu rõ được nguyên lý làm việc của message queue. Message Queue được thiết kế cho các tình huống như tasklist hoặc workqueue. Message Queue nhận được tin nhắn đến và đảm bảo rằng mỗi tin nhắn sẽ được gửi cho một topic hoặc channel và xử lý bởi chính xác một consumer.

Message Queue có thể nâng cao khả năng xử lý message lên cao bằng cách thêm nhiều consumers cho mỗi topic, nhưng chỉ duy nhất một consumer sẽ nhận được mỗi message ở topic này. Để đảm bảo rằng một message chỉ được xử lý bởi một consumer, mỗi message sẽ bị xóa khỏi queue sau khi nó được một consumer nhận và xử lý (tức là một khi consumer đã thừa nhận sử dụng message đến message system).



Hình 2.16: Hình ảnh mô tả cách hoạt động của Message Queue

Message Queue hỗ trợ các trường hợp mà trong đó điều quan trọng là mỗi message được xử lý chính xác một và chỉ một lần duy nhất, nhưng không cần thiết phải xử lý message theo thứ tự. Trong trường hợp lỗi từ mạng hoặc từ consumer, Message Queue sẽ thử gửi lại message sau (không nhất thiết phải cho cùng một consumer) và kết quả là message có thể bị xử lý không theo thứ tự.

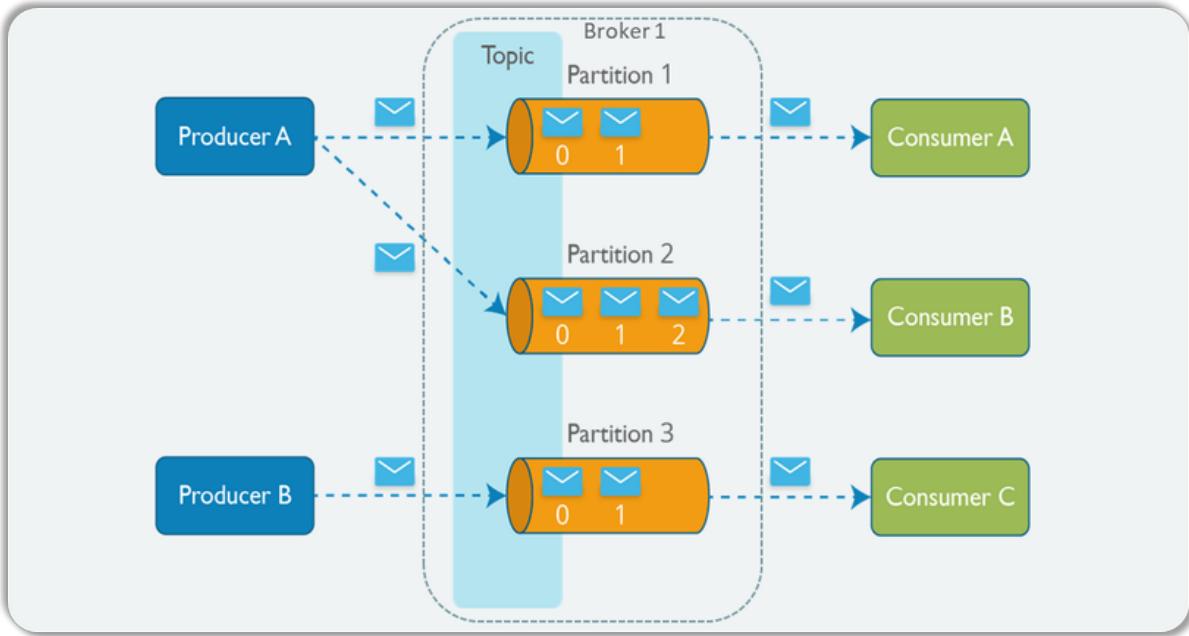
#### 2.4.2 Kafka là gì?

Kafka là một hệ thống message theo cơ chế Pub-Sub. Bên người gửi dữ liệu được gọi là producer, bên người nhận dữ liệu theo topic để xử lý được gọi là consumer.

Kafka có một số thuật ngữ riêng biệt, cụ thể như sau

Producer Kafka phân loại message theo topic, sử dụng producer để đẩy các message vào các topic khác nhau. Dữ liệu được gửi đến các partition của topic lưu trữ trên Broker. Mặc định các producer sử dụng thuật toán Load Balancing để đẩy các message vào Partition trên topic, ngoài ra còn có thể chỉ định các loại Message cụ thể lên Partition cụ thể bằng cách override lại thuật toán hashing của Producer

Consumer: để lấy message từ các topic khác nhau thì Kafka sử dụng consumer. Mỗi consumer thuộc về một group riêng được gọi là group-id, các consumer thuộc cùng group thì không thể cùng lấy message từ cùng Partition trên cùng topic, nhưng khác group-id thì có thể.



Hình 2.17: Hình ảnh mô tả tổng quát cơ chế hoạt động của Kafka

**Topic:** Dữ liệu truyền trong Kafka theo topic, khi cần truyền dữ liệu cho các ứng dụng khác nhau thì sẽ tạo ra các topic khác nhau.

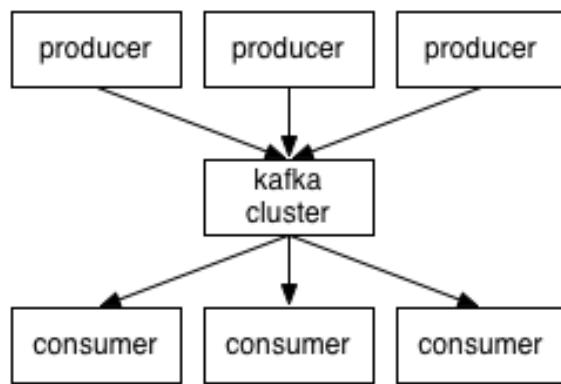
**Partition:** Đây là nơi dữ liệu cho một topic được lưu trữ. Một topic có thể có một hay nhiều partition. Trên mỗi partition thì dữ liệu lưu trữ cố định và được gán cho một ID gọi là offset. Trong một Kafka cluster thì một partition có thể replicate (sao chép) ra nhiều bản. Trong đó có một bản leader chịu trách nhiệm đọc ghi dữ liệu và các bản còn lại gọi là follower. Khi bản leader bị lỗi thì sẽ có một bản follower lên làm leader thay thế. Nếu muốn dùng nhiều consumer đọc song song dữ liệu của một topic thì topic đó cần phải có nhiều partition.

**Broker:** Kafka cluster là một tập các server, mỗi một tập này được gọi là 1 broker, cung cấp cơ chế master-slave đảm bảo kafka broker sẵn sàng cho các producer đưa các message đến.

**Zookeeper:** được dùng để quản lý thông tin metadata và điều phối các broker. Khi có một leader một lỗi thì Zookeeper sẽ nhận được thông tin và lựa chọn một trong các follower để đưa lên làm leader.

### 2.4.3 Cách hoạt động của Kafka

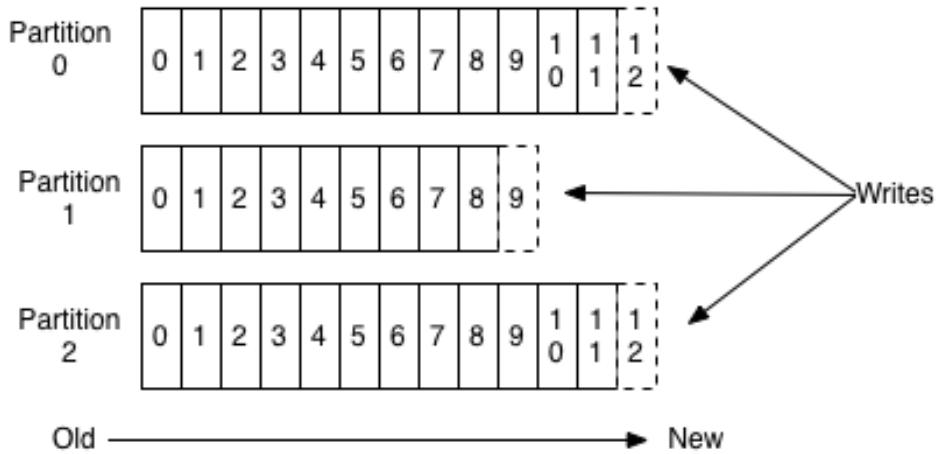
Application (producers) gửi các message đến (records) Kafka node (broker), sau khi Broker đã nhận được thì các producer được nhận thông báo là đã gửi thành công và đang đợi để được xử lý bởi ứng dụng khác (Consumers). Các consumers này trước đó phải subscribe vào các topic mà producer gửi lên.



Hình 2.18: Mối quan hệ giữa Producer, cluster và Consumer ở Kafka

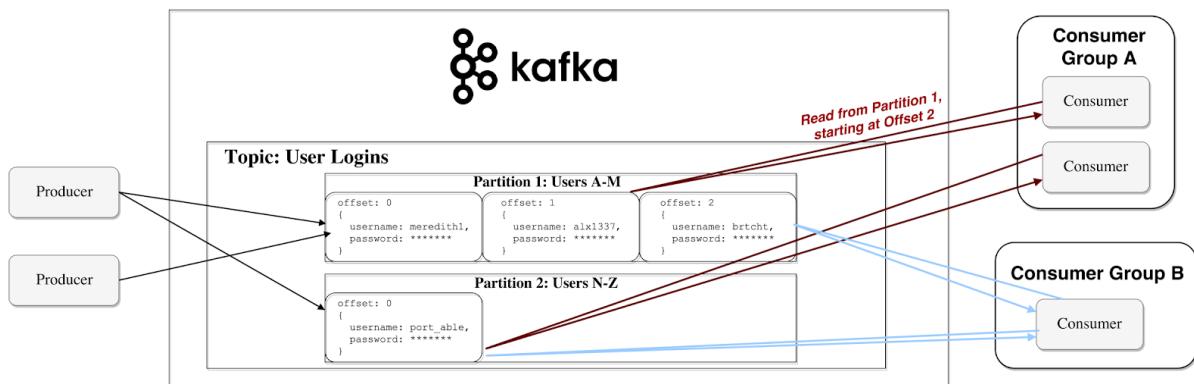
Vì Topic có thể quá tải khi application gửi các message liên tục, do đó chúng được chia nhỏ ra thành các Partition, vừa giúp chia tải tăng hiệu suất vừa có khả năng dễ dàng scale lên. Kafka đảm bảo rằng tất cả các message trên partition luôn được sắp xếp theo thứ tự đã được gửi lên và được các ứng dụng khác consume theo đúng nguyên lý First In First Out. Trên các Topic, để phân biệt các message với nhau thông qua offset, có thể xem các offset như là một vị trí các phần tử trong mảng. các vị trí được tăng dần với các message đến sau trong partition.

## Anatomy of a Topic



Hình 2.19: Cách tổ chức dữ liệu các record trên Broker

Kafka hoạt động theo nguyên tắc dumb broker và smart consumer. Có nghĩa là Kafka Broker không lưu lại thông tin vị trí record đã được đọc bởi consumer và xóa chung, thay vào đó Broker lưu chúng trong một khoảng thời gian và xóa đi các record cũ khi kích thước đạt ngưỡng. Đối với các Consumer thì báo với Broker là message nào muốn lấy, và có thể thông báo Broker rằng message này đã được lấy về thành công. Điều này cho phép các Consumer có thể tăng/giảm vị trí của các offset mà chúng muốn lấy về, cho phép khả năng xử lý các message khi xảy ra lỗi ở các Consumer.



Hình 2.20: Cơ chế hoạt động Smart Consumer ở Kafka

## 2.4.4 Tại sao sử dụng Kafka

Hiện nay, Kafka được hầu hết mọi người lựa chọn là ưu tiên bởi các ưu điểm nổi trội sau đây

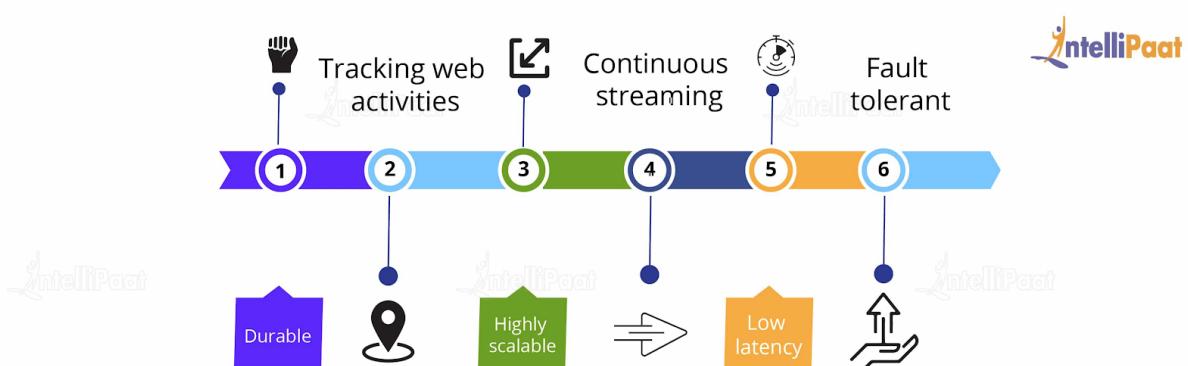
Low latency, High Throughput giúp tăng hiệu suất của hệ thống.

Fault tolerance cung cấp tính năng đảm bảo Kafka vẫn hoạt động với một số trường hợp một node bị lỗi với cơ chế cluster.

Durability Kafka cung cấp khả năng replication, các message được gửi lên topic được lưu trữ trên đĩa, vì vậy khi có lỗi xảy ra vẫn có thể đảm bảo các message không bị mất đi.

Reduces the need for multiple integrations tất cả dữ liệu được gửi thông qua Kafka. Do đó, chỉ cần các ứng dụng integration với kafka thông qua producer và consumer, giúp việc sử dụng và tích hợp dễ dàng.

Complexity việc giải quyết các vấn đề liên quan đến asynchronous đòi hỏi nhiều kiến thức và kinh nghiệm như vấn đề consumer xử lý hai lần cho cùng một yêu cầu, cách trả về kết quả cho client,... Distributed system, Scalability Apache Kafka cung cấp kiến trúc phân tán bên trong nó với producer, consumer và broker được tổ chức riêng biệt. Cùng với cơ chế của Partitioning and replication cung cấp khả năng Scalability một cách dễ dàng



Hình 2.21: Các ưu điểm nổi trội của Kafka

## 2.4.5 Các trường hợp sử dụng

Sử dụng như một hệ thống message queue thay thế cho ActiveMQ hay RabbitMQ

Tracking hành động người dùng : các thông số như page view, search action của user sẽ được publish vào một topic và sẽ được xử lý sau

Log Aggregation: log sẽ được gửi từ nhiều server về một nơi thống nhất, sau đó có thể được làm sạch và xử lý theo một logic nào đó

Event-Sourcing: Lưu lại trạng thái của hệ thống để có thể tái hiện trong trường hợp system bị down.

## 2.4.6 Kết luận

Với tất cả những tinh chất và ưu điểm đã kể phía trên, Kafka hoàn toàn phù hợp với những gì chúng em cần để sử dụng trong hệ thống, sử dụng kafka như một Message Queue cho cơ chế asynchronous trong hệ thống Microservice.

# 2.5 Container

## 2.5.1 Vấn đề

Có một cặp câu hỏi và câu trả lời rất thú vị trong ngành lập trình làm ứng dụng, khi mà bên làm ứng dụng đưa sản phẩm cho người dùng, và người dùng với mong muốn là mở ứng dụng lên nó sẽ chạy ngon lành. Nhưng thực tế không phải lúc nào mọi chuyện cũng suôn sẻ, vì là con người nên sẽ có những sai lầm, ứng dụng có thể lỗi nhỏ, có thể crash và mở không lên. Lỗi có thể là do ứng dụng, một use case nào đó chưa được test, hoặc mạng chập chờn, ... Có tỉ lệ lỗi có thể xảy ra và khiến trải nghiệm người dùng tệ đi khi gặp lỗi trên ứng dụng.

Dương nhiên là lỗi gì cũng có thể xảy ra, nhưng có thể hạn chế được một số lỗi nghe có vẻ là hiển nhiên. Ví dụ, phần mềm được lập trình

và chạy trên hệ điều hành windows, hẳn là không thể chạy trên linux và ngược lại. Tại vì sao ạ ? Tại vì một phần mềm khi lập trình, không thể tránh khỏi việc sử dụng những thư viện của hệ điều hành, hoặc thư viện độc lập bên ngoài. Mà chắc gì thư viện X của hệ điều hành windows cung cấp đã có trên hệ điều hành linux và ngược lại. Vậy ví dụ những phần mềm viết bằng ngôn ngữ lập trình C, trên các hệ điều hành linux và cụ thể là ubuntu có thể sử dụng compiler gcc, còn trên hệ điều hành windows thì dùng compiler do microsoft cung cấp nằm trong bộ phần mềm Visual Studio, khác biệt là điều khó tránh khỏi.

Dẫn đến khi khách hàng hỏi tại sao phần mềm không chạy được, thì câu trả lời huyền thoại là: "Nhưng mà phần mềm này chạy ngon lành trên máy của em". Do đó suy nghĩ đơn giản cho vấn đề này là, không chỉ cung cấp phần mềm cho khách hàng, mà cung cấp cả môi trường để phần mềm đó chạy được. Đó là cách mà container ra đời. đương nhiên là đây là thuật ngữ trong lĩnh vực công nghệ thông tin chứ không phải là trong lĩnh vực vận tải.

