

**ĐẠI HỌC QUỐC GIA HÀ NỘI  
TRƯỜNG ĐẠI HỌC CÔNG NGHỆ**



**Nguyễn Hải Đan**

**ĐÁNH GIÁ CÁC CƠ CHẾ GIAO TIẾP TRONG KIẾN  
TRÚC MICROSERVICE**

**KHÓA LUẬN TỐT NGHIỆP ĐẠI HỌC HỆ CHÍNH QUY**  
**Ngành: Khoa học máy tính**

**HÀ NỘI – 2025**

ĐẠI HỌC QUỐC GIA HÀ NỘI  
TRƯỜNG ĐẠI HỌC CÔNG NGHỆ

Nguyễn Hải Đan

ĐÁNH GIÁ CÁC CƠ CHẾ GIAO TIẾP TRONG KIẾN  
TRÚC MICROSERVICE

KHÓA LUẬN TỐT NGHIỆP ĐẠI HỌC HỆ CHÍNH QUY  
Ngành: Khoa học máy tính

Cán bộ hướng dẫn: PGS.TS. Võ Đình Hiếu

HÀ NỘI – 2025

**VIETNAM NATIONAL UNIVERSITY, HANOI  
UNIVERSITY OF ENGINEERING AND TECHNOLOGY**

**Nguyễn Hải Đan**

**STUDY OF COMMUNICATION MECHANISMS IN  
MICROSERVICE ARCHITECTURE**

**Major: Computer Science**

**Supervisor: Assoc. Prof., Dr. Vo Dinh Hieu**

**HÀ NỘI – 2025**

## TÓM TẮT

**Tóm tắt:** Kiến trúc microservice đã trở thành một xu hướng quan trọng trong phát triển phần mềm hiện đại, cho phép xây dựng các hệ thống phức tạp từ các dịch vụ nhỏ, độc lập. Một trong những thách thức chính trong kiến trúc này là việc lựa chọn và triển khai các mẫu giao tiếp phù hợp giữa các microservice. Khóa luận này tập trung nghiên cứu và phân tích các mẫu giao tiếp trong kiến trúc microservice, bao gồm mẫu giao tiếp đồng bộ (REST), mẫu giao tiếp bất đồng bộ dạng một-một (Message Queue) và mẫu giao tiếp bất đồng bộ dạng một-nhiều (Publish/Subscribe). Nghiên cứu đánh giá hiệu suất và độ tin cậy của từng mẫu giao tiếp thông qua triển khai và kiểm thử các tình huống thực tế như kiểm tra tồn kho, xử lý thanh toán và thông báo đa kênh. Từ đó, khóa luận đề xuất các hướng dẫn cụ thể cho việc lựa chọn mẫu giao tiếp phù hợp với từng ngữ cảnh nghiệp vụ, góp phần nâng cao hiệu suất và khả năng mở rộng của hệ thống microservice.

**Từ khóa:** *Microservice, Mẫu giao tiếp, Kiến trúc phần mềm, REST, Message Queue, Publish/Subscribe.*

## ABSTRACT

**Abstract:** Microservice architecture has become a significant trend in modern software development, enabling the construction of complex systems from small, independent services. One of the main challenges in this architecture is selecting and implementing suitable communication patterns between microservices. This thesis focuses on researching and analyzing communication patterns in microservice architecture, including synchronous patterns (REST), asynchronous one-to-one patterns (Message Queue), and asynchronous one-to-many patterns (Publish/Subscribe). The study evaluates the performance and reliability of each communication pattern through implementation and testing of real-world scenarios such as inventory checking, payment processing, and multi-channel notification. Based on these findings, the thesis proposes specific guidelines for selecting appropriate communication patterns for different business contexts, contributing to improving the performance and scalability of microservice systems.

**Keywords:** *Microservice, Communication Patterns, Software Architecture, REST, Message Queue, Publish/Subscribe.*

# Mục lục

<b>Chương 1. Mở đầu</b>	1
1.1. Bối cảnh và sự cần thiết của đề tài	1
1.2. Mục tiêu nghiên cứu	2
1.3. Phạm vi nghiên cứu	3
1.4. Phương pháp nghiên cứu	4
1.5. Ý nghĩa khoa học và thực tiễn	4
1.6. Cấu trúc khóa luận	5
<b>Chương 2. Kiến thức nền tảng</b>	6
2.1. Tổng quan về Microservice Architecture	6
2.1.1. Định nghĩa và đặc điểm	6
2.1.2. So sánh với kiến trúc nguyên khối (Monolithic)	6
2.1.3. Lợi ích và thách thức của kiến trúc microservice	7
2.1.4. Các nguyên tắc thiết kế	8
2.2. Giao tiếp trong kiến trúc vi dịch vụ	8
2.2.1. Vai trò của giao tiếp trong kiến trúc vi dịch vụ	9
2.2.2. Các thuộc tính quan trọng của giao tiếp vi dịch vụ	9
2.2.3. Các mô hình giao tiếp cơ bản	9
2.2.4. Kiểu tương tác	10
2.2.5. Các công nghệ và giao thức phổ biến	10
2.2.6. Thách thức trong giao tiếp vi dịch vụ	10
2.2.7. Các mẫu giao tiếp (Communication Patterns)	11
2.3. Công nghệ và phương pháp đo lường hiệu năng	12
2.3.1. Các công nghệ triển khai trong dự án	12
2.3.2. Các thông số đo lường chính	12
2.3.3. Phương pháp đo lường	13
2.3.4. Công cụ đo lường hiệu năng	14
2.4. Tổng kết	14
<b>Chương 3. Phân tích các mẫu giao tiếp</b>	17
3.1. Cách phân loại các mẫu giao tiếp	17
3.1.1. Tiêu chí phân loại theo communication mode (sync/async)	17
3.1.2. Tiêu chí phân loại theo communication scope (one-to-one/one-to-many)	18
3.1.3. Các yếu tố ảnh hưởng đến việc lựa chọn pattern	19
3.2. Synchronous Communication Patterns (one-to-one)	20
3.2.1. Request/Response Pattern	20
3.2.2. Ưu điểm và Hạn chế của Synchronous Communication	22
3.2.3. Use cases phù hợp cho Synchronous Communication	23
3.2.4. Case studies	24
3.2.5. Tương lai của Synchronous Communication trong Microservices	25

3.2.6. Kết luận .....	25
3.3. Asynchronous Communication Patterns (one-to-one) .....	26
3.3.1. Cơ chế hoạt động .....	26
3.3.2. One-way Notifications Pattern .....	27
3.3.3. Message Queue Pattern .....	27
3.3.4. Ưu điểm và Hạn chế của Asynchronous Communication (one-to-one) ....	28
3.3.5. Use cases phù hợp cho Asynchronous Communication (one-to-one).....	29
3.3.6. Case studies .....	30
3.3.7. Kết luận .....	31
3.4. Asynchronous Communication Patterns (one-to-many) .....	31
3.4.1. Cơ chế hoạt động .....	31
3.4.2. Publish/Subscribe Pattern .....	32
3.4.3. Event Sourcing Pattern .....	33
3.4.4. Ưu điểm và Hạn chế của Asynchronous Communication (one-to-many)..	33
3.4.5. Use cases phù hợp cho Asynchronous Communication (one-to-many) ....	34
3.4.6. Kết luận .....	35
3.5. Tổng kết.....	36
<b>Chương 4. Triển khai thử nghiệm.....</b>	<b>38</b>
4.1. Mô tả bài toán và yêu cầu .....	38
4.1.1. Giới thiệu bài toán.....	38
4.1.2. Yêu cầu hệ thống .....	38
4.1.3. Kiến trúc tổng thể hệ thống .....	39
4.1.4. Các mẫu giao tiếp đánh giá .....	39
4.2. Cài đặt và triển khai.....	40
4.2.1. Cấu trúc dự án.....	40
4.2.2. Triển khai các microservice chính.....	40
4.2.3. Triển khai các mẫu giao tiếp .....	49
4.2.4. Thiết lập môi trường kiểm thử .....	53
4.3. Kết quả triển khai.....	55
4.3.1. Phương pháp đánh giá.....	55
4.3.2. Kết quả đánh giá Order-Inventory.....	56
4.3.3. Kết quả đánh giá Order-Payment .....	56
4.3.4. Kết quả đánh giá Order-Notification.....	57
4.3.5. Kết quả đánh giá User Activity Logging.....	58
4.3.6. Đánh giá tổng thể.....	58
<b>Chương 5. Đánh giá và thảo luận .....</b>	<b>61</b>
5.1. Phương pháp và tiêu chí đánh giá .....	61
5.1.1. Tổng quan phương pháp đánh giá .....	61
5.1.2. Tiêu chí đánh giá cụ thể.....	61
5.1.3. Công cụ và môi trường đánh giá.....	62
5.2. Kết quả đánh giá .....	62

5.2.1. Mẫu giao tiếp trong kịch bản Order-Inventory .....	62
5.2.2. Mẫu giao tiếp trong kịch bản Order-Payment .....	63
5.2.3. Mẫu giao tiếp trong kịch bản Order-Notification .....	63
5.2.4. Mẫu giao tiếp trong kịch bản User Activity Logging .....	63
5.2.5. Ảnh hưởng của tải hệ thống đến các mẫu giao tiếp .....	63
5.3. Thảo luận .....	64
5.3.1. Lựa chọn mẫu giao tiếp phù hợp cho từng kịch bản .....	64
5.3.2. Mô hình tích hợp các mẫu giao tiếp.....	65
5.3.3. Tối ưu hóa hiệu suất trong thực tế.....	65
5.3.4. Xu hướng phát triển và nghiên cứu tiếp theo .....	66
5.3.5. Các hạn chế của dự án và hướng cải thiện.....	66



## **Danh sách hình vẽ**

## Danh sách bảng

3.1. So sánh các kiểu tương tác trong giao tiếp vi dịch vụ .....	17
4.1. Kết quả đo lường hiệu suất Order-Inventory .....	56
4.2. Kết quả đánh giá tính nhất quán dữ liệu .....	57
4.3. Kết quả đo lường hiệu suất Order-Payment .....	58
4.4. Kết quả đo lường hiệu suất Order-Notification .....	59
4.5. Kết quả đo lường hiệu suất User Activity Logging .....	60

## **LỜI CAM ĐOAN**

Tôi xin cam đoan đây là công trình nghiên cứu của riêng tôi. Các số liệu, kết quả nêu trong khóa luận là trung thực và chưa từng được ai công bố trong bất kỳ công trình nào khác. Các tham khảo, trích dẫn trong khóa luận đều được chỉ rõ nguồn gốc. Nếu sai tôi xin hoàn toàn chịu trách nhiệm.

Hà Nội, ngày .....tháng .....năm 2025

*Người cam đoan*

*Nguyễn Hải Đan*

# Chương 1.

## Mở đầu

### 1.1. Bối cảnh và sự cần thiết của đề tài

Trong bối cảnh phát triển phần mềm hiện đại, các hệ thống ngày càng trở nên phức tạp và đòi hỏi khả năng mở rộng cao để đáp ứng nhu cầu kinh doanh không ngừng thay đổi. Kiến trúc microservice đã nổi lên như một giải pháp hiệu quả, cho phép các nhóm phát triển độc lập tạo ra, triển khai và mở rộng các dịch vụ nhỏ, tập trung vào một chức năng cụ thể của hệ thống. Sự phân tách này tạo điều kiện cho việc phát triển nhanh chóng và linh hoạt, giúp các tổ chức thích ứng tốt hơn với các yêu cầu thay đổi của thị trường.

Theo một báo cáo của IDC, đến năm 2025, hơn 80% các tổ chức doanh nghiệp sẽ chuyển đổi sang kiến trúc phân tán như microservice để tăng tốc độ phát triển và triển khai ứng dụng [1]. Xu hướng này phản ánh nhu cầu ngày càng tăng về khả năng mở rộng, tính linh hoạt và tốc độ phát triển trong môi trường kinh doanh cạnh tranh. Tuy nhiên, việc phân tách một hệ thống thành nhiều dịch vụ nhỏ cũng đặt ra thách thức lớn về cách thức các dịch vụ này giao tiếp với nhau.

Trong kiến trúc microservice, giao tiếp giữa các dịch vụ là một yếu tố then chốt quyết định đến hiệu suất tổng thể của hệ thống. Mô hình giao tiếp không phù hợp có thể tạo ra các điểm nghẽn, làm tăng độ trễ và giảm khả năng đáp ứng của hệ thống. Khi nhiều microservice phải tương tác với nhau để hoàn thành một nhiệm vụ, hiệu suất của toàn bộ chuỗi dịch vụ có thể bị ảnh hưởng bởi thời gian phản hồi của dịch vụ chậm nhất hoặc khả năng xử lý thấp nhất trong chuỗi.

Về độ tin cậy, trong một hệ thống phân tán, các lỗi giao tiếp có thể xảy ra ở nhiều điểm, ảnh hưởng đến tính nhất quán và độ sẵn sàng của dịch vụ. Mạng không ổn định, dịch vụ quá tải, hoặc các sự cố không lường trước đều có thể dẫn đến lỗi giao tiếp. Các cơ chế phát hiện lỗi, xử lý lỗi và khôi phục là rất quan trọng để duy trì tính khả dụng của hệ thống.

Các mẫu giao tiếp phải hỗ trợ việc mở rộng số lượng dịch vụ và lưu lượng giao dịch mà không làm giảm hiệu suất, đồng thời đảm bảo hệ thống hoạt động ổn định ngay cả khi một số dịch vụ gặp sự cố. Khả năng mở rộng là đặc biệt quan trọng khi doanh nghiệp phát triển và nhu cầu về hệ thống tăng lên. Các mẫu giao tiếp không chỉ phải xử lý được lưu lượng hiện tại mà còn phải có khả năng thích ứng với sự tăng trưởng trong tương lai.