### 2.5.2 Mô tả

Container trong lĩnh vực vận tải giải quyết bài toán bốc và dỡ hàng hóa. Tại vì hàng hóa là đa dạng, có nông sản, có thịt, có cá, có cả máy móc và kim loại, ... Ứng với mỗi loại hàng hóa khác nhau sẽ có cách bốc dỡ hàng hóa khác nhau. Tại vì, mỗi loại hàng hóa tùy theo tính chất của nó mà sẽ được đóng gói lên phương tiện vận chuyển khác nhau, do đó cách xử lý cũng khác nhau. Không thể nào quăng quật các gói hàng dễ vỡ như thủy tinh giống như cách vận chuyển kim loại được. Mọi thứ cứ như thế cho đến khi container xuất hiện, và nó thực sự là cách mạng không ngành vận tải. Container trong vận tải là một hộp kim loại với kích thước tiêu chuẩn. Chiều dài, chiều rộng và chiều cao của mỗi container là như nhau. Đây là một điểm rất quan trọng. Nếu thế giới không có quy ước chung về kích thước tiêu chuẩn, việc sử dụng container sẽ không được phổ biến rộng

rãi và trở thành quy tắc chung của việc vận chuyển hàng hóa như bây giờ.

Container trong lĩnh vực công nghệ thông tin hoạt động có phần giống với container trong lĩnh vực vận tải. Nếu như container trong lĩnh vực vận tải đóng gói hàng hóa thì container trong lĩnh vực công nghệ thông tin đóng gói phần mềm. Container đóng gói phần mềm trước khi chuyển giao cho khách hàng hoặc trước khi cung cấp phần mềm rộng rãi trên mạng internet.

### 2.5.3 Kế thừa

Tất nhiên trước khi giải pháp container ra đời thì đã có những giải pháp khác, đó là máy ảo (Virtual Machine). Khi cung cấp phần mềm, thì đóng gói nó trong một máy ảo, và máy ảo đó chỉ dùng để chạy cụ thể phần mềm đó. Và việc cách ly này dường như hiệu quả và hạnh phúc, vì phần mềm bây giờ có thể tron tru nhờ vào máy ảo độc lập hóa, không xảy ra lỗi không tương thích, cũng như không có ứng dụng nào khác làm ảnh hưởng; nhưng máy ảo rất tốn tài nguyên. Tại vì một máy ảo là một hệ điều hành con trong hệ điều hành máy chủ đang chủ, nó yêu cầu ảo hóa phần cứng vô cùng phức tạp. Nói nôm na là bạn yêu cầu cả một con tàu to bự chỉ để chở vài kỳ chuối giao cho người dùng, nó cực kỳ lãng phí.

Giải pháp tiếp theo phải nhẹ hơn máy ảo và vừa phải đáp ứng đủ nhu cầu đóng gói ứng dụng. Và đó là cách container ra đời. Container có thể dùng để chạy một ứng dụng đơn giản cho đến ứng dụng phức tạp. Trong container chứa đầy đủ mọi file cần để chạy phần mềm, bao gồm mã nhị phân, thư viện và các tập tin cấu hình. So sánh với máy ảo và chạy phần mềm trên máy vật lý, thì container hiển nhiên nhẹ hơn vì không cần phải chứa cả hệ điều hành để chạy. Với một ứng dụng phức tạp, có thể phải cần nhiều container để chạy tạo thành cluster, phục vụ cho mục đích scale.

## 2.5.4 Lợi ích

Sử dụng ít tài nguyên hệ thống so với việc triển khai ứng dụng trên máy ảo hoặc máy vật lý, đơn giản vì container không bao gồm toàn bộ hệ điều hành và những phần mềm của hệ điều hành. Container chỉ chứa vừa đủ để chạy được phần mềm.

Tính linh động cao. Vì container đã chứa sẵn phần mềm, nên không cần phải quan tâm cụ thể phần mềm cần phải xử lý như thế nào, mà chỉ quan tâm vào việc xử lý container. Container đã thành chuẩn chung, thì mọi hệ điều hành đều xử lý như nhau, khỏi tốn công xử lý, cấu hình riêng từng phần mềm. Chuẩn chung mang đến lợi thế nhiều như vậy, giống như container trong lĩnh vực vận tải, mọi cách xử lý từng loại hàng hóa đều quy về cách xử lý container thì trong lĩnh vực công nghệ thông tin, từng hệ điều hành không cần phải biết cách chạy mọi loại phần mềm, chỉ cần biết cách xử lý container là đủ, container sẽ lo hết mọi việc còn lại. Container lúc này giống như giải pháp một cho tất cả. Hiếm khi phát sinh những vấn đề lạ khi chạy phần mềm trong container. Bởi vì môi trường mỗi lần chạy là giống nhau bất kể hệ điều hành bên ngoài là như thế nào.

Container còn là nền tảng cho Microservice hoạt động. Bởi vì từng service sẽ được đóng gói chạy trong 1 container, dễ dàng cho việc quản lý. Bởi vì kiến trúc Microservice sẽ cần rất nhiều service, nếu mỗi service đều cần một máy vật lý, thì tài nguyên sẽ rất lãng phí.

## 2.5.5 Hạn chế

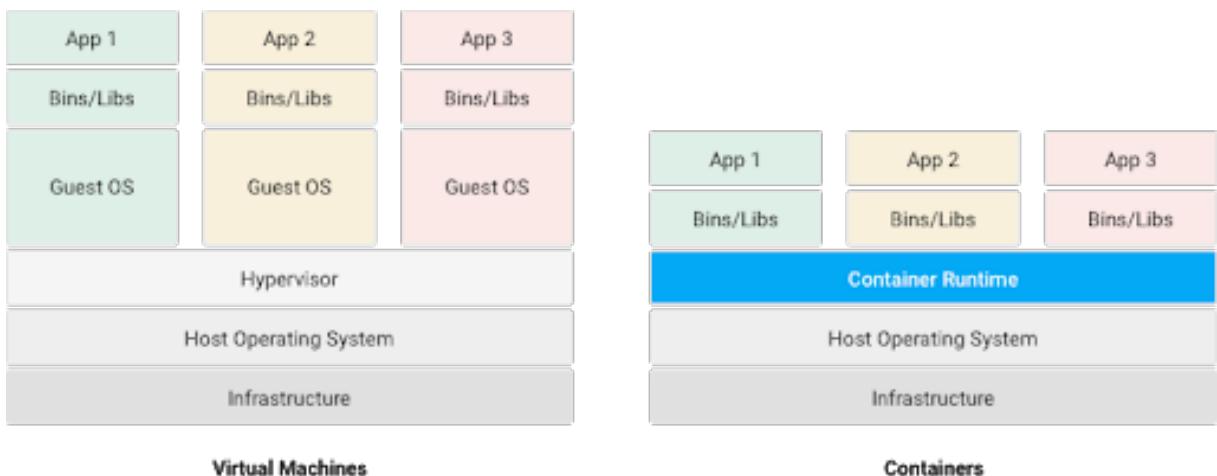
Tuy nhiên, không phải trường hợp nào cũng nên xài container. Một số use case mà container phát huy tác dụng của mình có thể kể đến như: Chuyển cấu trúc monolith qua Microservice. Microservice yêu cầu toàn bộ các phần mềm phải được đóng gói bằng container.

Container hoạt động bằng cách cung cấp một cơ chế đóng gói mà trong đó ứng dụng được có thể chạy trong môi trường được trừu tượng hóa khỏi môi trường thực sự mà ứng dụng đó chạy. Việc phân tách này cho phép ứng

dụng chạy trên nhiều nền tảng khác nhau, vì thực tế ứng dụng không còn giao tiếp với nền tảng hệ điều hành mà giao tiếp qua môi trường container cung cấp và sau đó container sẽ giao tiếp với hệ điều hành. Môi trường mà container cung cấp sẽ độc lập với hệ điều hành mà container chạy trên đó, và do đó thể hiện tính nhất quán. Ứng dụng X chạy trong container của máy chủ, của laptop lập trình viên, hoặc bất cứ máy nào, đều là như nhau. Container còn giúp cho lập trình viên tập trung vào duy nhất một việc, làm lập trình ứng dụng, bao gồm logic của ứng dụng cũng như các thư viện mà ứng dụng cần để chỉ. Còn team khác như devops sẽ lo phần deploy ứng dụng lên mạng internet hoặc cung cấp cho người dùng mà không cần quan tâm ứng dụng chạy như thế nào và logic ra sao, vì góc độ devops chỉ nhìn thấy container mà không thấy ứng dụng.

Container thường được so sánh với máy ảo nhiều nhất. Máy ảo đơn giản là một hệ điều hành con chạy trong hệ điều hành cha bên ngoài với sự truy cập đến phần cứng cũng được ảo hóa.

Máy ảo và container đều cho phép đóng gói phần mềm cùng với thư viện mà phần mềm yêu cầu đồng thời môi trường tách biệt so với môi trường của hệ điều hành bên ngoài. Nhưng container không ảo hóa phần cứng nên nhẹ hơn máy ảo.



Hình 2.22: So sánh giữa máy ảo và container

Thay vì ảo hóa toàn bộ phần cứng như cách mà máy ảo thực hiện,

container ảo hóa dựa trên hệ điều hành chạy bên ngoài container, bằng cách chạy trực tiếp trên nhân của hệ điều hành. Việc này làm cho container chạy cực kỳ nhẹ so với máy ảo, tại vì container sử dụng trực tiếp phần cứng được share từ nhân của hệ điều hành. Do đó container khởi động nhanh hơn, dùng ít bộ nhớ hơn là so với máy ảo.

Có nhiều định dạng container, nhưng nổi tiếng nhất và được sử dụng nhiều nhất là Docker, một định dạng container mã nguồn mở và được hỗ trợ bởi nhiều công ty lớn như Google, Microsoft.

Container cung cấp tính nhất quán về mặt môi trường, cho phép lập trình viên từ cài đặt được môi trường và không cho ứng dụng khác ảnh hưởng đến môi trường này. Container thực hiện điều này. Container còn có thể thêm các thư viện để phần mềm chạy được đúng phiên bản mà phần mềm yêu cầu. Ví dụ phần mềm yêu cầu thư viện đúng phiên bản v1.1.0 trong khi thư viện phiên bản mới nhất là v2.2.0 có thể không tương thích với phần mềm. Bởi vì thư viện đôi khi không hỗ trợ tương thích ngược, phiên bản mới sẽ thay đổi định nghĩa cũng như API của phiên bản cũ, làm phần mềm có thể không chạy được. Đứng từ góc độ lập trình viên, phần mềm được lập trình trên máy của lập trình viên thế nào thì sẽ chạy đúng như môi trường ấy, bất kể được deploy ở đâu. Điều này làm tăng năng suất, team devops sẽ bớt chuyện vò đầu bứt tai để debug lỗi môi trường của phần mềm, mà tập trung vào những thức khác.

Lợi ích của tính nhất quán về mặt môi trường đó là việc phần mềm chạy đa nền tảng. Tại vì các hệ điều hành linux, windows, mac đều thống nhất chung về cách chạy container, nên bản chất phần mềm đóng gói bằng container thì có thể chạy trên mọi hệ điều hành mà container có thể chạy. Bên cạnh đó, container có thể chạy trên máy cá nhân, máy server của công ty, và trên nền tảng điện toán đám mây. Điều này là nhờ chuẩn docker container trở nên quá rộng rãi và phổ biến. Do đó nếu muốn phần mềm có thể chạy đa nền tảng, đa môi trường, có thể đóng gói bằng container.

Container còn mang đến trong nó sự cô lập, mỗi phần mềm chạy trong môi trường độc lập với nhau. Container chạy độc lập với container khác,

độc lập về cả tài nguyên CPU, bộ nhớ, lưu trữ, mạng. Khả năng này giống như sandbox, phần mềm chạy trong container như chạy trong một hệ điều hành độc lập với mọi tài nguyên đều độc lập, không đụng chạm cùng như giàn giật từ tài nguyên của những phần mềm khác.

	Container Benefits	Virtual Machine Benefits
Consistent Runtime Environment	✓	✓
Application Sandboxing	✓	✓
Small Size on Disk	✓	
Low Overhead	✓	

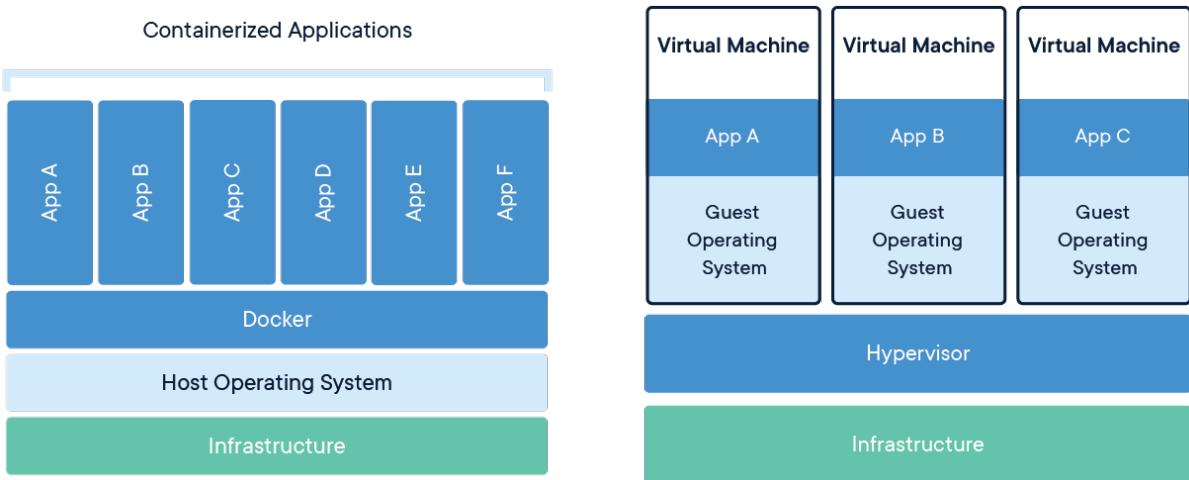
Hình 2.23: So sánh lợi ích của Container và máy ảo

### 2.5.6 Docker

Như chúng em đã nói ở trước là có nhiều định dạng container, trong đó docker container là phổ biến nhất. Docker container đã hầu như trở thành chuẩn chung khi đóng gói ứng dụng, và được các tập đoàn lớn hỗ trợ.

Docker đứng ra làm trung gian, các phần mềm đóng gói trong docker container giao tiếp với các tài nguyên hệ thống thông qua docker như được minh họa trong hình dưới.

Để đóng gói phần mềm bằng docker container, docker cung cấp cái gọi là docker image. Docker image là một file với ngữ pháp do Docker quy định dùng để khởi tạo môi trường, khởi tạo các thư viện cần cho phần mềm sử dụng, và phần mềm; docker image thường được gọi là Dockerfile.



Hình 2.24: So sánh thiết kế của Container và máy ảo

Ví dụ cho một Dockerfile cho một ứng dụng viết bằng ngôn ngữ Go như sau:

```

1 FROM golang
2
3 COPY .
4
5 RUN go get -d -v ./...
6
7 RUN go install -v ./...
8
9 CMD ["go-sample-app"]

```

Docker sẽ hiện thực hóa Dockerfile này trong quá trình chạy thành docker container.

Hình ảnh một docker container chạy như sau.

CONTAINER ID	IMAGE	COMMAND	NAMES	CREATED	STATUS	PORTS
36224fc90f04	minio/minio	"./usr/bin/docker-entrypoint.s...	external_minio_1	4 days ago	Up 4 days	0.0.0.0:9000->9000/tcp
4c642f99fa58	redis:alpine	"docker-entrypoint.s...	external_redis_1	4 days ago	Up 4 days	0.0.0.0:6379->6379/tcp
7d73e4de8164	rabbitmq:management-alpine	"docker-entrypoint.s..."	external_rabbitmq_1	4 days ago	Up 4 days	4369/tcp, 5671/tcp, 0.0.0.0:5672->5672/tcp
72->5672/tcp, 15671/tcp, 25672/tcp, 0.0.0.0:15672->15672/tcp	dd0b4387e818	"/bin/sh -c 'apt-get..."	vigilant_bassi	5 days ago	Exited (100)	5 days ago

Hình 2.25: Danh sách các Docker container đang chạy trên máy

## 2.6 AI Platform

### 2.6.1 Vấn đề

Hiện nay các phần mềm sử dụng AI được phổ biến rộng rãi, ví dụ như nhận diện mặt người trong ảnh, nhận diện mặt động vật trong ảnh. Chúng đều có một điểm chung khá giống nhau, đó là người sử dụng chọn input, input có thể là ảnh, video, là text, ... sau đó cho chạy phần mềm AI tương ứng, và chờ xuất ra output. Phần mềm AI chạy thường phải tốn kha khá một đoạn thời gian xử lý. Vậy nếu người dùng muốn, chạy phần mềm AI A, sau đó lấy kết quả chạy phần mềm AI B, rồi lấy kết quả chạy phần mềm AI C, để ra kết quả cuối cùng. Ví dụ như đầu tiên nhận dạng mặt người trong ảnh sau đó nhận diện mặt theo giới tính trong ảnh. Hoặc, cùng với một input, mà người dùng có thể chạy cùng lúc nhiều phần mềm AI xử lý input đó đồng thời.

### 2.6.2 Giải pháp

Chúng em đưa ra giải pháp là nền tảng cung cấp các phần mềm AI. Frontend sẽ là một website cho phép người dùng upload lên input, sau đó cho người dùng kéo thả các phần mềm AI mà người dùng muốn xài theo thứ tự mà người dùng muốn cho chạy. Có thể là chạy song song, có thể là chạy tuần tự, hoặc có thể rối rắm hơn xí như là: chạy phần mềm A, sau đó chạy song song B, C, D; lấy kết quả của B, C chạy tiếp E, sau đó chạy tiếp F.