Thực tế cho thấy, việc lựa chọn cơ chế giao tiếp không phù hợp có thể dẫn đến các vấn đề nghiêm trọng. Theo nghiên cứu của Gartner, hơn 70% các dự án microservice gặp khó khăn trong giai đoạn đầu triển khai do thiếu hiểu biết về các mô hình giao tiếp và cách thức áp dụng chúng hiệu quả [2]. Nhiều tổ chức đã phải thiết kế lại kiến trúc của họ sau khi gặp phải các vấn đề về hiệu suất, độ tin cậy và khả năng quản lý.

Các tổ chức phải đối mặt với sự đánh đổi giữa tính nhất quán và hiệu suất, giữa độ tin cậy và độ phức tạp. Những quyết định về cơ chế giao tiếp có ảnh hưởng sâu sắc đến kiến trúc tổng thể và thành công của hệ thống. Ví dụ, giao tiếp đồng bộ có thể đơn giản và dễ triển khai, nhưng có thể ảnh hưởng đến hiệu suất và khả năng chịu lỗi. Ngược lại, giao tiếp bất đồng bộ có thể cải thiện khả năng mở rộng và chịu lỗi, nhưng làm tăng độ phức tạp trong việc quản lý trạng thái và đảm bảo tính nhất quán của dữ liệu [5].

Do đó, việc phân tích và đánh giá các mẫu giao tiếp (communication patterns) trong kiến trúc microservice là vô cùng cần thiết, giúp các nhà phát triển và kiến trúc sư hệ thống hiểu rõ các lựa chọn có sẵn và những đánh đổi liên quan. Thông qua việc phân tích có hệ thống, các nhà phát triển có thể áp dụng các mẫu phù hợp với bối cảnh cụ thể, tối ưu hóa hiệu suất và độ tin cậy của hệ thống, đồng thời đảm bảo khả năng mở rộng trong tương lai.

Bài đánh giá này nhằm cung cấp một cái nhìn toàn diện về các mẫu giao tiếp trong kiến trúc microservice, phân tích ưu nhược điểm của từng mẫu, và đưa ra các hướng dẫn thực tiễn cho việc lựa chọn và triển khai các mẫu giao tiếp phù hợp.

## 1.2. Mục tiêu nghiên cứu

Khóa luận này hướng đến việc phân tích và đánh giá các mẫu giao tiếp trong kiến trúc microservice nhằm cung cấp cơ sở lý thuyết và thực tiễn cho việc lựa chọn, thiết kế và triển khai các cơ chế giao tiếp hiệu quả. Việc hiểu rõ và áp dụng đúng các mẫu giao tiếp không chỉ giúp tối ưu hóa hiệu suất hệ thống mà còn đảm bảo tính linh hoạt và khả năng mở rộng - những yếu tố then chốt trong thành công của các hệ thống microservice hiện đại.

Đồng thời, khóa luận sẽ phân tích chi tiết và so sánh các mẫu giao tiếp trên nhiều khía cạnh. Về cơ chế hoạt động, khóa luận sẽ mô tả chi tiết cách thức các mẫu giao tiếp hoạt động, bao gồm các thành phần, luồng dữ liệu và tương tác giữa các dịch vụ. Về hiệu suất và độ trễ, khóa luận sẽ đánh giá thời gian phản hồi, thông lượng và khả năng xử lý đồng thời của các mẫu giao tiếp. Về khả năng mở rộng, khóa luận sẽ phân tích khả năng của các mẫu giao tiếp trong việc hỗ trợ việc mở rộng số lượng dịch vụ và lưu lượng giao dịch. Về độ tin cậy và khả năng chịu lỗi, khóa luận sẽ đánh giá khả năng của các mẫu giao tiếp trong việc duy trì hoạt động ổn định khi gặp lỗi hoặc sự cố. Về độ phức tạp trong triển khai và bảo trì, khóa luận sẽ xem xét mức độ phức tạp và nguồn lực cần thiết để triển khai và duy trì các mẫu giao tiếp. Cuối cùng, về tính phù hợp với các tình huống cụ thể, khóa luận sẽ xác định các ngữ cảnh và yêu cầu mà mỗi mẫu giao tiếp phù hợp nhất.

Để đánh giá các mẫu giao tiếp một cách khách quan, khóa luận sẽ xây dựng và triển khai môi trường thử nghiệm mô phỏng các kịch bản thực tế. Các kịch bản này bao gồm kiểm tra và cập nhật tồn kho, xử lý thanh toán, phân phối thông báo, và ghi nhận hoạt động người dùng. Những kịch bản này được chọn vì chúng đại diện cho các tình huống phổ biến trong các ứng dụng thực tế và đòi hỏi các đặc tính giao tiếp khác nhau. Thông

qua việc triển khai và đánh giá các mẫu giao tiếp trong các kịch bản này, khóa luận có thể cung cấp một cái nhìn thực tế về hiệu quả của từng mẫu giao tiếp.

Khóa luận sẽ đo lường và phân tích hiệu năng của các mẫu giao tiếp trong điều kiện khác nhau về tải và độ trễ mạng. Việc đo lường này bao gồm các chỉ số như thời gian phản hồi, thông lượng, tỷ lệ lỗi, tính nhất quán dữ liệu, và khả năng phục hồi sau lỗi. Thông qua việc phân tích các chỉ số này, khóa luận có thể xác định các điểm mạnh và điểm yếu của từng mẫu giao tiếp trong các điều kiện khác nhau.

Kết quả cuối cùng của khóa luận là tổng hợp các nguyên tắc và hướng dẫn thực tiễn cho việc lựa chọn mẫu giao tiếp phù hợp dựa trên yêu cầu cụ thể của ứng dụng. Các hướng dẫn này sẽ giúp các nhà phát triển và kiến trúc sư hệ thống đưa ra quyết định sáng suốt về cách thức các dịch vụ giao tiếp với nhau, từ đó tối ưu hóa hiệu suất, độ tin cậy và khả năng mở rộng của hệ thống microservice.

### 1.3. Phạm vi nghiên cứu

Để đảm bảo tính khả thi và giá trị thực tiễn của khóa luận, phạm vi đánh giá được giới hạn như sau:

Về nội dung, khóa luận tập trung vào các mẫu giao tiếp phổ biến trong kiến trúc microservice. Các mẫu này được lựa chọn không chỉ vì tính phổ biến của chúng mà còn vì chúng đại diện cho các phương pháp tiếp cận khác nhau trong việc giải quyết vấn đề giao tiếp giữa các dịch vụ. Cụ thể, khóa luận sẽ tập trung vào giao tiếp đồng bộ thông qua REST API, giao tiếp bất đồng bộ one-to-one thông qua Message Queue (RabbitMQ), và giao tiếp bất đồng bộ one-to-many thông qua Pub/Sub (Kafka) [6].

Các khía cạnh được khóa luận phân tích bao gồm mô hình giao tiếp và luồng dữ liệu, mô tả cách thức các dịch vụ tương tác và trao đổi thông tin với nhau. Đo lường hiệu năng và tối ưu hóa là một khía cạnh quan trọng khác, bao gồm việc đánh giá hiệu suất của các mẫu giao tiếp và cách thức tối ưu hóa hiệu suất. Cuối cùng, khóa luận xem xét tính mở rộng và khả năng chịu tải của các mẫu giao tiếp, đánh giá khả năng của chúng trong việc hỗ trợ hệ thống mở rộng khi nhu cầu tăng lên.

Khóa luận không đi sâu vào chi tiết kỹ thuật triển khai của từng công nghệ cụ thể, vì điều này nằm ngoài phạm vi của một khóa luận tổng quan về các mẫu giao tiếp. Các vấn đề bảo mật và quản lý danh tính được đề cập nhưng không phải là trọng tâm, vì chúng đòi hỏi một phân tích chuyên sâu riêng biệt. Tương tự, các vấn đề liên quan đến cơ sở dữ liệu và quản lý trạng thái cũng nằm ngoài phạm vi chính của khóa luận này.

Về thực nghiệm, khóa luận sẽ xây dựng ứng dụng mẫu với 4 kịch bản thực tế để đánh giá hiệu quả của các mẫu giao tiếp khác nhau. Kịch bản đầu tiên là Order-Inventory, liên quan đến việc kiểm tra và cập nhật tồn kho. Đây là một tình huống phổ biến trong các hệ thống thương mại điện tử, đòi hỏi tính nhất quán dữ liệu cao. Kịch bản thứ hai là Order-Payment, liên quan đến việc xử lý thanh toán. Quá trình thanh toán thường đòi hỏi độ tin cậy cao và cần xử lý các tình huống lỗi một cách cẩn thận. Kịch bản thứ ba là Order-Notification, liên quan đến việc phân phối thông báo. Việc gửi thông báo thường

yêu cầu một cơ chế giao tiếp một-đến-nhiều, phản ánh nhu cầu thông báo cho nhiều đối tượng khác nhau. Kịch bản cuối cùng là User Activity Logging, liên quan đến việc ghi nhận hoạt động người dùng. Đây là một tình huống cần khả năng xử lý lượng lớn các sự kiện, đòi hỏi một cơ chế giao tiếp có khả năng chịu tải cao.

Các tiêu chí đánh giá trong thực nghiệm bao gồm độ trễ trung bình và percentile thứ 95, phản ánh thời gian phản hồi trung bình và trong trường hợp xấu. Thông lượng là một chỉ số khác, đo lường số lượng giao dịch có thể xử lý trong một đơn vị thời gian. Tỷ lệ lỗi được theo dõi để đánh giá độ tin cậy của các mẫu giao tiếp. Tính nhất quán dữ liệu, khả năng phục hồi sau lỗi, và khả năng mở rộng theo chiều ngang cũng là những tiêu chí quan trọng được đánh giá.

## **1.4. Phương pháp nghiên cứu**

Khóa luận kết hợp phương pháp nghiên cứu lý thuyết và thực nghiệm để đánh giá toàn diện các mẫu giao tiếp trong kiến trúc microservice.

Về phương pháp lý thuyết, khóa luận phân tích tài liệu học thuật, báo cáo kỹ thuật từ các nguồn uy tín như IEEE, ACM, và các tài liệu từ công ty công nghệ hàng đầu như Netflix và Uber. Khóa luận thực hiện phân tích so sánh các mẫu giao tiếp dựa trên các tiêu chí định lượng và định tính như hiệu suất, độ tin cậy, khả năng mở rộng, và phân tích các trường hợp thực tế về triển khai mẫu giao tiếp trong các tổ chức lớn.

Về phương pháp thực nghiệm, khóa luận phát triển ứng dụng microservice mẫu theo kiến trúc tham chiếu, mô phỏng các tình huống thực tế như quản lý đơn hàng, thanh toán, gửi thông báo và ghi nhận hoạt động người dùng. Khóa luận thiết kế các kịch bản thử nghiệm mô phỏng các điều kiện tải khác nhau, thu thập dữ liệu về hiệu suất, độ tin cậy và khả năng mở rộng, và áp dụng phương pháp thống kê để phân tích dữ liệu.

Quy trình đánh giá gồm 5 giai đoạn: Giai đoạn 1 - nghiên cứu lý thuyết và tổng hợp tài liệu; Giai đoạn 2 - thiết kế và phát triển ứng dụng mẫu; Giai đoạn 3 - thiết lập môi trường và thực hiện thử nghiệm; Giai đoạn 4 - phân tích dữ liệu và đánh giá hiệu quả các mẫu giao tiếp; Giai đoạn 5 - tổng hợp kết quả và xây dựng hướng dẫn thực tiễn.

Phương pháp nghiên cứu này đảm bảo cung cấp cái nhìn toàn diện về hiệu quả của các mẫu giao tiếp trong kiến trúc microservice, từ lý thuyết đến thực tiễn, từ định tính đến định lượng.

## **1.5. Ý nghĩa khoa học và thực tiễn**

Khóa luận này mang lại đóng góp giá trị cả về khoa học và ứng dụng thực tiễn trong lĩnh vực kiến trúc phần mềm và hệ thống phân tán.

Về mặt khoa học, khóa luận hệ thống hóa các mẫu giao tiếp trong kiến trúc microservice và xây dựng khung phân tích toàn diện cho việc đánh giá chúng. Qua việc tổng hợp tài liệu học thuật, khóa luận xác định được các nguyên tắc cơ bản và yếu tố ảnh hưởng đến hiệu quả của từng mẫu giao tiếp, tạo nền tảng lý thuyết cho nghiên cứu tương

lai. Khóa luận cũng phát triển phương pháp đánh giá mới kết hợp các tiêu chí định lượng và định tính, giúp các nhà nghiên cứu đưa ra quyết định dựa trên dữ liệu thực nghiệm.

Về mặt thực tiễn, khóa luận cung cấp hướng dẫn cụ thể cho việc lựa chọn và triển khai các mẫu giao tiếp trong dự án thực tế. Thông qua phân tích trường hợp thực tế, khóa luận rút ra các nguyên tắc và thực tiễn tốt nhất, giúp nhà phát triển áp dụng các mẫu giao tiếp hiệu quả và tránh lỗi phổ biến. Ứng dụng microservice mẫu và các kịch bản thử nghiệm được phát triển trong khóa luận cho phép các nhà phát triển đánh giá các mẫu giao tiếp trong môi trường kiểm soát trước khi triển khai thực tế, giảm thiểu rủi ro và tăng cường sự tự tin.

Ngoài ra, khóa luận nâng cao nhận thức về các mẫu giao tiếp trong cộng đồng phát triển, khuyến khích chia sẻ kiến thức và thúc đẩy phát triển công cụ hỗ trợ triển khai mẫu giao tiếp hiệu quả. Điều này đóng góp vào việc xây dựng hệ sinh thái phong phú cho phát triển và triển khai hệ thống microservice.

## **1.6. Cấu trúc khóa luận**

Khóa luận được tổ chức thành 5 chương chính, mỗi chương tập trung vào một khía cạnh cụ thể của nghiên cứu.