Khi người sử dụng kéo thả xong, kết quả là cây tính toán, bên server backend sẽ nhận cây này và bắt đầu xử lý lần lượt với từng phần mềm AI.

Bài toán đặt ra là với từng phần mềm AI, yêu cầu tài nguyên khác nhau, với những cách chạy cách truyền file vào container để phần mềm AI xử lý thì tụi em sử dụng lớp bọc phần mềm AI một lần nữa, ở ngoài là một service nhận input.

Khác nhau, làm sao có thể tích hợp nhanh nhất mà ít tốn kém việc

lập trình nhất? Dáp án đưa ra đó là đóng gói phần mềm AI trong một container. Những cách truyền file vào container để phần mềm AI xử lý thì tụi em sử dụng lớp bọc phần mềm AI một lần nữa, ở ngoài là một service nhận input.

Có thêm một vấn đề nữa xảy ra là, nếu đưa platform lên internet sẽ có rất nhiều người sử dụng cùng lúc. Nếu có 3 người dùng cùng một phần mềm AI cùng lúc, mà phần mềm AI lúc đó chỉ chạy trong 1 container, thì người đến sau sẽ phải chờ người đến trước hoàn thành xong rồi mới tới lượt, dẫn đến độ trễ cao và khiến trải nghiệm người dùng tệ đi. Cách giải quyết rất đơn giản đó là scale. Cho nhiều container chạy cùng lúc, sau đó áp dụng load balancing, lần lượt từng request sử dụng phần mềm AI sẽ vào từng container.

Như lúc nãy chúng em đã trình bày là gói phần mềm AI thêm một lớp service nhận input. Service này hoạt động vừa là HTTP service hoặc là GRPC service, tùy theo cách mà người dùng gọi lên. Thiết kế kiểu này có một nhược điểm là độ trễ, khi người dùng gọi lên, thì người dùng tạo một request và gọi lên cho server. Nếu lúc này chờ server xử lý xong thì sẽ rất lâu, và người dùng tưởng là server bị lỗi chảng hạn. Do đó giải pháp chúng em đề xuất là sử dụng async API. Khi người dùng tạo một request thì trên server sẽ gửi một mã id. Sao đó người dùng liên tục tạo request lên server với mã id này để kiểm tra xem xong chưa. Ở trên server, tụi em tách ra làm 2 service nhỏ, 1 con chuyên nhận request và 1 con chuyên xử lý request. Đối với con chuyên nhận request thì, mỗi request mới sẽ đưa vào hàng đợi cho con chuyên xử lý chờ xử lý. Còn con xử lý lúc nào cũng sẽ chờ hàng đợi có việc cần xử lý thì sẽ lấy ra xử lý. Cách thiết kế này có lợi cho việc scale. Vì chỉ cần hàng đợi có tồn tại, số lượng con nhận request và con xử lý request bao nhiêu cũng được. Và đương nhiên là càng nhiều càng tốt.

Lúc này có thêm một vấn đề nữa, đó là với càng nhiều phần mềm AI, thì số con nhận nhận xử lý request sẽ nhiều lên tương ứng. Phía bên người dùng sẽ phải ghi nhớ hết địa chỉ của những con nhận request, điều này là

không hợp lý lắm, vì cứ thêm một phần mềm AI thì người dùng phải thêm một địa chỉ, gây bất tiện. Giải pháp cho vấn đề này là một con đường ra trung gian, nhận request từ người dùng và sau đó đẩy request này đi tới đúng service AI để nhận request đó. Chúng em gọi service trung gian này là Gateway.

Có một vấn đề là phải lưu lại input của người dùng. Tại vì không thể lưu lại mọi input của người dùng trên server, vì đó sẽ vi phạm kiến trúc stateless của Microservices.

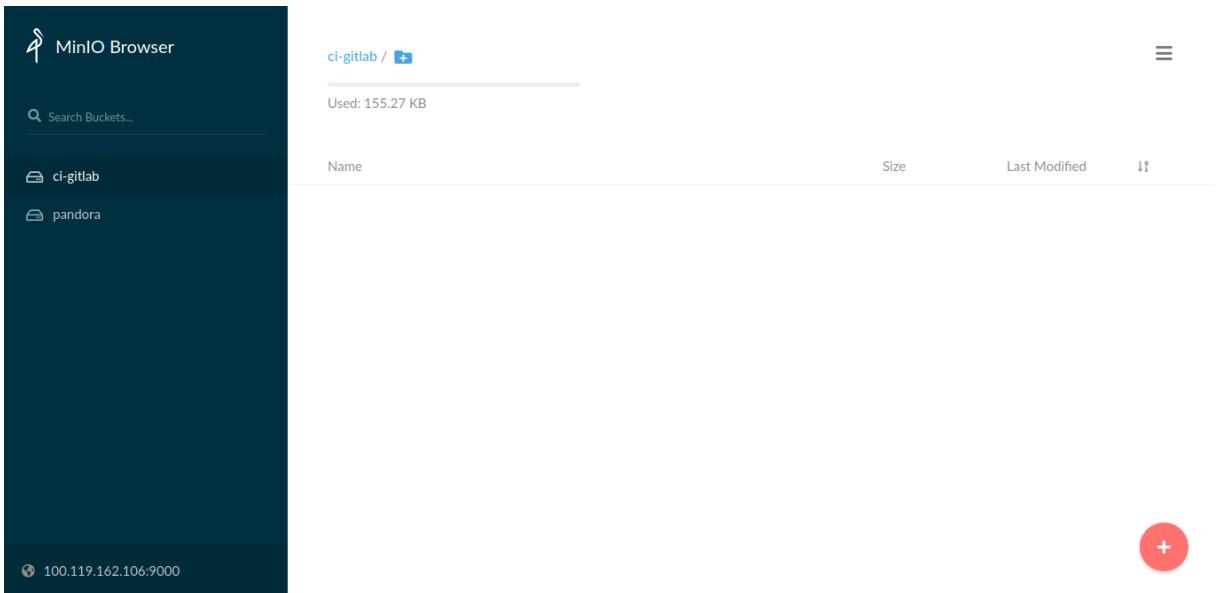
Stateless là thiết kế không lưu dữ liệu của client trên server. Có nghĩa là sau khi client gửi dữ liệu lên server, server thực thi xong, trả kết quả thì “quan hệ” giữa client và server bị “cắt đứt” – server không lưu bất cứ dữ liệu gì của client. Như vậy, khái niệm “trạng thái” ở đây được hiểu là dữ liệu.

Stateful là một thiết kế ngược lại, chúng ta cần server lưu dữ liệu của client, điều đó đồng nghĩa với việc ràng buộc giữa client và server vẫn được giữ sau mỗi request (yêu cầu) của client. Data được lưu lại phía server có thể làm input parameters cho lần kế tiếp.

## 2.7 Lưu trữ

Giải pháp cho việc lưu trữ input của người dùng là gửi lên một trung tâm quản lý file tập trung. Chúng em đề xuất sử dụng Minio làm dịch vụ quản lý file.

Minio là một object storage server được implement những public API giống như AWS S3. Điều đó có nghĩa là những ứng dụng có thể config để giao tiếp với Minio thì cũng có thể giao tiếp với AWS S3. Là một server lưu trữ object nên có thể được sử dụng để lưu trữ những unstructured data như ảnh, video, log files, backups và container/VM images. Dung lượng của 1 object có thể dao động từ một vài KB tới tối đa là 5TB. File cũng được gom lại trong 1 buckets, nó là được chỉ cùng với access key khi dùng app. Đây là giao diện của minio:



Hình 2.26: Giao diện của MinIO

Sẽ là một vấn đề lớn nếu lưu trữ ở cùng một server vì lượng dữ liệu này khá lớn. Chưa kể dữ liệu phải luôn được sao lưu. Minio là công cụ tốt để handle những điều trên. Nó tách những dữ liệu lưu trữ khỏi phần mềm và có thể truy cập thông qua HTTP. Ngoài tính năng tải xuống và tải lên dữ liệu, minio còn có tính năng chia sẻ đường link và đặt thời gian hết hạn cho đường link.

Chúng em đề xuất cách thiết kế là một service làm gateway, qua đó mọi yêu cầu mà người dùng muốn gọi đến các phần mềm AI đều phải qua gateway xử lý. Cụ thể gateway cung cấp cho người dùng API, gọi là public API. Khi người dùng sử dụng API này, thì gateway sẽ gọi đến những internal API giữa các service với nhau. Người dùng chỉ quan tâm đến API được public ra bên ngoài mà không cần quan tâm đến những API bên trong, giúp cho mọi chuyện đơn giản hơn. Nhớ 1 API dễ hơn là việc nhớ rất nhiều API. Chưa kể gateway còn giúp cho việc bảo mật được tốt hơn. Chỉ cần bảo mật đầu API gateway là đủ, vì những API nội bộ không public ra ngoài mạng internet nên không cần bảo mật nhiều. Việc bảo mật 1 đầu API so với bảo mật nhiều đầu API đương nhiên là lợi ích hơn. Chưa kể gateway còn có thể cài đặt rate-limiter, ngăn chặn việc API bị

tấn công. Gateway còn cho phép thống kê API một cách đơn giản hơn, chỉ cần thống kê ở đầu API của Gateway là được, không cần phải thống kê từng đầu API internal.

Cụ thể lợi ích của gateway bao gồm:

Ngăn chặn việc phơi bày các API nội bộ đến người dùng. Gateway tách API public và API nội bộ làm 2 phần. Do đó khi kiến trúc Microservice thay đổi, dẫn đến các API nội bộ thay đổi, thì API public vẫn không thay đổi về mặt cấu trúc. Dẫn đến khả năng refactor và thay đổi linh hoạt cho Microservice mà không ảnh hưởng nghiêm trọng đến API public đang phơi bày cho người dùng sử dụng. Ngăn chặn việc người dùng tò mò các service của hệ thống bằng cách chỉ cho phép người dùng truy cập thông qua một entry duy nhất.

Thêm một lớp bảo mật nữa cho kiến trúc Microservice. Vì việc bảo vệ một đầu API dễ hơn là việc bảo vệ tất cả các đầu API. Tại vì có rất nhiều kiểu tấn công vào API ví dụ như SQL Injection, lỗ hổng XML Parser, và tấn công kiểu từ chối dịch vụ (denial of service - DoS).

Gateway cho phép giải pháp sử dụng nhiều kiểu giao thức giao tiếp khác nhau. Nếu như người dùng thường sử dụng API qua các giao thức cung cấp qua HTTP như API theo chuẩn REST, thì các service nội bộ có thể giao tiếp với nhau bằng rất nhiều cách, có thể là theo chuẩn REST, hoặc gRPC với protobuf, hoặc sử dụng queue làm trung gian. Gateway tổng hợp hết mọi thứ làm 1, và người dùng chỉ cần biết public API là đủ.

Giảm sự phức tạp của Microservice xuống. Thông thường một API ngoài việc xử lý logic còn phải xử lý thêm một số chuyện như xác thực người dùng, rate limiting, và thống kê số lượng truy cập API theo ngày theo giờ chẳng hạn. Mỗi sự quan tâm như thế thêm một lớp phức tạp khi lập trình ứng dụng, gây tốn thêm thời gian cũng như công sức và tiền bạc. Gateway sẽ làm mọi công việc đó cho từng API nội bộ, cho phép lập trình viên quan tâm hơn vào việc lập trình và xử lý logic.

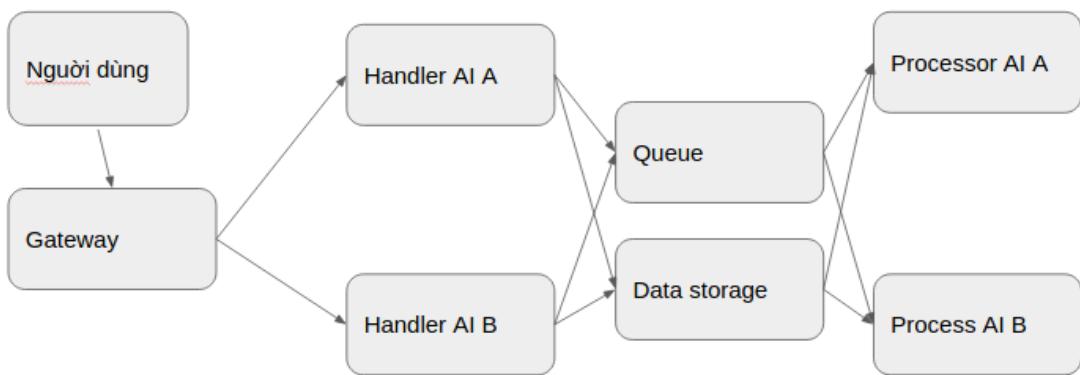
Gateway cho phép dễ dàng test các API public bằng cách mock các service trong kiến trúc Microservice mà không ảnh hưởng gì đến public

API nhờ vào việc phân tách sự phụ thuộc giữa người dùng và các service nội bộ.

Mặc dù lợi thế nhiều là vậy, thiết kế sử dụng gateway có một số hạn chế nhất định: Single point of failure. Nếu gateway gặp lỗi và crash thì toàn bộ service đứng sau nó đều không truy cập được. Mọi kiến trúc và service khi xử lý đều cần thêm API của gateway. Việc cấu hình kết nối từ API gateway đều phải chính xác ngay từ lúc deploy để chắc chắn là việc routing từ public API đến internal API chính xác và ổn định.

Gateway hiện nay có nhiều giải pháp cung cấp hỗ trợ sẵn, nhưng chúng em quyết định tự viết để tăng khả năng tùy biến trong nền tảng cung cấp phần mềm AI của mình.

Chúng em đề xuất kiến trúc như sau:



Hình 2.27: Tổng quan kiến trúc AI Platform thuở sơ khai

Khi người dùng gửi yêu cầu lên cho gateway, lúc này gateway sẽ phân tích yêu cầu này thành nhiều yêu cầu nhỏ. Ví dụ như khi người dùng gửi lên yêu cầu xử lý ảnh, và yêu cầu cả hai phần mềm AI cùng thực hiện: nhận diện mặt người và nhận diện động vật cùng lúc. Thì lúc này

gateway sẽ tạo 2 yêu cầu gửi đến cho handler của AI nhận diện mặt người và handler của AI nhận diện động vật. Sau đó. Handler của từng AI sẽ trả về cho gateway từng transID tương ứng. Gateway dùng transID này để gọi lên lại cho từng handler AI để biết được rằng phần mềm AI đã xử lý yêu cầu xong chưa. Sau khi trả về transID cho gateway, handler AI đẩy vào queue công việc.

Lúc này các processor của từng AI sẽ lấy công việc từ trong queue ra và xử lý, xử lý xong kết quả sẽ lưu trong data storage. Sau đó, gateway dùng transID gọi lên handler AI thì sẽ thấy yêu cầu đã được hoàn thành và có trả về kết quả.

Sau khi gateway hoàn thành xong hết các yêu cầu nhỏ thì sẽ tổng hợp lại ra kết quả và gửi về cho người dùng. Dường nhiên quá trình này tốn thời gian. Do đó ngay lúc đầu gateway trả về cho người dùng chỉ có transID, người dùng sử dụng transID này gửi lên gateway để biết được là yêu cầu của mình đã hoàn thành chưa.

Ưu điểm dễ thấy nhất của thiết kế này đó là, mọi yêu cầu của người dùng đều được xử lý nhanh chóng không phải chờ lâu, vì có trả về transID ngay lập tức. Sau đó có thể scale ra rất nhiều handler AI và rất nhiều processor AI để tăng tốc thời gian nhận và xử lý công việc liên quan đến phần mềm AI.

Việc gateway gọi API của handler AI chúng em đề xuất sử dụng gRPC để tăng tốc việc nhận và xử lý yêu cầu. Data storage chúng em sử dụng Minio, là để lưu đầu vào cũng như là kết quả của các phần mềm AI, có thể là ảnh, là video, hoặc là file văn bản, ...

Việc lưu trạng thái của từng công việc chúng em đề xuất sử dụng Redis để tăng tốc thời gian truy xuất nhanh chóng so với việc sử dụng database.

Thiết kế của chúng em đề xuất rất dễ dàng cho việc scale.

## 2.8 Scale

### 2.8.1 Vấn đề

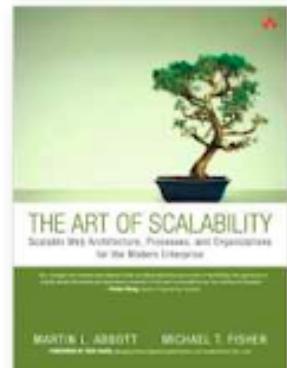
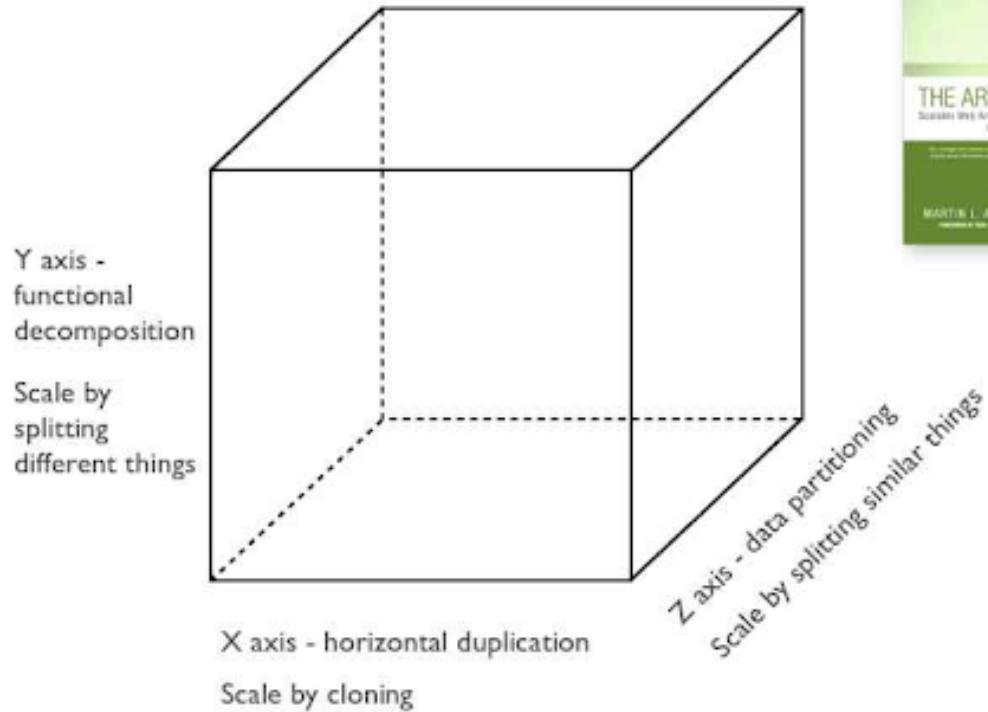
Hiện tại hệ thống của một trang web nào đó đang chịu tải là 1000 lượt truy cập cùng lúc trong 1 giây. Bây giờ muốn hệ thống có thể chịu tải thêm 1000 lượt truy cập nữa, tức là 2000 lượt truy cập trong 1 giây, thì phải làm thế nào? Có một cách suy nghĩ đơn giản thế này, hiện tại server đang chạy trên 1 máy thì có khả năng chịu tải 1000 lượt truy cập trong 1 giây, nếu giả sử có thêm 1 máy nữa để 2 máy cùng chạy server thì hy vọng sẽ chịu tải được 2000 lượt truy cập trong 1 giây. Đây chính là tinh thần của scale. Một máy tính chạy chương trình chậm thì cho nhiều máy tính cùng chia sẻ chương trình để cùng chạy. Ram ít thì mua thêm ram, ổ cứng SSD không đủ thì mua thêm nhiều ổ SSD. Có thể là nhân 2, nhân 3, thậm chí nhân nhiều lần các bộ phận của hệ thống lên để đáp ứng nhu cầu sử dụng lớn.

### 2.8.2 Mô tả

Scale hiểu nôm na tức là nhân đôi nhân ba thậm chí là nhân nhiều lần hệ thống lên. Để hệ thống có thể xử lý được nhiều yêu cầu từ người dùng nhất có thể. Giống như bài toán tiểu học, một người hoàn thành xong công việc trong 3 ngày, nhưng 6 người chỉ hoàn thành công việc trong nửa ngày. Tất nhiên mọi thứ không đơn giản như thế, nhưng nó cũng có một phần đúng.

Trong định nghĩa hình học, để scale một hình hộp, có thể thay đổi theo tỷ lệ chiều dài, chiều rộng cũng như chiều cao. Thì trong thiết kế hệ thống, scale cũng 3 chiều X, Y, Z

## 3 dimensions to scaling



Hình 2.28: Tổng quan 3 chiều scale

Scale theo chiều X nghĩa là chạy nhiều phiên bản copy của phần mềm cùng một lúc với phần mềm cân bằng tải (load balancer). Nếu có N bản copy phần mềm cùng chạy, thì mỗi bản copy của phần mềm chỉ chịu tải  $1/N$  so với việc chỉ có duy nhất một phần mềm chạy. Đây là tư duy đơn giản và thường được áp dụng khi scale ứng dụng.

Một nhược điểm của kĩ thuật scale này là, mỗi phiên bản của phần mềm đều xử lý và truy cập đến cùng dữ liệu cho nên thay vì phần mềm chịu tải thì đưa đến database chịu tải. Dù có scale 1 phần mềm lên 10 bản copy chạy song song thì 10 bản copy đều truy cập đến một database, dẫn đến dễ bottleneck ở phần database.

Scale theo chiều Y nghĩa là tách phần mềm bự ra thành nhiều service nhỏ hơn. Mỗi service chỉ làm một hành động duy nhất hoặc một số các hành động tương tự nhau. Có nhiều phương pháp để tách service: một

cách tiếp cận là dựa trên hành động, mỗi hành động cụ thể của phần mềm tách ra làm một service. Cách tiếp cận khác là dựa trên đối tượng, tất cả những hành động liên quan đến một đối tượng sẽ tách ra làm một service riêng. Scale theo chiều Y nghĩa là tách phần mềm bự ra thành nhiều service nhỏ hơn. Mỗi service chỉ làm một hành động duy nhất hoặc một số các hành động tương tự nhau. Có nhiều phương pháp để tách service: một cách tiếp cận là dựa trên hành động, mỗi hành động cụ thể của phần mềm tách ra làm một service. Cách tiếp cận khác là dựa trên đối tượng, tất cả những hành động liên quan đến một đối tượng sẽ tách ra làm một server riêng.

Scale theo chiều Z cũng có điểm giống so với scale theo chiều X ở chỗ, mỗi server chạy một phiên bản copy của phần mềm. Khác biệt ở chỗ, mỗi server phục vụ một tập nhỏ các yêu cầu trong toàn bộ yêu cầu. Ví dụ như cùng là website đặt hàng online, nhưng yêu cầu khách hàng VIP sẽ được xử lý trên một server riêng và yêu cầu của những khách hàng thường sẽ được xử lý trên một server riêng. Điều này khiến cho việc xử lý yêu cầu của khách hàng VIP nhanh hơn vì được dành riêng server chỉ để xử lý.

### 2.8.3 Kỹ thuật

Các kỹ thuật scale tập trung vào 3 hướng scale như trong mô tả. Để scale theo chiều X, thì các service nên thiết kế theo hướng không trạng thái (stateless) hơn là có trạng thái stateful. Không trạng thái và có trạng thái là hai thuật ngữ mới mà giới lập trình viên trong quá trình áp dụng hướng thiết kế microservice định nghĩa ra.

Với kiến trúc microservices, một hệ thống lớn sẽ có nhiều service con. Thường khi muốn chuyển đổi hệ thống monolith sang hệ thống microservices, một service lớn sẽ được tách thành nhiều service con. Ví dụ, với một ứng dụng đặt vé xem phim, sẽ có frontend và backend. Frontend có thể bao gồm mobile và web, frontend kết nối với backend thông qua các API. Đơn giản như người dùng vào màn hình đăng nhập, lúc nhập xong

tài khoản và mật khẩu và nhấn gửi đi, thì lúc này frontend sẽ gửi đi thông tin này cho backend. Sau khi backend nhận được thì sẽ kiểm tra tài khoản này thông tin có chính xác không. Cũng là ở frontend, khi người dùng chọn vào danh sách phim thì frontend cũng lập tức gọi API cho backend lấy danh sách phim. Nếu backend chỉ có một service thì service backend phải xử lý nhiều công việc không liên quan với nhau. Vấn đề này có thể được giải quyết bằng cách tách service cũ ra làm 2 service mới, một service xử lý phần đăng nhập đăng xuất, một service xử lý những công việc liên quan đến phim. Sau đó có thể scale từng service tương ứng, ví dụ service đăng nhập đăng xuất có thể ít lượng truy cập thì có thể chạy scale ra 3 instance. Trong khi service liên quan đến xử lý phim thì lượng truy cập sẽ nhiều hơn nên sẽ scale ra 5 instance chẳng hạn.

Để service có thể scale được thì service nên được thiết kế theo hướng stateless. Stateless là đối lập với stateful. Để hiểu được stateful, ta phải làm rõ về statefulness. Trước hết là state, state nghĩa là trạng thái. Đối với một hệ thống máy tính, state thể hiện trạng thái của một đối tượng tại một thời điểm nhất định. Ví dụ state của máy tính có thể là 2 trạng thái bật hoặc tắt. Và stateful nghĩa là dựa vào những khoảnh khắc thời gian, state hiện tại cùng với dữ liệu nhập vào và thay đổi trạng thái. Lấy ví dụ có một phần mềm, trả về có hoặc không dựa vào trạng thái mà phần mềm đang có. Trạng thái của phần mềm lưu dưới dạng 0 và 1, 0 thì trả lời không và 1 thì trả lời có. Và cứ cách một đoạn thời gian là sẽ thay đổi trạng thái 1 hoặc 0, thì đây là stateful. Tại vì phần mềm lúc nào cũng trả lời dựa trên trạng thái mà phần mềm đang giữ, câu trả lời không thể độc lập với trạng thái, nên đây là stateful.

Trong web service, stateful có thể được thể hiện thông qua hệ thống session. Mỗi người dùng khi đăng nhập sẽ được server cung cấp 1 session và lưu trữ ở ngay tại web service. Tuy nhiên, nếu web service được scale ra nhiều instance, thì thông tin session này phải được chia sẻ hết toàn bộ instance. Vì nếu không sẽ xảy ra trường hợp người dùng đăng nhập thành công ở instance 1, lúc sau làm mới trang web lại thì phát hiện chưa

đăng nhập vì đang ở instance 2.

Stateless là trái ngược với stateful, trở lại với căn phòng mới nãy. Giả sử lúc này câu trả lời không bị ảnh hưởng bởi trạng thái 0, 1, mà dựa vào điều kiện như nhập vào a thì trả lời có và nhập vào b thì trả lời không. Đây chính là stateless. Tại vì vào bất cứ thời gian nào, bất kể là trạng thái của phần mềm ra sao, thì câu trả lời luôn cố định tùy thuộc vào giá trị nhập vào, chứ không phải cùng một giá trị nhập vào, mà lúc này là có lúc kia là không.

Trở lại với web service, stateless có thể được thể hiện ở hệ thống token. Người dùng khi đăng nhập thành công sẽ được cung cấp một token. Sau đó người dùng sử dụng token này để làm mọi việc trong hệ thống. Và token này khi scale web service lên nhiều instance thì vẫn còn hợp lệ, và mỗi instance đều không cần lưu trữ token này làm gì.

# Chương 3

# Kiến trúc AI Platform

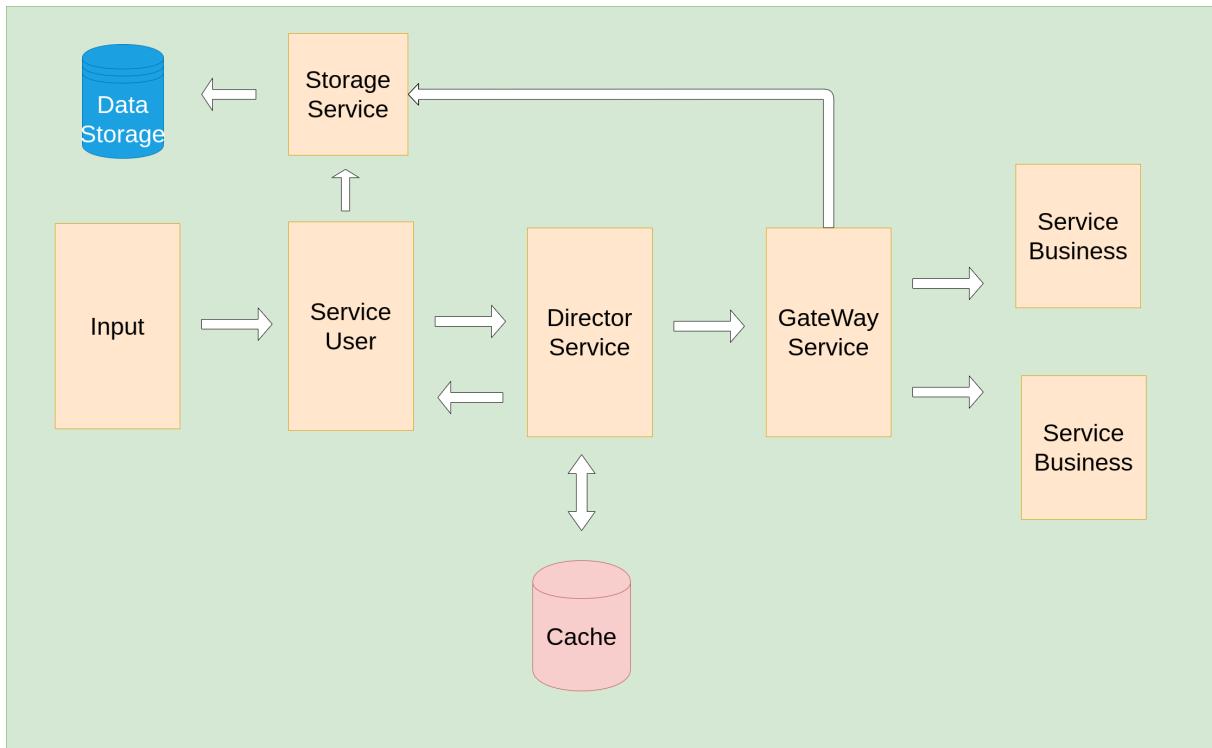
*Nội dung của chương 3 mô tả kiến trúc tổng quát và chi tiết các thành phần của các service trong hệ thống platform.*

## 3.1 Tổng quan

Với những vấn đề mà chúng em đã đề cập ở chương trước, thì Platform được thiết kế ra phải đảm bảo giải quyết những điều đó như: giúp deploy các service một cách dễ dàng vào hệ thống, tích hợp cơ chế nhiều instance cho service, chia tải cho các instance, cơ chế chịu lỗi cho các service trong kiến trúc Microservice, tối ưu performance cho hệ thống Platform... Để làm được điều đó thì kiến trúc platform của chúng em áp dụng những lý thuyết, giải pháp đã được đề cập ở chương II, áp dụng tất cả những điều đó để tạo nên một kiến trúc Platform như hiện tại.

## 3.2 Kiến trúc của Backend

Hiện tại kiến trúc tổng quan của hệ thống platform chúng em như sau



Hình 3.1: Các service trong hệ thống Platform

### 3.2.1 Ưu điểm

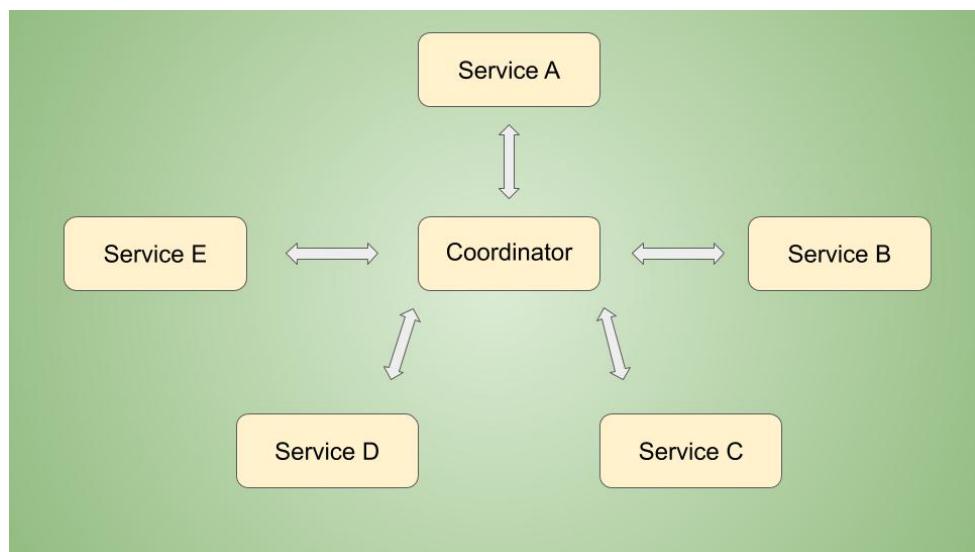
Với các khó khăn các lập trình viên thường gặp mà đã được đề cập ở Chương I, chúng em tạo ra một Platform nhằm khắc phục những vấn đề đó. Cơ bản, platform của chúng em cung cấp hai tính năng chính

Cung cấp công cụ deploy cho các lập trình viên: Các lập trình viên thường tập trung chuyên môn vào việc phát triển phần mềm và gặp khó khăn trong việc deploy các service của mình lên server, hoặc nếu được thì chỉ dùng đơn giản là service với một instance. Hệ thống của chúng em cung cấp khả năng deploy nhiều instance cùng với cơ chế load balancing các instance khi các yêu cầu vào hệ thống. Hơn thế nữa với K8s còn có thể cung cấp cơ chế tự động khôi phục lại các instance bị lỗi. Giúp các service của người sử dụng có thể đảm bảo chạy ổn định.

Cung cấp một Coordinator cho các service trong hệ thống: chịu trách nhiệm thực hiện kết nối đến các service, điều phối luồng cho hệ thống. Các

service không cần quan tâm mình kết nối với những server nào trong kiến trúc, Coordinator sẽ thực hiện nhiệm vụ đó, lấy kết quả và tiếp tục thực thi cho đến khi hoàn thành một transaction, giảm logic ở từng service.

Hơn nữa, do chúng em sử dụng kiến trúc Microservice, chia nhỏ kiến trúc ra thành nhiều service, mỗi service đảm bảo nhiệm vụ riêng và giao tiếp với nhau. Do đó chúng em đã cung cấp cơ chế bảo mật để tránh việc bị service ngoài tấn công, giả mạo ảnh hưởng đến người dùng hệ thống.