Chương 1, Mở đầu, giới thiệu bối cảnh và sự cần thiết của đề tài, xác định mục tiêu, phạm vi và phương pháp nghiên cứu, và trình bày ý nghĩa khoa học và thực tiễn của nghiên cứu. Chương 2, Kiến trúc nền tảng, cung cấp tổng quan về kiến trúc microservice, khái niệm và phân loại các mẫu giao tiếp, và giới thiệu các công nghệ và giao thức giao tiếp phổ biến. Chương 3, Phân tích các mẫu giao tiếp, phân loại các mẫu giao tiếp theo tiêu chí đồng bộ/bất đồng bộ và one-to-one/one-to-many, phân tích chi tiết các mẫu giao tiếp đồng bộ (one-to-one), phân tích chi tiết các mẫu giao tiếp bất đồng bộ (one-to-one), phân tích chi tiết các mẫu giao tiếp bất đồng bộ (one-to-many), và so sánh và đánh giá các mẫu giao tiếp. Chương 4, Triển khai thử nghiệm, mô tả bài toán và yêu cầu, thiết kế và cài đặt ứng dụng mẫu, cài đặt và triển khai các mẫu giao tiếp, và kết quả triển khai và đánh giá hiệu năng. Chương 5, Đánh giá và thảo luận, phân tích kết quả thực nghiệm, thảo luận về các phát hiện chính, đề xuất các nguyên tắc lựa chọn mẫu giao tiếp, và kết luận và hướng phát triển.



# Chương 2.

## Kiến thức nền tảng

### 2.1. Tổng quan về Microservice Architecture

#### 2.1.1. Định nghĩa và đặc điểm

Kiến trúc Microservice là một phương pháp phát triển phần mềm trong đó một ứng dụng được cấu thành từ nhiều dịch vụ nhỏ, độc lập và có khả năng triển khai riêng biệt. Mỗi dịch vụ này được thiết kế để thực hiện một chức năng cụ thể trong phạm vi nghiệp vụ được định nghĩa rõ ràng, và giao tiếp với các dịch vụ khác thông qua các cơ chế giao tiếp nhẹ, thường là API [8].

Các đặc điểm chính của kiến trúc microservice bao gồm tính tự trị cao, trong đó mỗi dịch vụ có thể được phát triển, triển khai và mở rộng độc lập với các dịch vụ khác [5]. Các dịch vụ được tổ chức xoay quanh các khả năng nghiệp vụ thay vì các lớp công nghệ, thể hiện sự phân tách theo chức năng nghiệp vụ. Quản lý dữ liệu trong microservice được thực hiện phi tập trung, với mỗi dịch vụ quản lý dữ liệu riêng và chỉ có thể truy cập dữ liệu thông qua API của dịch vụ sở hữu dữ liệu đó.

Thiết kế hướng lỗi là một đặc điểm quan trọng khác của microservice, trong đó các dịch vụ được thiết kế để xử lý lỗi và khả năng các dịch vụ khác không khả dụng. Cuối cùng, microservice cho phép tiến hóa độc lập, nghĩa là các dịch vụ có thể thay đổi và phát triển theo thời gian mà không ảnh hưởng đến toàn bộ hệ thống [4].

#### 2.1.2. So sánh với kiến trúc nguyên khối (Monolithic)

Để hiểu rõ hơn về kiến trúc microservice, việc so sánh với kiến trúc nguyên khối là rất hữu ích. Trong kiến trúc nguyên khối, toàn bộ ứng dụng được xây dựng như một đơn vị duy nhất. Tất cả các chức năng nằm trong một codebase và được triển khai cùng nhau.

Về triển khai, kiến trúc nguyên khối đòi hỏi toàn bộ ứng dụng được triển khai cùng một lúc, trong khi kiến trúc microservice cho phép các dịch vụ được triển khai độc lập [5]. Điều này có ý nghĩa quan trọng trong việc giảm thiểu rủi ro và tăng tốc độ phát hành.

Khả năng mở rộng cũng khác biệt đáng kể giữa hai kiến trúc. Trong kiến trúc nguyên khối, toàn bộ ứng dụng phải được mở rộng, ngay cả khi chỉ một phần cần thêm tài nguyên. Ngược lại, kiến trúc microservice cho phép mở rộng từng dịch vụ riêng biệt, tối ưu hóa việc sử dụng tài nguyên.

Về phát triển, kiến trúc nguyên khối thường có một nhóm phát triển làm việc trên một codebase, dẫn đến các xung đột trong quá trình phát triển và triển khai. Trong khi đó, kiến trúc microservice cho phép nhiều nhóm làm việc độc lập trên các dịch vụ khác nhau, tăng tốc độ phát triển và giảm thiểu xung đột [4].

Công nghệ là một khía cạnh khác có sự khác biệt. Kiến trúc nguyên khối thường bị giới hạn trong một stack công nghệ, trong khi mỗi microservice có thể sử dụng công nghệ phù hợp nhất với yêu cầu của nó. Điều này tạo ra sự linh hoạt và khả năng thích ứng với các công nghệ mới.

Khả năng chịu lỗi cũng là một điểm khác biệt quan trọng. Trong kiến trúc nguyên khối, lỗi ở một phần có thể ảnh hưởng đến toàn bộ ứng dụng, trong khi trong kiến trúc microservice, lỗi được cô lập trong một dịch vụ, giảm thiểu tác động đến toàn bộ hệ thống [8].

Cuối cùng, về độ phức tạp, kiến trúc nguyên khối đơn giản hơn trong các ứng dụng nhỏ, nhưng phức tạp hơn khi ứng dụng phát triển. Ngược lại, kiến trúc microservice phức tạp hơn ngay từ đầu do tính phân tán, nhưng độ phức tạp này được quản lý tốt hơn khi hệ thống phát triển.

### **2.1.3. Lợi ích và thách thức của kiến trúc microservice**

Kiến trúc microservice mang lại nhiều lợi ích đáng kể cho việc phát triển và vận hành phần mềm. Một trong những lợi ích chính là khả năng mở rộng có mục tiêu. Các dịch vụ có thể được mở rộng độc lập dựa trên nhu cầu, tối ưu hóa việc sử dụng tài nguyên. Điều này đặc biệt quan trọng trong môi trường cloud, nơi chi phí tỷ lệ thuận với tài nguyên được sử dụng.

Phát triển nhanh hơn là một lợi ích khác của kiến trúc microservice. Các nhóm nhỏ có thể làm việc trên các dịch vụ độc lập, cho phép phát triển song song và chu kỳ phát hành nhanh hơn. Mỗi nhóm có thể tập trung vào một dịch vụ cụ thể, hiểu rõ nó và phát triển nó một cách hiệu quả.

Tính linh hoạt công nghệ cũng là một lợi thế đáng kể. Mỗi dịch vụ có thể sử dụng công nghệ phù hợp nhất với yêu cầu của nó. Ví dụ, một dịch vụ xử lý giao dịch có thể sử dụng một ngôn ngữ chú trọng vào tính nhất quán, trong khi một dịch vụ phân tích dữ liệu có thể sử dụng một ngôn ngữ tối ưu cho xử lý dữ liệu lớn.