Hình 3.2: Coordinator quản lý điều phối đến các service

Tất cả các logic về việc giao tiếp giữa các service sẽ do Coordinator quản lý và điều phối, các service khác chỉ cần quan tâm đến logic của bản thân, trách gây phức tạp thêm cho từng service trong hệ thống. Hơn nữa, với Coordinator chúng em có thể dễ dàng thay tích hợp các service vào hệ thống, đổi thứ tự luồng đi của các service nếu có nhu cầu.

### 3.2.2 Nhuược điểm

Do các yêu cầu phải đi qua trung gian là Director Service nên *latency* cho một yêu cầu sẽ tốn thêm một ít thời gian để hoàn thành so với đi trực tiếp từ các service.

Hiện tại service Director cung cấp API để người dùng có thể đưa dữ liệu vào và hiện tại thì service tạo yêu cầu cho Director là Service User. Do đó người dùng cần deploy service mới để sử dụng hoặc dùng các công cụ hỗ trợ như BloomgRPC để có thể tạo yêu cầu gửi vào service Director.

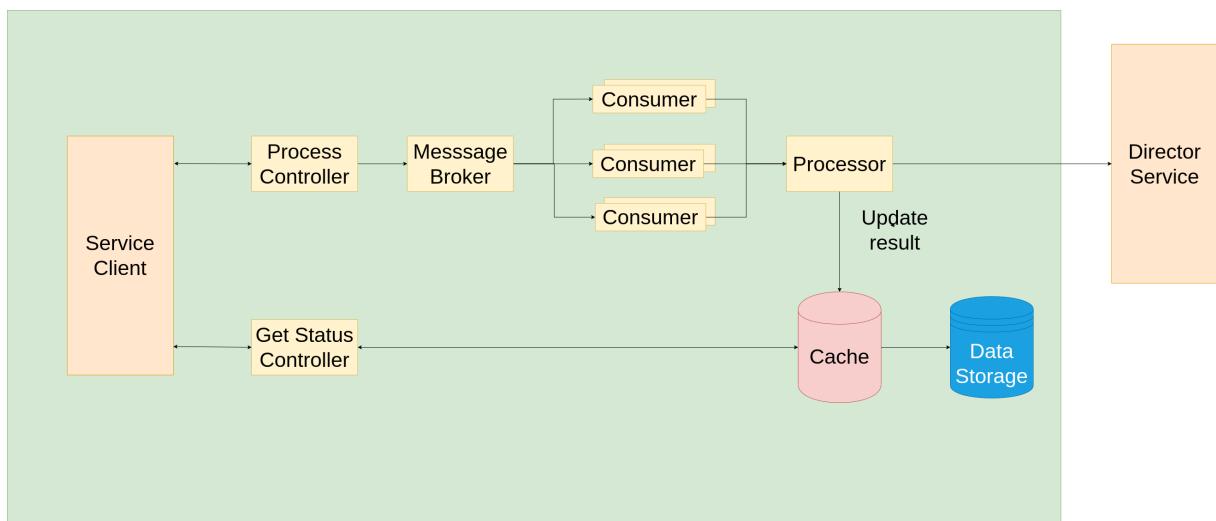
### 3.2.3 Tích hợp

- Để tích hợp vào Platform thì cần có một số thao tác
  - Phải triển khai thành hai API riêng biệt và xử lý theo cơ chế Asynchronous. Hai API lần lượt là *Process* và *GetStatus* để có thể tích hợp vào Gateway.
    - Process: API này có chức năng nhận các yêu cầu vào, sau đó trả về một ID cho người dùng cùng với trạng thái là đang xử lý. ID dùng để lấy trạng thái của quá trình xử lý và kết quả đi kèm nếu trạng thái thành công. Sau khi nhận yêu cầu thì sẽ chuyển vào thread khác để xử lý. Sau đó cập nhật kết quả cuối cùng vào bộ nhớ đệm.
    - GetStatus: API này có chức năng lấy trạng thái xử lý của các yêu cầu của người dùng. Dựa vào ID ở API Process, lấy dữ liệu từ bộ nhớ đệm ra và kiểm tra trạng thái của tiến trình xử lý và trả về cho người dùng.
  - Phải chỉnh sửa cấu hình các thứ tự service trên file cấu hình của Service Director và cấu trúc dữ liệu định nghĩa trong Service Director. Sau đó Build lại Image, push lên docker Hub. Ở server director cần pull để lấy image mới nhất về từ docker Registry và rerun lại Image.

### 3.3 Service User

Service User cung cấp một số API *CreateTask* cho người sử dụng platform để có thể đưa các dữ liệu, yêu cầu vào hệ thống platform, từ đây service sẽ ghi nhận bắt đầu một transaction, cập nhật trạng thái cho yêu cầu này và trả về trạng thái đang xử lý cho người dùng, lưu lại trạng thái này vào bộ nhớ đệm, để tiện dụng cho việc cập nhật trạng thái với tốc độ xử lý nhanh thay vì lưu vào trong database. Trong khi đó, Service User sẽ gửi yêu cầu đến Director để thực hiện yêu cầu của user.

Ngoài ra, còn đón nhận các tin nhắn được gửi về từ Service Director để cập nhật trạng thái cho transaction đã xử lý đến service nào trong hệ thống, giúp cho người dùng có thể theo dõi trạng thái, phát hiện lỗi cụ thể ở service nào, để user có thể biết được là trạng thái của giao dịch. Để lấy được trạng thái thì User sử dụng API *GetStatus* dựa trên ID mà API *CreateTask* đã trả về trước đó.



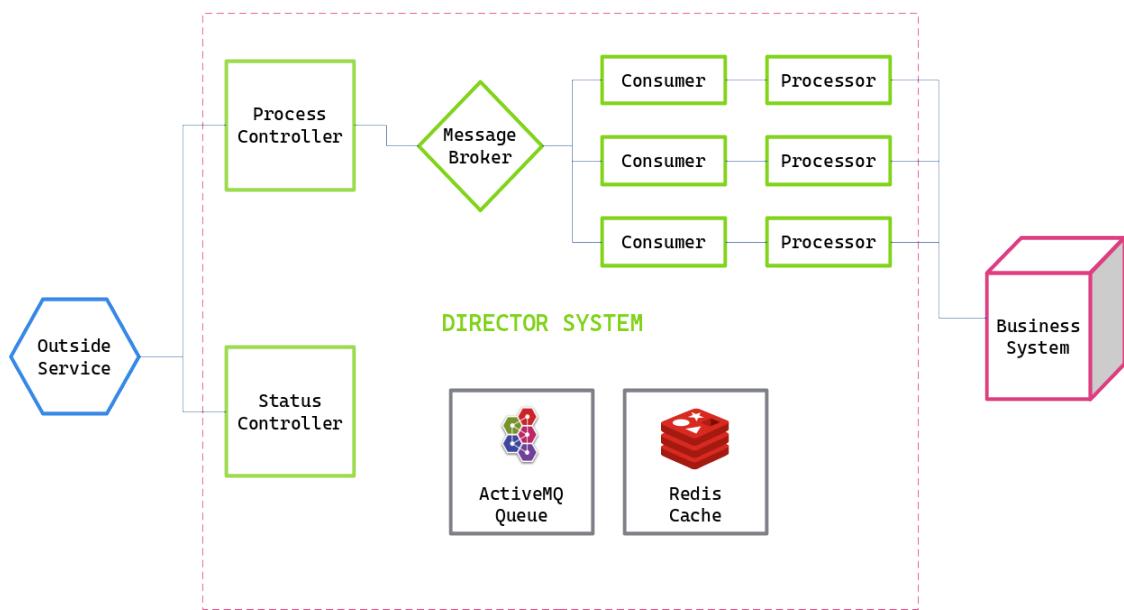
Hình 3.3: Component system của User Service

Sau khi đã có kết quả cuối cùng nhận được từ Director Service, Service User sẽ cập nhật trạng thái là lưu vào trong database MySQL. Lưu các trạng thái cuối vào database để tránh việc mất dữ liệu khi lưu trên bộ nhớ đệm khi dữ liệu hết hạn sẽ bị xóa đi hoặc xảy ra sự cố dẫn đến Redis bị

dừng. Các dữ liệu được lưu dưới database được dùng cho API *GetAllTask* để lấy tất cả các transaction mà user đã thực hiện.

Ở Service User tổ chức các API theo phương thức Asynchronous kết hợp activeMQ, Redis để tăng throughput và latency cho Service. Giao thức để kết nối với service Director thì sử dụng gRPC để tăng tối đa hiệu năng.

### 3.4 Kiến trúc hệ thống Director



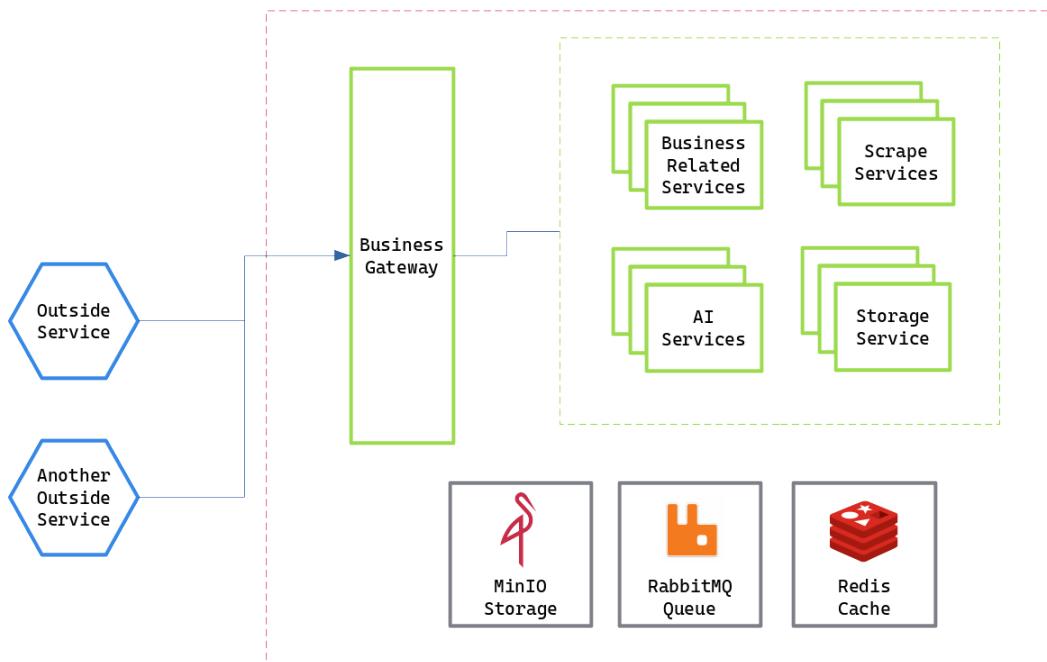
Hình 3.4: Tổng quan kiến trúc hệ thống Director

Service Director cung cấp API process nhận danh sách các service cần xử lý. Sau đó Director sẽ chịu trách nhiệm xử lý yêu cầu đến từng service trong danh sách thông qua gửi yêu cầu đến service Gateway. Với mỗi lần gọi Gateway, Director sẽ có một ID để dựa vào đó sử dụng API *emphGetStatus* xem trạng của tiến trình đó như thế nào. Nếu hoàn thành tiến trình, cập nhật trạng thái tại service đó của yêu cầu, gửi callback về cho User Service

để người sử dụng có thể theo dõi trạng thái yêu cầu đã đi đến đâu, trạng thái xử lý như thế nào. Tiếp tục lấy kết quả của việc xử lý cùng với danh sách các service còn lại và tiếp tục xử lý như vậy cho đến khi hoàn thành phần tử cuối cùng trong danh sách.

Service Director có vai trò quan trọng trong hệ thống, chịu trách nhiệm quản lý trạng thái, điều khiển luồng đi của hệ thống. Nhờ vào Director mà Platform có thể linh động, dễ dàng thay đổi để phù hợp với từng nhu cầu khác nhau khi tích hợp vào hệ thống, dễ dàng sử dụng thay đổi thông qua file cấu hình của service.

### 3.5 Kiến trúc hệ thống Business



Hình 3.5: Tổng quan kiến trúc hệ thống Business

Hệ thống Business là lõi chính của hệ thống AI Platform. Mọi xử lý logic chính đều xảy ra ở hệ thống Business.

Hệ thống bao gồm:

- Business gateway là service nhận yêu cầu từ những service bên ngoài hệ thống gọi tới.
- Các AI service tận dụng sức mạnh của AI để xử lý ảnh, xử lý video, xử lý dữ liệu,...
- Các scrape service lấy dữ liệu từ các trang web.
- Storage service xử lý việc tải dữ liệu lên và tải dữ liệu xuống.
- MinIO storage là nơi lưu trữ toàn bộ dữ liệu, hoạt động giống các trang lưu trữ dữ liệu Google Drive hay Dropbox.
- RabbitMQ Queue là nơi trung gian chuyển giao công việc giữa các service.
- Redis Cache là nơi lưu trữ trạng thái từng công việc của các service.

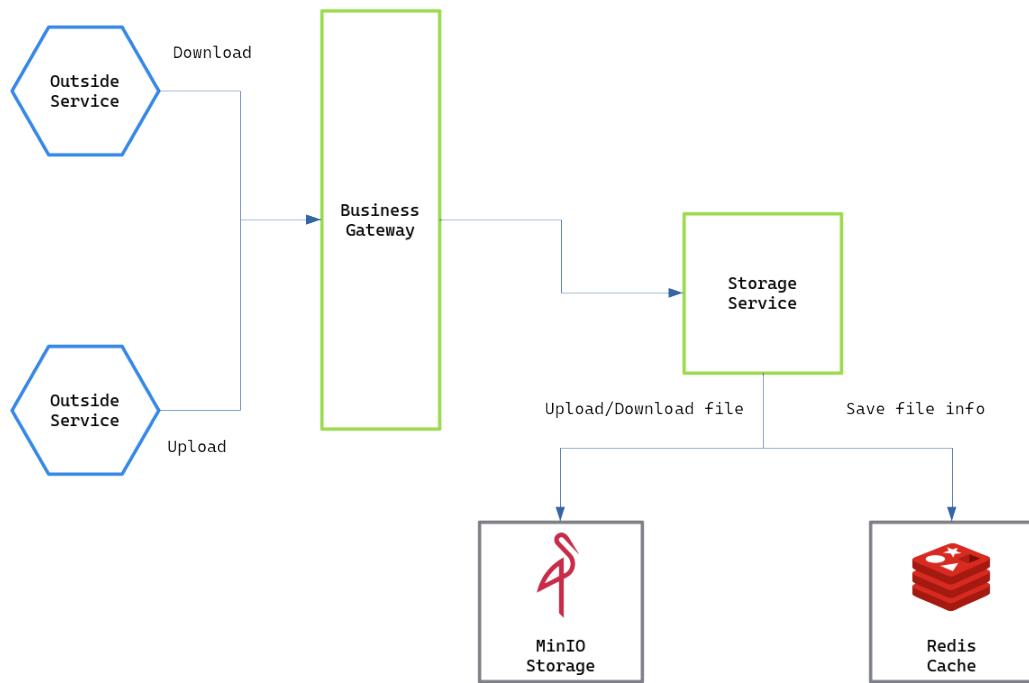
### 3.5.1 Business gateway

Business gateway thể hiện vai trò của mình là hứng toàn bộ yêu cầu của các service bên ngoài gửi lên. Service bên ngoài có thể là service của AI platform, nhưng nằm ngoài hệ thống Business, cũng có thể là service của một bên thứ 3.

Lý do chính chúng em sử dụng Business gateway là để bảo vệ toàn bộ các AI service, Scape service, và các service khác đứng đằng sau nó. Các service này cần được bảo vệ bởi vì chúng không có cơ chế tự bảo vệ nào cả. Chúng đơn giản chỉ là những service xử lý các yêu cầu được gửi lên, các service này không cần thiết phải quan tâm đến việc bảo mật, xác thực tài khoản, danh tính của người tạo yêu cầu. Các service cũng không nên quan tâm đến việc hạn chế số lần nhận yêu cầu để tránh tấn công DDOS. Tất cả nên để Business gateway xử lý, service làm tốt nhiệm vụ của chính nó là được.

Business gateway nhận xử lý yêu cầu có dạng REST API và kể cả gRPC. Điều này dễ dàng cho các service bên ngoài, vì service bên ngoài có thể là web, chỉ tạo được yêu cầu REST API. Hoặc nếu service bên ngoài có thể cài đặt gRPC để tăng tốc độ gửi nhận yêu cầu, thì Business gateway đều xử lý được.

### 3.5.2 Storage service



Hình 3.6: Kiến trúc storage service

Storage service nằm sau Business gateway, các service bên ngoài muốn truy cập đến Storage service phải thông qua Business gateway.

Storage service có 2 nhiệm vụ chính:

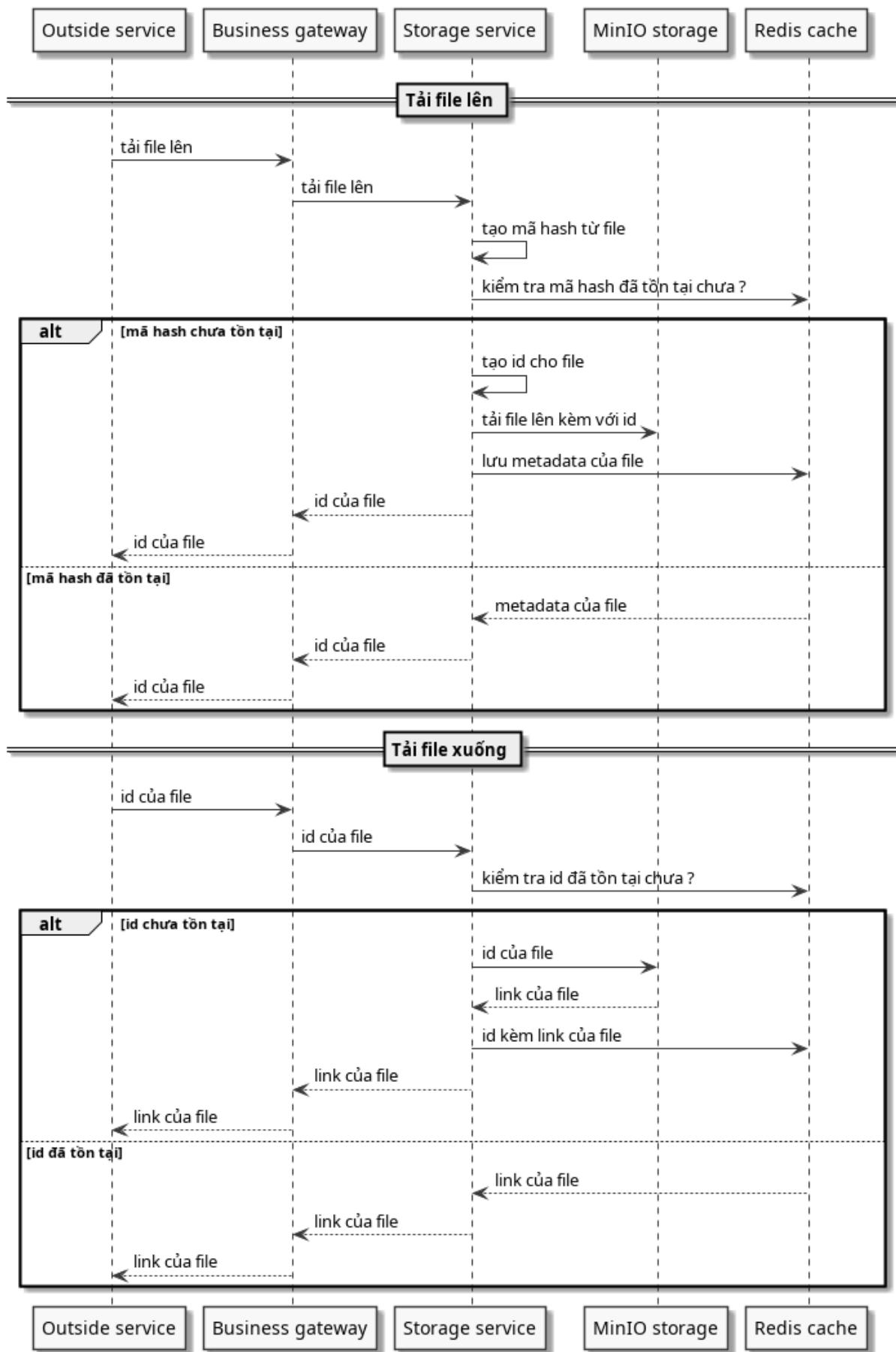
- Xử lý file được tải lên từ service bên ngoài và lưu nó vào MinIO storage.
- Trả về link để service bên ngoài tải file về.

Storage service lợi dụng sức mạnh của MinIO storage. Mỗi file mà service ngoài tải lên sẽ được lưu trữ an toàn bên trong MinIO storage.

Trước khi tải file vào trong MinIO storage, Storage service làm nhiệm vụ tiền xử lý. Bởi vì bất cứ kho lưu trữ nào dù lớn đến đâu cũng là có hạn, cho nên Storage service sẽ kiểm tra liệu file này đã có trong MinIO storage chưa, nếu chưa thì thực sự thêm vào MinIO storage. Còn nếu file này đã tồn tại rồi thì Storage service chỉ trả về id của file đã có trong MinIO storage. Điều này giảm tải rất nhiều cho hệ thống vì Storage service sẽ bớt việc lúc nào cũng phải tải file lên MinIO storage.

Để kiểm tra file tải lên đã tồn tại hay chưa trong MinIO storage, chúng em sử dụng Redis cache. Chúng em lưu toàn bộ thông tin về file lên Redis cache mỗi lần Storage service xử lý chúng. Có thể nói chúng em sử dụng Redis cache để lưu dữ liệu của dữ liệu, hay còn gọi là siêu dữ liệu (metadata). Siêu dữ liệu này bao gồm mã hash của file, kích thước của file, định dạng của file và id của file được lưu trữ ở trong MinIO storage.

Chúng em sử dụng thuật toán SHA256 để tạo ra mã hash của file, lý do chọn thuật toán này là vì SHA256 là thuật toán hash bảo mật, và không thể bị đụng độ với những công nghệ hiện tại. Độn độ nghĩa là 2 file có cùng chung mã hash, thì lúc này sẽ dẫn đến lỗi không thể tải file lên. Ví dụ file A và file B có chung mã hash, và file A đã có sẵn trong MinIO storage, thì khi file B được tải lên, Storage service sẽ kiểm tra mã hash của file B trong Redis cache. Vì file A đã tồn tại cho nên mã hash của nó cũng phải tồn tại trong Redis cache. Cho nên Storage service thấy mã hash của file B, cũng chính là mã hash của file A ở trong Redis cache, nên trả về id lấy được trong Redis cache. Nhưng thực chất id này là của file A chứ không phải file B.



Khi các service bên ngoài muốn lấy link để tải file về, dùng đúng id mà Storage service gửi về khi tải file lên. Bởi vì mỗi id chỉ gắn với duy nhất một file theo quan hệ 1-1. Lý do chúng em sử dụng id thay cho link dạng url là vì, dùng id trên cùng một hệ thống chúng em sẽ tiện để theo dõi hơn, vì mỗi id được chúng em tạo ra là duy nhất. Một lý do khác là link dạng url sẽ hết hạn theo thời gian, để tránh việc sử dụng link tải về liên tục gây spam và làm nghẽn hệ thống. Với việc sử dụng id để định danh file, mỗi lần gửi lên hệ thống để lấy link tải về, hệ thống sẽ kiểm tra link dạng url đã lưu trong Redis cache có hết hạn hay chưa, nếu chưa hết hạn thì trả về link này. Nếu đã hết hạn, thì mới gọi qua MinIO storage để lấy link mới. Điều này cũng nhằm hạn chế việc gọi qua MinIO storage. Tại vì hệ thống khi thực hiện yêu cầu gọi qua Redis cache thì ít tốn tài nguyên hơn là việc gọi qua MinIO storage.

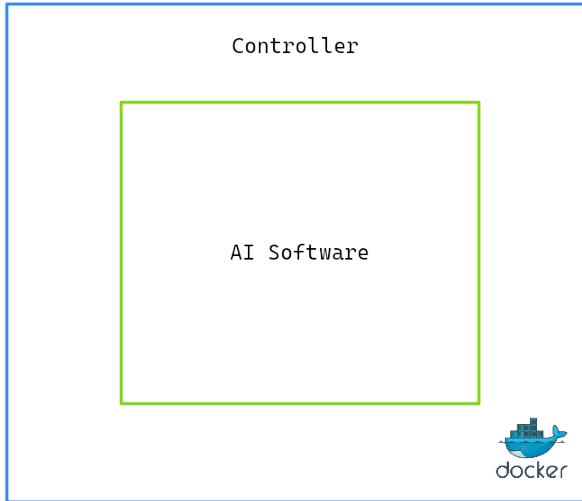
### 3.5.3 Thiết kế đóng gói cho AI service

Vấn đề chúng em gặp phải đầu tiên khi đóng gói các phần mềm sử dụng AI trong AI service, cũng như là đóng gói bất cứ phần mềm bằng công nghệ Docker container, đó là mỗi phần mềm đều đa dạng về loại dữ liệu nhập và loại dữ liệu xuất. Ví dụ một phần mềm có dữ liệu nhập là hình ảnh và dữ liệu xuất là file dạng chữ, một phần mềm khác lại có dữ liệu nhập và dữ liệu xuất là hình ảnh chẳng hạn.

Chúng em đề xuất giải pháp là sử dụng Docker container gói 2 lần phần mềm. Lần thứ nhất đóng gói phần mềm lại thành một Docker image với đầy đủ để thư viện để phần mềm có thể chạy được trên mọi nền tảng mà Docker container hỗ trợ như Windows, Linux, Mac. Lớp đóng gói 2 chúng em vừa gói lại lớp đóng gói 1, và chúng em viết thêm Controller dùng để điều khiển phần mềm nằm trong lớp đóng gói 1.

Việc tiền xử lý hay hậu xử lý dữ liệu đầu vào và đầu ra của AI service sẽ được thực hiện các service khác. Lợi ích là để không lặp đi lặp lại công việc, vì có thể 2 AI service khác nhau nhưng đều giống qua ở chỗ hậu xử

lý đối với dữ liệu đầu ra. Thì việc để code xử lý ở trong cả 2 AI service đều không lợi ích bằng việc tách riêng ra để 1 service thực hiện.

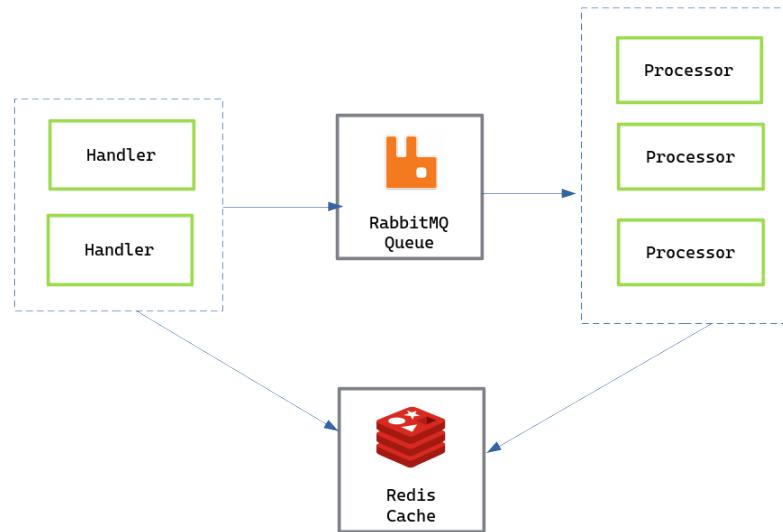


Hình 3.8: Mô tả việc đóng gói của service AI

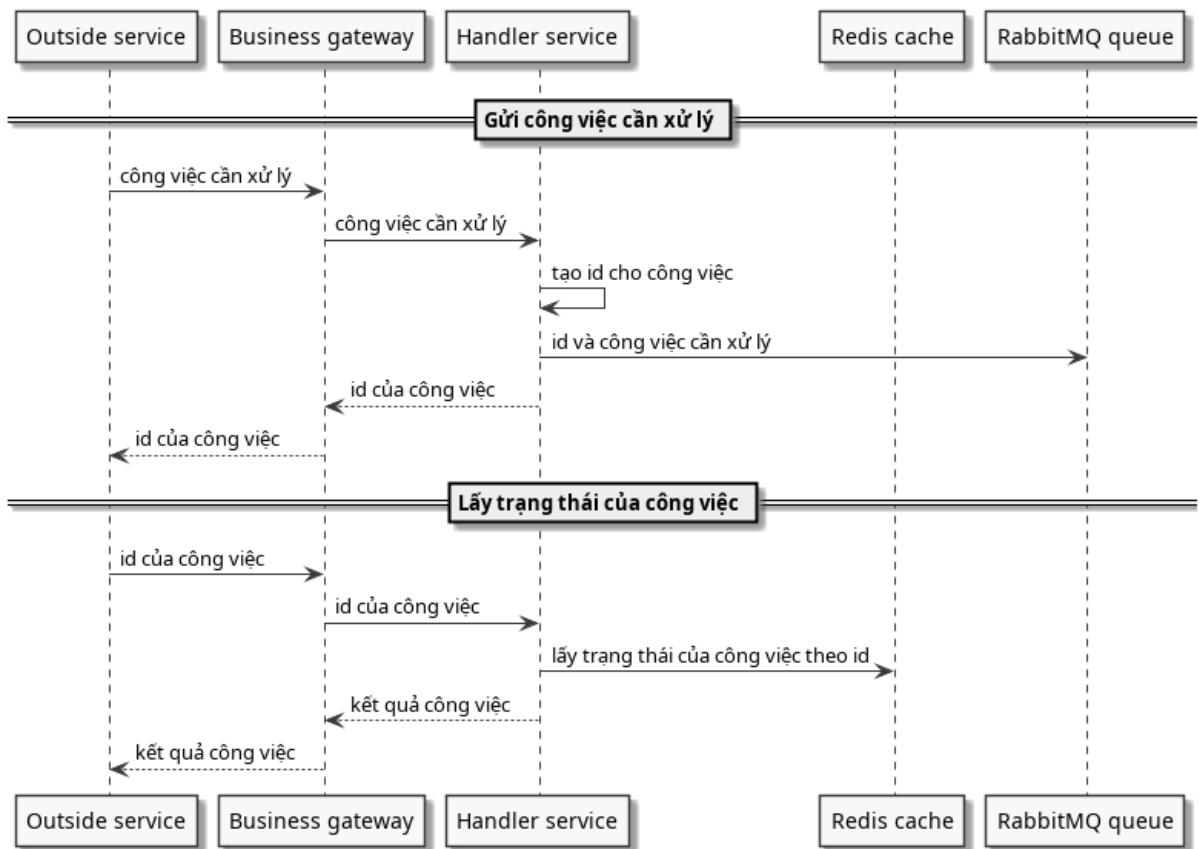
Có một vấn đề khác, là thời gian mỗi lần phần mềm AI xử lý là không nhỏ, có thể từ vài giây cho đến cả phút. Nếu thiết kế hợp lý, khi AI service nhận một yêu cầu và xử lý nó hơi lâu, thì các yêu cầu khác đều không được xử lý và có thể bị mất đi. Cho nên chúng em đề xuất thiết kế kiểu chia tách công việc, sẽ có service riêng nhận yêu cầu, và service khác xử lý yêu cầu. Điều này đảm bảo yêu cầu không bị thất lạc, cũng như không chặn việc nhận yêu cầu mới trong khi đang xử lý yêu cầu cũ.

Hình 3.9 thể hiện kiến trúc của AI service mà chúng em đề xuất và cài đặt. Handler là nơi hứng toàn bộ yêu cầu xử lý, sau đó những yêu cầu này sẽ được chuyển vào hàng đợi trong RabbitMQ queue. Processor là nơi xử lý toàn bộ yêu cầu, lấy yêu cầu từ hàng đợi RabbitMQ queue, sau đó xử lý và lưu kết quả vào Redis cache. Sau đó Handler sẽ lấy kết quả xử lý xong từ Redis cache để trả về cho yêu cầu.

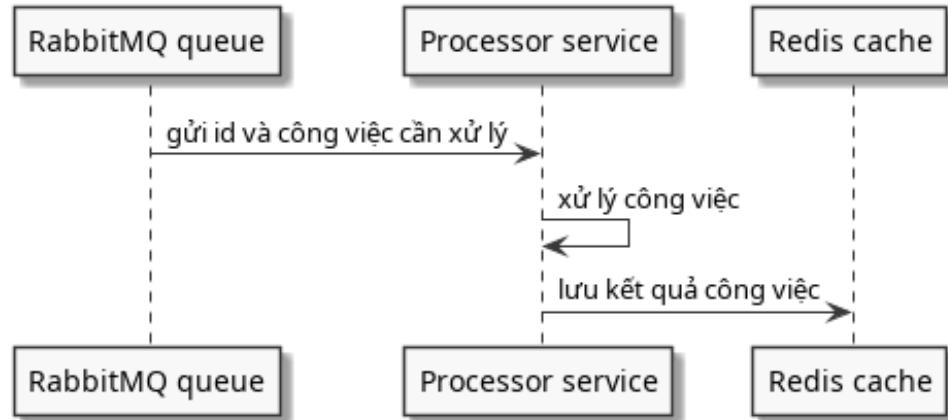
Lý do về việc Handler không gọi trực tiếp Processor mà phải gọi trung gian qua RabbitMQ queue là để tách sự phụ thuộc giữa Handler và Processor. Cả 2 hoạt động mà không cần biết đến sự tồn tại cũng như trạng thái của nhau. Trong mắt Handler chỉ có RabbitMQ queue, Redis cache, và trong mắt Processor cũng vậy. Lợi ích của việc này là scale dễ dàng và không bị ràng buộc giữa Handler và Processor, ví dụ có 2 Handler và 4 Processor chạy cùng lúc song song.



Hình 3.9: Kiến trúc của AI Service



Hình 3.10: Luồng xử lý của Handler



Hình 3.11: Luồng xử lý của Processor

# Chương 4

## Một số kịch bản xây dựng

*Nội dung chương 4 trình bày một số kịch bản có thể ứng dụng kiến trúc hệ thống AI Platform của chúng em.*

Có vô vàn kịch bản để ứng dụng kiến trúc AI Platform của chúng em. Lý do đơn giản là vì chúng em thiết kế hệ thống theo hướng pipeline của hệ điều hành UNIX. Mỗi service có một công việc riêng và làm tốt công việc của chính nó. Tùy theo cách tổ hợp các service mà sẽ tạo ra một kịch bản mới.