Khả năng chịu lỗi tốt hơn là một lợi ích khác của kiến trúc microservice. Lỗi trong một dịch vụ không nhất thiết phải làm cho toàn bộ hệ thống không khả dụng. Ví dụ, nếu dịch vụ gợi ý sản phẩm không hoạt động, người dùng vẫn có thể duyệt và mua sản phẩm.

Khả năng bảo trì và hiểu biết tốt hơn cũng là một lợi thế của kiến trúc microservice. Các dịch vụ nhỏ hơn dễ hiểu và bảo trì hơn các ứng dụng lớn. Mã nguồn của mỗi dịch vụ nhỏ hơn và tập trung vào một chức năng cụ thể, giúp nhà phát triển dễ dàng hiểu và thay đổi nó.

Tuy nhiên, kiến trúc microservice cũng đặt ra một số thách thức đáng kể. Độ phức tạp phân tán là một thách thức lớn. Hệ thống phân tán vốn phức tạp hơn, đòi hỏi kiến thức và công cụ chuyên biệt. Các vấn đề như latency mạng, xử lý lỗi và đồng bộ hóa dữ liệu trở nên phức tạp hơn trong một hệ thống phân tán [5].

Giao tiếp giữa các dịch vụ là một thách thức khác. Thiết kế và quản lý giao tiếp giữa các dịch vụ đòi hỏi cân nhắc kỹ lưỡng về hiệu suất, độ tin cậy và khả năng mở rộng. Việc

lựa chọn giao thức giao tiếp phù hợp và xử lý các trường hợp lỗi trong giao tiếp là các vấn đề phức tạp.

Quản lý dữ liệu cũng là một thách thức đáng kể trong kiến trúc microservice. Duy trì tính nhất quán dữ liệu giữa các dịch vụ có thể phức tạp, đặc biệt là khi mỗi dịch vụ có cơ sở dữ liệu riêng. Các mẫu như Saga và Event Sourcing được sử dụng để giải quyết vấn đề này, nhưng chúng cũng đưa ra sự phức tạp riêng.

Vận hành và giám sát là một thách thức khác của kiến trúc microservice. Triển khai và giám sát nhiều dịch vụ đòi hỏi công cụ và quy trình tinh vi hơn. Các công cụ như Kubernetes và Prometheus đã được phát triển để giải quyết vấn đề này, nhưng chúng cũng đòi hỏi kiến thức và nỗ lực đáng kể để sử dụng hiệu quả.

Cuối cùng, kiểm thử cũng trở nên phức tạp hơn trong kiến trúc microservice. Kiểm thử tích hợp đòi hỏi sự phối hợp giữa nhiều dịch vụ, có thể chạy trên các máy khác nhau và sử dụng các công nghệ khác nhau. Các kỹ thuật như kiểm thử hợp đồng và môi trường kiểm thử tích hợp được sử dụng để giải quyết vấn đề này [5].

#### **2.1.4. Các nguyên tắc thiết kế**

Để thiết kế một kiến trúc microservice hiệu quả, một số nguyên tắc thiết kế chính cần được tuân thủ. Nguyên tắc đầu tiên là Single Responsibility Principle (Nguyên tắc Trách nhiệm Đơn lẻ), theo đó mỗi dịch vụ nên chịu trách nhiệm cho một chức năng nghiệp vụ duy nhất. Điều này giúp giữ các dịch vụ đơn giản và tập trung, dễ hiểu và bảo trì.

Domain-Driven Design (DDD) là một phương pháp thiết kế hữu ích cho kiến trúc microservice. DDD sử dụng các khái niệm như Bounded Context để định nghĩa ranh giới giữa các dịch vụ. Bounded Context giúp xác định phạm vi trách nhiệm của mỗi dịch vụ và cách chúng tương tác với nhau.

API First là một nguyên tắc khác, nhấn mạnh việc thiết kế API trước, xem nó như một hợp đồng giữa các dịch vụ. Điều này giúp đảm bảo rằng các dịch vụ có thể giao tiếp hiệu quả và rằng các thay đổi không phá vỡ tương thích ngược.

Tự động hóa là một phần quan trọng của kiến trúc microservice thành công. Tự động hóa quá trình xây dựng, kiểm thử và triển khai giúp quản lý sự phức tạp của việc phát triển và vận hành nhiều dịch vụ. Các công cụ CI/CD (Continuous Integration/Continuous Deployment) là rất quan trọng trong môi trường microservice.

Monitoring và Observability là các nguyên tắc quan trọng khác. Thiết kế hệ thống để dễ dàng giám sát và hiểu được hoạt động nội bộ giúp phát hiện và giải quyết vấn đề một cách nhanh chóng. Các công cụ như logging tập trung, theo dõi phân tán và thu thập số liệu là rất quan trọng.

Cuối cùng, Fault Tolerance (Khả năng chịu lỗi) là một nguyên tắc thiết kế quan trọng cho kiến trúc microservice. Các dịch vụ nên được thiết kế để xử lý lỗi một cách thanh nhã, sử dụng các kỹ thuật như Circuit Breaker. Circuit Breaker ngăn lỗi lan truyền bằng cách ngừng gửi yêu cầu đến các dịch vụ không phản hồi.

## **2.2. Giao tiếp trong kiến trúc vi dịch vụ**

### **2.2.1. Vai trò của giao tiếp trong kiến trúc vi dịch vụ**

Giao tiếp đóng vai trò then chốt trong kiến trúc microservice. Khác với các ứng dụng nguyên khối, các microservice giao tiếp qua mạng, thường thông qua HTTP, gRPC hoặc middleware messaging.

Giao tiếp không chỉ trao đổi dữ liệu, mà còn định hình toàn bộ kiến trúc và ảnh hưởng đến tính khả dụng, hiệu suất và khả năng mở rộng của hệ thống [7].

Giao tiếp tạo điều kiện cho các dịch vụ phối hợp hoàn thành các tác vụ nghiệp vụ phức tạp. Ví dụ, quy trình đặt hàng liên quan đến các dịch vụ quản lý đơn hàng, thanh toán, kho hàng và vận chuyển cần giao tiếp để đảm bảo xử lý chính xác.

Giao tiếp đóng vai trò quan trọng trong việc đảm bảo tính nhất quán dữ liệu. Trong hệ thống dữ liệu phân tán, mỗi dịch vụ quản lý một phần dữ liệu, cần giao tiếp khi dữ liệu thay đổi để duy trì nhất quán.

Giao tiếp hỗ trợ khả năng chịu lỗi thông qua các cơ chế như Circuit Breaker, giúp hệ thống phục hồi và tiếp tục hoạt động khi gặp lỗi.

Cuối cùng, thiết kế giao tiếp tốt cho phép mở rộng hệ thống, thêm dịch vụ mới hoặc phiên bản mới mà không ảnh hưởng đến dịch vụ khác.