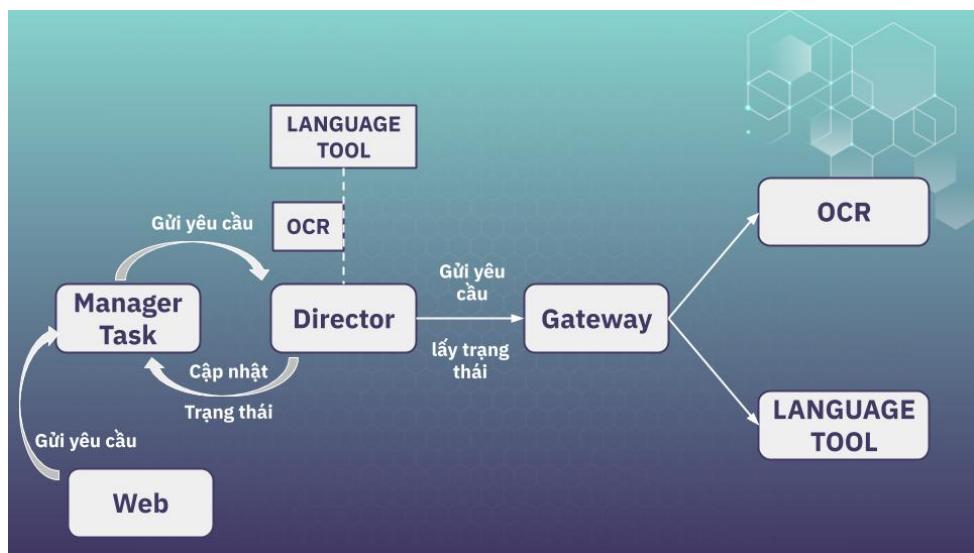
### 4.1 Kiểm tra chính tả sau khi nhận dạng chữ từ ảnh

Có rất nhiều phần mềm để lấy chữ có trong ảnh, gọi là loại phần nhận dạng ký tự quang học (OCR). Phần mềm OCR có thể được tận dụng vào rất nhiều việc hữu ích, ví dụ như sao lưu dữ liệu giấy. Vì ở những công ty lâu năm, tài liệu dạng giấy tờ rất là nhiều và có thể nói là hầy hết dữ liệu ở công ty này đều được lưu dưới dạng giấy tờ. Có thể là đánh máy in ra hoặc là chữ viết tay. Với sự trợ giúp của phần mềm OCR, các tài liệu dạng giấy này có thể được quét và lưu trữ bằng file trên máy tính.

Sau khi nhận dạng chữ từ ảnh, kết quả có được là file dữ liệu dạng chữ. Với kết quả này, chúng em có thể sử dụng các phần mềm AI để xử lý thêm

với dữ liệu chữ hiện có. Chúng em chọn phần mềm kiểm tra chính tả để kiểm tra trên dữ liệu chữ lấy được từ phần mềm OCR.

Bên ngoài chúng em xây dựng hệ thống Frontend để người dùng giao tiếp ở mức độ cơ bản, như tải file ảnh lên, xem quá trình xử lý và tải file kết quả kiểm tra chính tả về. Trên giao diện này người dùng có thể theo dõi quá trình xử lý đã đi qua các service nào của hệ thống, để có thể biết chi tiết được yêu cầu đã được xử lý đến đâu. Nếu thất bại có thể biết được service nào, từ đó có thể xem và biết được vấn đề sớm nhất có thể, giúp cho hệ thống được ổn định và không rơi vào trạng thái UNAVAILABLE.

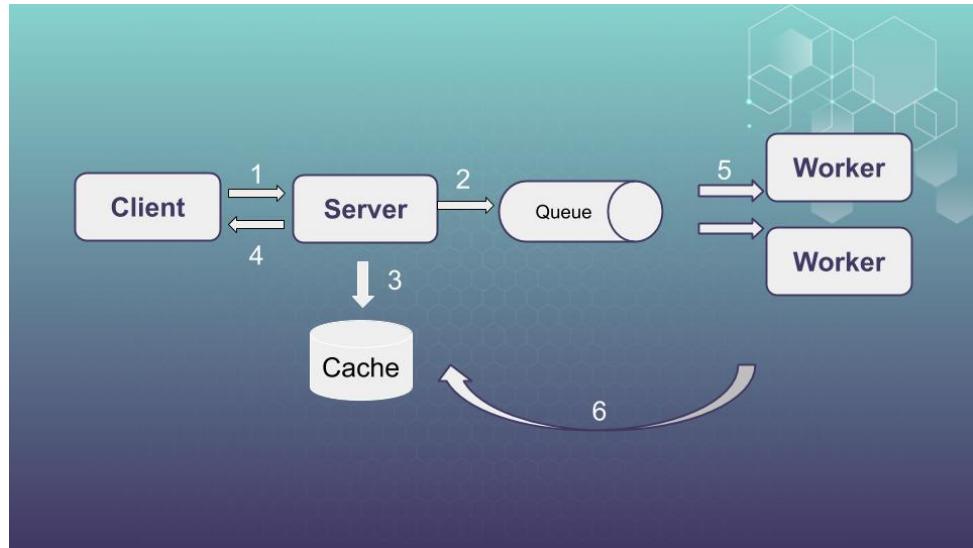


Hình 4.1: Sơ đồ hệ thống với OCR và Language tool

Khi có một yêu cầu đến Management Task, thì service này sẽ tạo ra một transaction với ID được tự động tạo ra và trạng thái cho transaction đó là đang xử lý. Trong khi đó Controller sẽ gửi transaction vào queue, lưu vào cache để tăng tốc độ xử lý cho hệ thống khi web gửi cái yêu cầu lấy trạng thái xử lý hiện tại liên tục sau một khoảng thời gian (polling). Ở trường hợp này chúng em sử dụng công nghệ Redis đã được trình bày ở chương II làm bộ nhớ đệm để tăng tốc độ xử lý cho hệ thống.

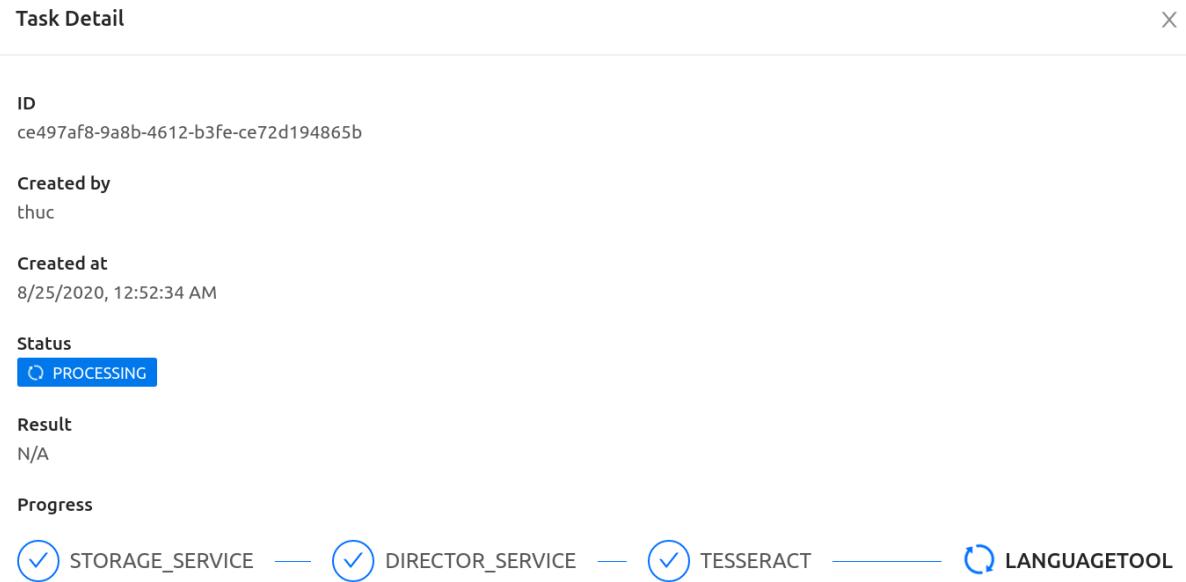
Bên cạnh đó chúng em sử dụng cơ chế xử lý Asynchronous kết hợp với Kafka và Redis để thực hiện điều này. Sử dụng Redis như một cơ chế caching lại trạng thái xử lý của transaction trên hệ thống. Sử dụng Kafka

để đẩy các transaction có trạng thái là Processing vào hàng đợi để chờ xử lý.



Hình 4.2: Cơ chế Asynchronous cho Task manager service

Ngoài ra, trong hệ thống này, Kafka còn được dùng để giao tiếp giữa Director Service và Task Management service, Với mỗi lần đi xử lý đến Service cụ thể, Director sẽ bắn lần lượt các tin nhắn đến Task Management để cập nhật trạng thái hiện tại trên service đó, Ở Task Management mỗi lần như vậy cần lấy trạng thái được lưu trữ ra và cập nhật với thông tin vừa được nhận từ Director thông qua Kafka và lưu vào bộ nhớ. Việc phải cập nhật liên tục, lấy từ bộ nhớ và lưu như vậy cần lựa chọn cơ sở dữ liệu có tốc độ cao, vì vậy chúng em sử dụng Redis cho trường hợp này. Nhưng đợi đến khi Task được xử lý hoàn thành (có thể ở trạng thái thành công hoặc thất bại) để bảo đảm cho việc không mất dữ liệu (Vì Redis lưu trên Ram), chúng em sẽ lưu vào database mysql để đảm bảo rằng không mất thông tin của transaction đó khi người dùng yêu cầu.



Hình 4.3: Hình ảnh mô tả quá trình xử lý trên từng service

Ta thấy rằng trên hình là danh sách các service được thực thi, TESSERACT là tên khác của service ORC và LANGUAGETOOL dùng để kiểm tra lỗi chính tả của dữ liệu text mà OCR vừa tách ra được. Trên web thể hiện luồng đi theo thứ tự của các service mà người sử dụng đã cấu hình. Web cập nhật trạng thái bằng cơ chế polling, sau mỗi khoảng thời gian, sẽ gọi API lấy trạng thái tương ứng với ID được thể hiện trên màn hình, đến khi nào hoàn thành, cập nhật trạng thái trên giao diện và ngừng lại.

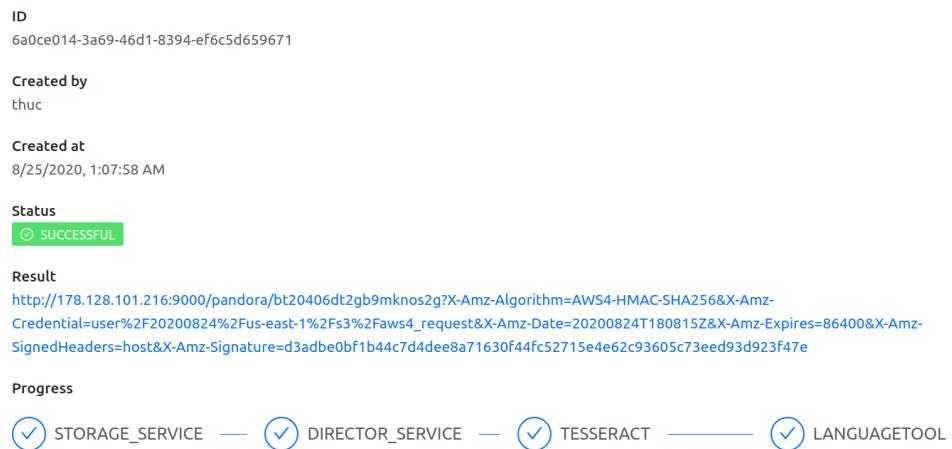
Kết quả của quá trình xử lý là một URL, trong đó thể hiện thông tin ngữ pháp sai được phát hiện cho người sử dụng. Chúng em sẽ lấy một ví dụ cụ thể như sau, dưới đây là một tấm hình có một dòng chữ đúng và sai, chúng em sẽ thử đưa chúng vào hệ thống xử lý

**THSI IS INCORCT WODR**

**THIS IS CORRECT WORD**

Hình 4.4: Hình ảnh cho việc demo trên hệ thống

Sau khi xử lý và chờ đợi đến khi có kết quả cuối cùng thì trên thuộc tính result sẽ hiện ra một đường link chứa thông tin kết quả của quá trình xử lý.



Hình 4.5: Kết quả xử lý trên giao diện người sử dụng

Với đường dẫn này, khi người dùng nhấn vào thì sẽ hiện giao diện một web browser để mô tả chi tiết kết quả của quá trình xử lý.

```

1.) Line 1, column 1, Rule ID: MORFOLOGIK_RULE_EN_US prio=-10
Message: Possible spelling mistake found.
Suggestion: Thai; AHSI; IHSI; MHSI; THS; TSI; TH SI; THS I
THSI IS INCORCT WODR THIS IS CORRECT WORD
^^^^^

2.) Line 1, column 9, Rule ID: MORFOLOGIK_RULE_EN_US prio=-10
Message: Possible spelling mistake found.
Suggestion: Incorrect; Infarct
THSI IS INCORCT WODR THIS IS CORRECT WORD
^^^^^^^

3.) Line 1, column 17, Rule ID: MORFOLOGIK_RULE_EN_US prio=-10
Message: Possible spelling mistake found.
Suggestion: ODR; W ODR
THSI IS INCORCT WODR THIS IS CORRECT WORD
^^^

```

Hình 4.6: Kết quả xử lý của ảnh 4.4

Trên kết quả sẽ mô tả chi tiết dòng nào đang gặp lỗi và các từ được gợi ý cho việc sửa lỗi đó. Với kịch bản được trình bày ở phần này, chúng em không chú trọng vào tỉ lệ phân tích cú pháp của Service Language mà chú trọng vào việc tích hợp hai service OCR và LanguageTool vào hệ thống và các công nghệ được sử dụng đã được đề cập trong chương II. Tiếp theo chúng em xin tiếp tục trình bày thêm một số kịch bản tích hợp vào hệ thống.

## 4.2 Kiểm tra mật độ xe của giao thông thành phố Hồ Chí Minh

Vấn đề giao thông đô thị đang là vấn đề trọng điểm của các đô thị. Các đô thị càng phát triển nhanh thì dân số càng trở nên đông đúc, kéo theo số lượng phương tiện tham gia giao thông tăng lên. Nhưng tốc độ xây dựng hạ tầng của đô thị không nhanh bằng tốc độ tăng dần của số lượng các phương tiện tham gia giao thông. Hệ quả kéo theo tất nhiên là kẹt xe, các tuyến đường vốn chỉ thiết kế cho một lưu lượng xe nhất định, bây giờ phải thêm một lưu lượng xe gấp 2 thậm chí gấp 3 lần lưu lượng thì tuyến đường trở nên đông đúc và có thể tắc đường.



Hình 4.7: Kẹt xe ở một đoạn đường tại TP.HCM

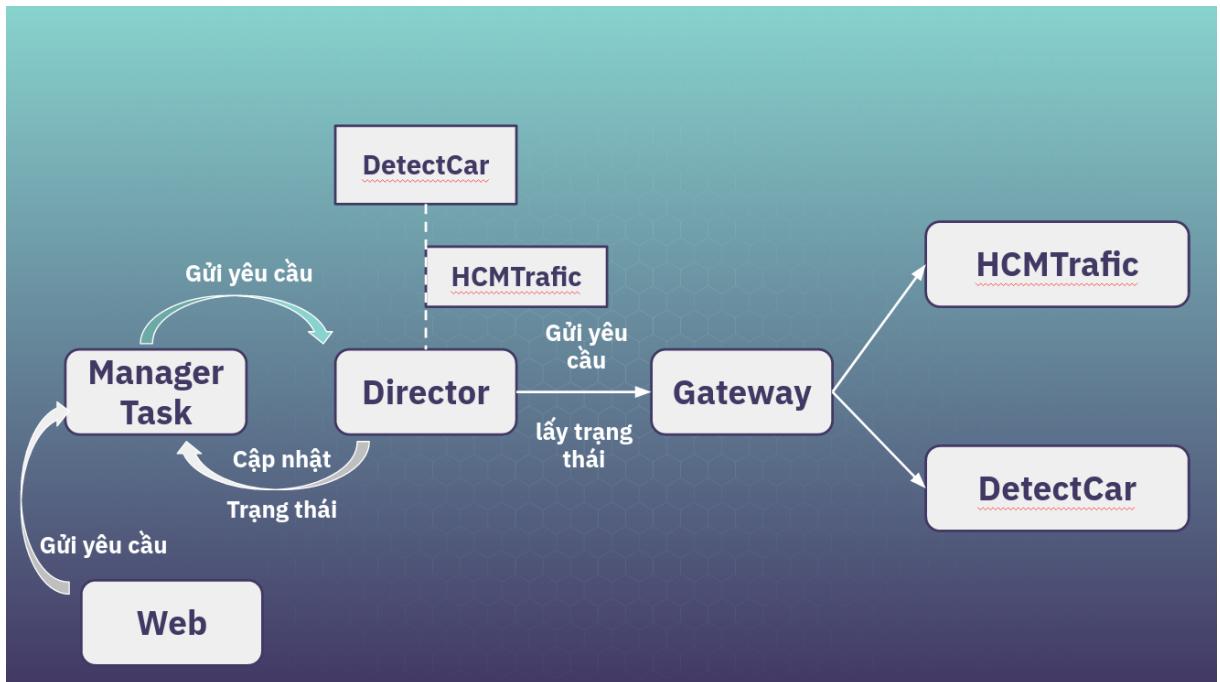
Cổng thông tin giao thông của Thành phố Hồ Chí Minh có cung cấp cho mọi người hình ảnh và video từ camera của một số tuyến đường trên website <http://giaothonh.hochiminhcity.gov.vn/>. Chúng em dựa vào thông tin lấy được để xây dựng một hệ thống nhận diện và đếm số lượng các phương tiện giao thông tại được chụp trên camera.

Hệ thống chúng em đề xuất thực chất chia làm 4 hệ thống có chung kiến trúc. Kiến trúc này là kiến trúc chúng em đề xuất ở chương trước. 4 hệ thống này chỉ khác ở chỗ chúng em tráo service xử lý.

4 hệ thống bao gồm:

- Nhận diện xe hơi tham gia giao thông.
- Đếm số lượng xe hơi tham gia giao thông.
- Nhận diện người tham gia giao thông.
- Đếm số lượng người tham gia giao thông.

Cả 4 hệ thống này ngoài kiến trúc chung ra, còn tích hợp chung service HCMTraffic dùng để lấy dữ liệu ảnh từ camera của giao thông Thành phố Hồ Chí Minh.



Hình 4.8: Hệ thống nhận diện xe hơi tham gia giao thông

Khi nhận được dữ liệu đầu vào là địa chỉ của khu vực cần nhận diện, đầu tiên hệ thống sẽ lấy hình ảnh từ camera của giao thông thành phố, sau đó sẽ xử lý ảnh và đưa ra kết quả là ảnh đã nhận diện được xe hơi.

Service HCMTraffic chúng em cài đặt đơn giản bằng Python và đóng gói bằng Docker với mục đích chỉ lấy ảnh từ camera của các tuyến đường mà giao thông thành phố cho phép. Service này hoạt động như là một scrapper, thay vì để người dùng vào website của cổng thông tin giao thông thành phố rồi chụp màn hình lại thì chúng em làm demo service này để việc lấy ảnh từ camera trở nên dễ dàng hơn.

Sau khi lấy được dữ liệu hình ảnh từ tuyến đường đã chọn, tiếp theo Service Director có nhiệm vụ tìm thứ tự Service tiếp theo cần được xử lý, cùng với hình ảnh đã được lấy phía trên để gửi đến Gateway bắt đầu phân tích hình ảnh.



Hình 4.9: Service HCMTraffic

Vì đóng gói bằng Docker nên service HCMTraffic chạy trên mọi nền tảng như Linux, Windows, macOS. Phần cài đặt chúng em sử dụng ngôn ngữ Python vì tính đơn giản và có thể chạy như một script, dễ dàng đóng gói. Trong container này chứa tất cả những thư viện và tập tin cần thiết để chạy được một cách độc lập trên các nền tảng khác nhau, giúp ngăn chặn tình trạng chạy được trên máy vật lý này nhưng khi chuyển sang máy vật lý khác thì lại lỗi.

```

1:yoshie@archlinux: ~/go/src/hcmtraffic 
~/go/src/hcmtraffic master
> cat input.txt
nguyen_van_cu_tran_hung_dao_2
~/go/src/hcmtraffic master
> python main.py input.txt output
output output.jpg
url http://giaothong.hochiminhcity.gov.vn:8007/Render/CameraHandler.ashx?id=5b0b
7bbe0e517b00119fd806
~/go/src/hcmtraffic master
>

```

Hình 4.10: Chạy service HCMTraffic trên terminal

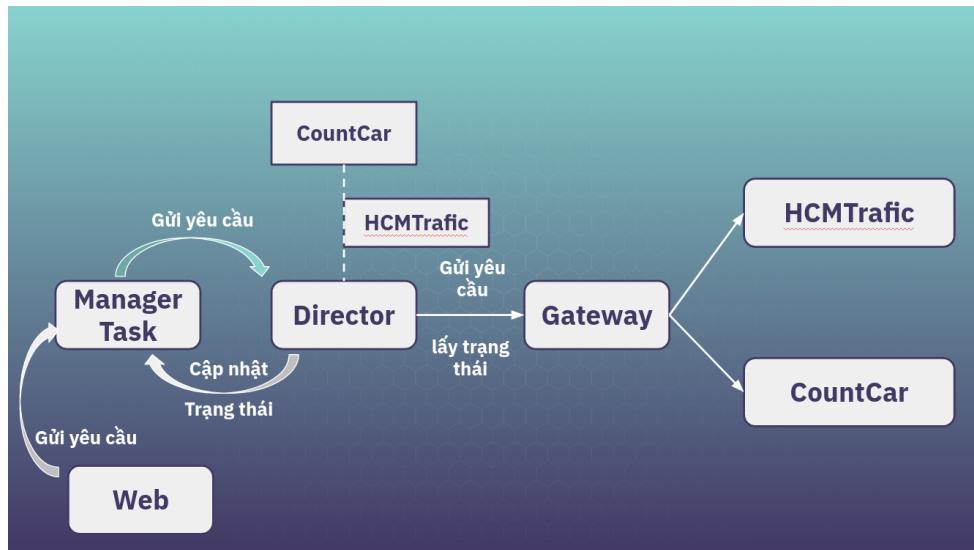


Hình 4.11: Kết quả đầu ra của service HCMTraffic

Sau khi có được kết quả từ service HCMTraffic, chúng em đưa vào làm dữ liệu vào của service DetectCar. Service DetectCar chúng em đóng gói lại thư viện CVLib của Python về nhận diện hình ảnh bằng Docker.

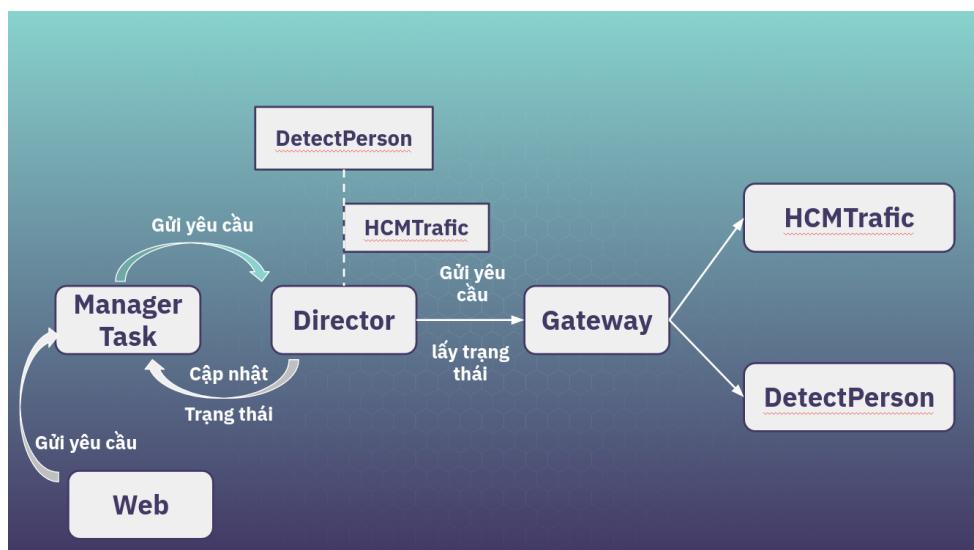
Đối với 3 hệ thống còn lại, chúng em chỉ cần thay đổi service DetectCar, thành các service CountCar, DetectPerson, CountPerson là sẽ có các hệ thống mới với mục đích khác nhau, nhưng cùng kiến trúc và chỉ thay đổi rất nhỏ ở bên dưới.

Hình 4.12 mô tả kiến trúc của hệ thống đếm số lượng xe tham gia giao thông. Kiến trúc của hệ thống này chỉ khác kiến trúc 4.8 ở service xử lý CountCar.



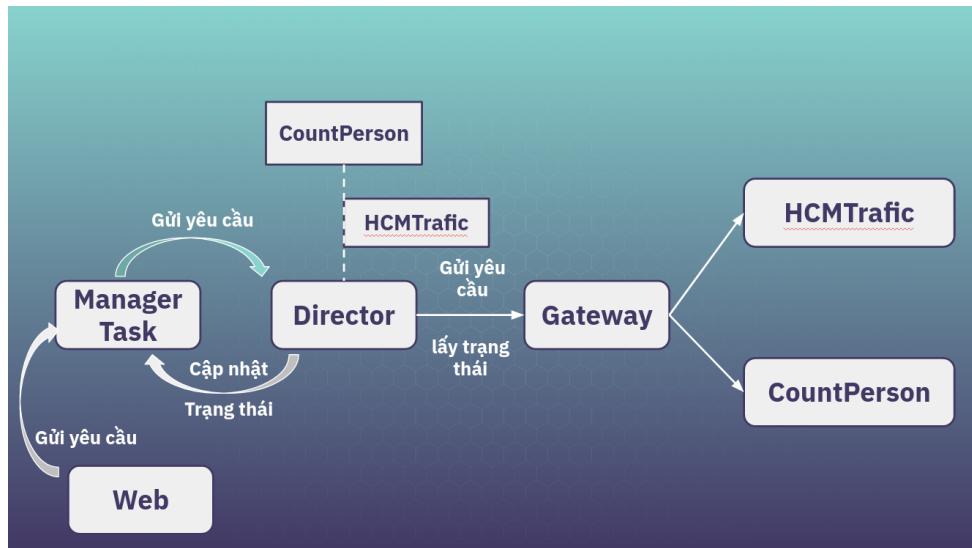
Hình 4.12: Hệ thống đếm số lượng xe hơi tham gia giao thông

Hình 4.13 mô tả kiến trúc của hệ thống nhận diện người tham gia giao thông. Kiến trúc của hệ thống này chỉ khác kiến trúc 4.8 ở service xử lý DetectPerson.



Hình 4.13: Hệ thống nhận diện người tham gia giao thông

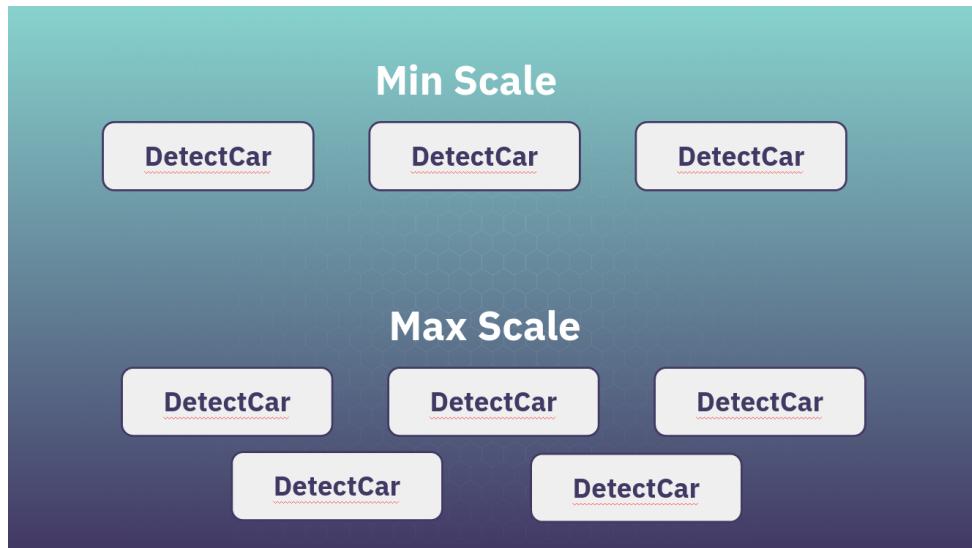
Hình 4.14 mô tả kiến trúc của hệ thống đếm số lượng người tham gia giao thông. Kiến trúc của hệ thống này chỉ khác kiến trúc 4.8 ở service xử lý CountPerson.



Hình 4.14: Hệ thống đếm số lượng người tham gia giao thông

Tất nhiên, hệ thống này không chỉ giới hạn bởi xe hơi và người tham gia giao thông, chỉ cần thay đổi service DetectCar thành service DetectBike thì chúng em đã có thêm hệ thống nhận diện xe đạp riêng mà không phải cài đặt lại một hệ thống mới từ đầu nữa, mà chỉ việc thêm module cần để xử lý mà thôi.

Việc các service được chúng em đóng gói bằng Docker có ý nghĩa rất lớn. Chúng em có thể áp dụng nhiều công nghệ dựa trên việc này. Dễ dàng nhất đó chính là chạy nhiều phiên bản copy của service trên cùng một máy, hay còn gọi là chạy với nhiều instance. Nếu không có Docker, việc chạy một service với nhiều phiên bản copy có thể sẽ bị đụng độ việc chiếm dụng tài nguyên. Đơn giản nhất là service gateway chạy trên port 8082, nếu có 2 bản copy cùng chạy thì cả 2 đều giành quyền truy cập đến port 8082, nhưng chỉ 1 trong số đó chạy trước thì sẽ thành công và những bản copy còn lại sẽ bị hệ điều hành từ chối.



Hình 4.15: Mô tả min scale và max scale

Với Docker, mọi chuyện dễ dàng hơn, vì lúc này chúng em sử dụng thêm công nghệ Kubernetes để quản lý toàn bộ các service chạy bằng Docker. Việc này giúp chúng em dễ dàng scale hơn nhiều. Ví dụ với service DetectCar nǎng về xử lý, tính toán thì chúng em cho mặc định min scale là 3, nghĩa là ban đầu chúng em chạy 3 bản copy của DetectCar song song với nhau, nghĩa là cùng một lúc chúng em có thể xử lý 3 yêu cầu về nhận diện xe hơi đang tham gia giao thông từ ảnh của camera. Bên cạnh đó, chúng em cài đặt thêm max scale là 5, điều này có nghĩa là khi số yêu cầu trả về quá nhiều và số instance hiện tại không đáp ứng kịp, hệ thống sẽ tự scale service lên đến max scale để đáp ứng việc xử lý nhanh chóng. Tránh tình trạng yêu cầu chờ được xử lý quá lâu.

# Chương 5

## Kết luận

*Nội dung của chương 5 trình bày tóm lược những nội dung mà chúng em đã tìm hiểu và các kết quả đạt được trong quá trình thực hiện đề tài, qua đó mở ra hướng phát triển của đề tài trong tương lai.*

### 5.1 Các kết quả đạt được

Qua đề tài này, chúng em đã tìm hiểu và nghiên cứu về kiến trúc của hệ thống và các vấn đề liên quan để làm thế nào tổ chức hệ thống của chúng em đảm bảo khả năng chịu tải lớn, hiệu năng cao để cho người sử dụng để có thể đáp ứng nhu cầu về hiệu năng cho người sử dụng. Qua đó chúng em nắm rõ hơn cách tổ chức, khả năng thiết kế hệ thống đảm bảo hiệu năng cao với từng bài toán cụ thể.

Ngoài ra, chúng em còn nắm vững hơn các kiến thức liên quan đến các kĩ thuật được các lập trình viên trên thế giới sử dụng ở thời điểm hiện tại và các công nghệ, framework phổ biến nhất liên quan đến việc áp dụng kĩ thuật đó.

Dựa vào các kiến thức được tìm hiểu, chúng em đã tiến hành xây dựng một platform phục vụ cho các lập trình viên sử dụng, đặc biệt là các lập trình viên về mảng AI. Giúp họ tập trung xây dựng logic các service của mình, sau đó hỗ trợ việc deploy, kết hợp các service đó lại với nhau thông

qua hệ thống của chúng em và chạy thành một luồng hoàn chỉnh mà không cần quá tốn nhiều công sức nhưng vẫn đảm bảo được hiệu năng cho toàn bộ hệ thống.

## 5.2 Hướng phát triển của đề tài

Cùng với sự phát triển không ngừng của công nghệ và chúng ta đang trong thời kì cách mạng công nghiệp 4.0, nên việc phát triển phần mềm sẽ ngày càng có nhu cầu cao và phổ biến. Vì vậy nhu cầu phát triển các ứng dụng và deploy để đáp ứng nhu cầu cho người tiêu dùng của lập trình viên sẽ ngày càng cao, nhằm phục vụ nhu cầu cho người tiêu dùng trong thời đại công nghệ ngày nay.

Trong tương lai, chúng em sẽ tiếp tục tìm hiểu và cải thiện hiệu năng của platform để đảm bảo các service của các lập trình viên trên platform của chúng em có tốc độ nhanh. Bên cạnh đó chúng em sẽ cố gắng tối ưu hóa, làm cho việc tích hợp vào hệ thống đơn giản hiệu quả hơn, thay vì vào từng file cấu hình để chỉnh sửa, chúng em mong muốn trong tương lai cung cấp một giao diện cho người sử dụng tương tác trên đó để thay đổi cấu hình theo mong muốn của mình.

Bên cạnh đó với giao diện hiện tại khá đơn giản, chưa được hoàn thiện về trải nghiệm người dùng. Trong tương lai có thể phát triển hơn nữa để hỗ trợ việc thao tác trên giao diện để tạo các yêu cầu đến dữ liệu, cải thiện giao diện sử dụng, cải thiện các bước cập nhật trạng thái xử lý trên từng service, hiển thị chi tiết và rõ ràng thông tin ở từng service hơn hiện tại.

# Tài liệu tham khảo

- [1] “Everything you need to know about kafka in 10 minutes.” <https://kafka.apache.org/intro>.
- [2] “Kafka use cases.” <https://kafka.apache.org/uses>.
- [3] “An introduction to redis data types and abstractions.” <https://redis.io/topics/data-types-intro>.
- [4] “Redis cluster tutorial.” <https://redis.io/topics/cluster-tutorial>.
- [5] “Rabbitmq tutorials.” <https://www.rabbitmq.com/getstarted.html>.
- [6] “Caching overview.” <https://aws.amazon.com/caching/>.
- [7] “Pattern: Microservice architecture.” <https://microservices.io/patterns/microservices.html>.
- [8] “The api gateway pattern versus the direct client-to-microservice communication.” <https://docs.microsoft.com/en-us/dotnet/architecture/microservices/architect-microservice-container-applications/direct-client-to-microservice-communication-versus-the-api-gateway-pattern>.
- [9] “Spring boot framework.” <https://spring.io/projects/spring-framework>.

- [10] “Mysql service.” <https://dev.mysql.com/doc/refman/8.0/en/>.
- [11] “Minio quickstart guide.” <https://docs.min.io/docs/minio-quickstart-guide.html>.