### **2.2.2. Các thuộc tính quan trọng của giao tiếp vi dịch vụ**

Khi thiết kế giao tiếp cho microservice, cần xem xét một số thuộc tính quan trọng:

Độ tin cậy đảm bảo thông điệp được gửi và nhận thành công trong hệ thống phân tán. Các cơ chế như xác nhận, thử lại và hàng đợi bền vững giúp tăng độ tin cậy [9].

Độ trễ là thời gian thông điệp di chuyển từ nguồn đến đích. Các yếu tố ảnh hưởng bao gồm khoảng cách vật lý giữa dịch vụ, phương pháp tuần tự hóa và tải mạng.

Khả năng mở rộng giúp xử lý khối lượng thông điệp tăng khi hệ thống phát triển, thông qua việc thêm nhiều phiên bản dịch vụ.

Cách ly lỗi ngăn lỗi lan truyền giữa các dịch vụ. Các mẫu như Circuit Breaker và Bulkhead hạn chế ảnh hưởng của lỗi từ một dịch vụ đến các dịch vụ khác.

Tính nhất quán liên quan đến cách đảm bảo dữ liệu nhất quán. Trong hệ thống phân tán, có sự đánh đổi giữa tính nhất quán, khả năng sẵn sàng và khả năng chịu đựng phân vùng (định lý CAP).

Định dạng dữ liệu đề cập cách dữ liệu được cấu trúc và tuần tự hóa. Các định dạng phổ biến như JSON, XML và Protocol Buffers có ưu nhược điểm riêng.

Khả năng tương tác cho phép các dịch vụ sử dụng công nghệ khác nhau giao tiếp với nhau, quan trọng trong môi trường đa ngôn ngữ và đa nền tảng.

Bảo mật bảo vệ thông điệp khỏi truy cập trái phép qua mã hóa, xác thực và ủy quyền khi dịch vụ giao tiếp qua mạng.

### **2.2.3. Các mô hình giao tiếp cơ bản**

Có hai mô hình giao tiếp cơ bản trong microservices: đồng bộ và bất đồng bộ.

Trong giao tiếp đồng bộ, người gửi đợi phản hồi trước khi tiếp tục xử lý. Ví dụ, một dịch vụ gửi yêu cầu HTTP đợi phản hồi trước khi tiếp tục. Mô hình này đơn giản, dễ hiểu và cung cấp phản hồi tức thì [5]. Tuy nhiên, có thể dẫn đến hiệu suất kém vì phải đợi phản hồi, hiệu ứng xung hồi khi dịch vụ chậm, và vấn đề khả năng mở rộng khi số yêu cầu tăng.

Trong giao tiếp bất đồng bộ, người gửi không đợi phản hồi. Ví dụ, dịch vụ gửi thông điệp vào hàng đợi rồi tiếp tục xử lý. Mô hình này tạo coupling lỏng lẻo và khả năng đệm tốt [9]. Tuy nhiên, phức tạp hơn để triển khai, có thể có độ trễ cao, và gây khó khăn về tính nhất quán dữ liệu.

#### **2.2.4. Kiểu tương tác**

Ngoài mô hình giao tiếp, các microservice còn tương tác theo các kiểu khác nhau:

Kiểu one-to-one là một dịch vụ giao tiếp với một dịch vụ khác, phổ biến nhất như dịch vụ đơn hàng gọi dịch vụ thanh toán.

Kiểu one-to-many là một dịch vụ giao tiếp với nhiều dịch vụ, dùng cho broadcasting như dịch vụ đơn hàng thông báo cho dịch vụ kho hàng, vận chuyển và thông báo.

Kiểu many-to-one là nhiều dịch vụ giao tiếp với một dịch vụ, dùng cho tổng hợp như nhiều dịch vụ gửi dữ liệu đến dịch vụ phân tích.

Kiểu many-to-many là nhiều dịch vụ giao tiếp với nhiều dịch vụ khác, phức tạp nhất, thường qua middleware như message broker hoặc service mesh.

#### **2.2.5. Các công nghệ và giao thức phổ biến**

HTTP/REST là giao thức phổ biến nhất cho giao tiếp đồng bộ, sử dụng phương thức HTTP và tài nguyên đại diện. REST đơn giản, dễ hiểu và được hỗ trợ rộng rãi.

gRPC là framework RPC hiệu suất cao, sử dụng HTTP/2 và Protocol Buffers. Cung cấp hiệu suất tốt hơn REST nhờ multiplexing và định dạng tuần tự hóa hiệu quả, hỗ trợ streaming hai chiều.

Message Queue là mẫu bất đồng bộ với dịch vụ gửi và nhận thông điệp qua hàng đợi như RabbitMQ, ActiveMQ và AWS SQS. Cung cấp coupling lỏng lẻo và khả năng đệm tốt, nhưng thêm độ phức tạp và độ trễ.

Pub/Sub là mẫu bất đồng bộ với nhà xuất bản gửi thông điệp mà không biết người nhận, thường qua Apache Kafka, AWS SNS/SQS, Google Pub/Sub hoặc NATS. Cung cấp coupling lỏng lẻo cao và khả năng mở rộng tốt.

GraphQL là ngôn ngữ truy vấn và thao tác dữ liệu hiệu quả, cho phép chỉ định chính xác dữ liệu cần thiết, tránh over-fetching và under-fetching [4]. Hữu ích cho ứng dụng di động với băng thông hạn chế.

#### **2.2.6. Thách thức trong giao tiếp vi dịch vụ**

Giao tiếp microservice đặt ra nhiều thách thức:

Network Reliability là thách thức quan trọng khi mạng không đáng tin cậy, gây mất thông điệp hoặc độ trễ cao. Giải pháp bao gồm retry, timeout và circuit breaker.

Service Discovery giải quyết việc các dịch vụ tìm thấy nhau trong môi trường động. Các giải pháp gồm Client-side Discovery với service registry và Server-side Discovery qua load balancer hoặc API gateway.

Load Balancing phân phối tải giữa các phiên bản dịch vụ, đảm bảo hiệu suất và độ tin cậy. Các cơ chế gồm Round Robin, Least Connections và Hash-based.

Data Consistency là thách thức duy trì tính nhất quán dữ liệu trong giao tiếp microservice. Các mẫu như Saga, Event Sourcing và CQRS [4] giúp quản lý giao dịch phân tán, lưu trạng thái và tách biệt đọc/ghi.

Versioning quản lý thay đổi API giữa các dịch vụ. Các giải pháp gồm Semantic Versioning với hệ thống đánh số (MAJOR.MINOR.PATCH), API Versioning duy trì nhiều phiên bản API, và Backward Compatibility đảm bảo tương thích ngược.

Error Handling xử lý lỗi trong hệ thống phân tán. Các cơ chế gồm Retry thử lại yêu cầu thất bại, Circuit Breaker ngăn yêu cầu đến dịch vụ không phản hồi, và Fallback cung cấp phản hồi thay thế.

Monitoring and Debugging giám sát và gỡ lỗi trong hệ thống phức tạp. Các công cụ gồm Logging tập trung, Distributed Tracing theo dõi yêu cầu qua nhiều dịch vụ, và Metrics Collection thu thập chỉ số hiệu suất.

### **2.2.7. Các mẫu giao tiếp (Communication Patterns)**

Trong kiến trúc microservice, các dịch vụ cần trao đổi thông tin để phối hợp và hoàn thành chức năng. Dưới đây là năm mẫu giao tiếp chính:

Request-Response là mẫu đồng bộ phổ biến nhất, dịch vụ gửi yêu cầu đến dịch vụ khác và đợi phản hồi. Dịch vụ gửi thiết lập kết nối HTTP/REST hoặc gRPC, chờ đợi phản hồi từ dịch vụ nhận. Mẫu này đơn giản, dễ hiểu và đảm bảo tính nhất quán dữ liệu cao. Tuy nhiên, tạo coupling chặt chẽ, hiệu suất kém khi độ trễ cao, và có nguy cơ lỗi cascade.

Event-Driven là mẫu các dịch vụ giao tiếp qua phát và lắng nghe sự kiện thông qua message broker. Dịch vụ phát hành không cần biết ai xử lý sự kiện, tạo decoupling cao và khả năng mở rộng tốt. Tuy nhiên, việc theo dõi luồng thực thi và gỡ lỗi phức tạp hơn, khó duy trì tính nhất quán dữ liệu.

Publish-Subscribe là dạng cụ thể của Event-Driven, cho phép phân phối thông tin từ một nguồn đến nhiều người nhận. Publisher gửi thông điệp đến kênh, nhiều subscribers nhận từ kênh đó. Triển khai qua Apache Kafka, RabbitMQ hoặc NATS. Phù hợp cho truyền thông tin một-đến-nhiều, dễ mở rộng, nhưng phức tạp trong quản lý tính nhất quán và có thể xử lý trùng lặp.

Point-to-Point Messaging gửi thông điệp từ nguồn đến đích cụ thể qua hàng đợi. Producer gửi thông điệp vào hàng đợi, chỉ một consumer xử lý mỗi thông điệp. Đảm bảo tin cậy cao, phù hợp cho phân phối tác vụ và cân bằng tải. Tuy nhiên, có thể nghẽn hàng đợi nếu xử lý chậm và không phù hợp khi nhiều dịch vụ cần nhận cùng thông tin.

Asynchronous Request-Response là biến thể bất đồng bộ của Request-Response. Dịch vụ gửi yêu cầu và tiếp tục xử lý, dịch vụ nhận xử lý và gửi phản hồi qua hàng đợi.

Dịch vụ gửi được thông báo qua callback, webhook hoặc long polling. Tránh blocking, cải thiện hiệu suất, nhưng phức tạp hơn trong triển khai và quản lý.

Mỗi mẫu có ưu nhược điểm riêng, lựa chọn phù hợp phụ thuộc vào yêu cầu về tính nhất quán, hiệu suất, khả năng mở rộng và độ tin cậy. Thực tế, hệ thống microservice thường kết hợp nhiều mẫu để giải quyết các tình huống khác nhau hiệu quả.

## **2.3. Công nghệ và phương pháp đo lường hiệu năng**

### **2.3.1. Các công nghệ triển khai trong dự án**

Trong triển khai kiến trúc vi dịch vụ, việc lựa chọn công nghệ phù hợp đóng vai trò quan trọng, ảnh hưởng trực tiếp đến hiệu suất, khả năng mở rộng và bảo trì của hệ thống [5]. Khóa luận sử dụng NestJS làm framework chính cho việc phát triển microservices, một framework Node.js tiên bộ dựa trên TypeScript, cung cấp kiến trúc ứng dụng lấy cảm hứng từ Angular với các nguyên tắc SOLID và mô hình MVC. Framework này mang lại lợi ích như hỗ trợ dependency injection, kiến trúc mô-đun hóa cao và tích hợp sẵn với nhiều công nghệ khác.

Mỗi dịch vụ được triển khai như một ứng dụng NestJS độc lập, với cấu trúc gồm controllers (xử lý yêu cầu HTTP), services (chứa logic nghiệp vụ), modules (đóng gói thành phần liên quan) và entities (đại diện đối tượng dữ liệu). NestJS cung cấp module microservices chuyên dụng hỗ trợ các giao thức như TCP, Redis, MQTT, gRPC, và Kafka, giúp đơn giản hóa việc triển khai các mẫu giao tiếp.

TypeScript được chọn làm ngôn ngữ lập trình chính với ưu điểm hệ thống kiểu dữ liệu tĩnh, giúp phát hiện lỗi sớm, tăng cường khả năng đọc hiểu và bảo trì mã nguồn. Trong môi trường microservices, TypeScript giúp đảm bảo tính nhất quán của dữ liệu được truyền giữa các dịch vụ thông qua các contract rõ ràng.

Về lưu trữ dữ liệu, nguyên tắc "mỗi dịch vụ có cơ sở dữ liệu riêng" được tuân thủ. TypeORM được sử dụng để tương tác với cơ sở dữ liệu, hỗ trợ nhiều hệ quản trị và cung cấp tính năng như quan hệ, kế thừa, migrations. TypeORM sử dụng cả Active Record và Data Mapper, hỗ trợ lazy/eager loading, transactions và query builder để tối ưu hóa hiệu suất truy vấn.

PostgreSQL được chọn làm hệ quản trị cơ sở dữ liệu chính do tính ổn định, hiệu suất cao và hỗ trợ dữ liệu phức tạp (JSON, JSONB, arrays). Khả năng xử lý đồng thời và transaction của PostgreSQL đảm bảo tính nhất quán dữ liệu trong môi trường phân tán.

Về giao tiếp giữa microservices, khóa luận sử dụng nhiều công nghệ cho các mẫu giao tiếp khác nhau. HTTP/REST API là nền tảng cho giao tiếp đồng bộ, với Axios làm HTTP client. RabbitMQ được triển khai cho mẫu Point-to-Point và Asynchronous Request-Response, cung cấp cơ chế tin cậy cao với xác nhận tin nhắn và hàng đợi bền vững. Apache Kafka được sử dụng cho Publish/Subscribe và Event-Driven, nổi bật với khả năng xử lý hàng triệu sự kiện mỗi giây, độ trễ thấp và lưu trữ sự kiện lâu dài.

### 2.3.2. Các thông số đo lường chính

Để đánh giá hiệu năng của các mẫu giao tiếp, khóa luận xem xét một tập hợp thông số toàn diện. Latency (Độ trễ) là thông số quan trọng nhất, đại diện cho thời gian cần thiết để hoàn thành một yêu cầu, từ khi gửi đến khi nhận phản hồi [3]. Độ trễ được phân tích theo nhiều khía cạnh: độ trễ đầu cuối (tổng thời gian từ client đến phản hồi), độ trễ dịch vụ (thời gian xử lý trong một microservice) và độ trễ mạng (thời gian di chuyển dữ liệu giữa dịch vụ).

Throughput (Thông lượng) đo lường số lượng yêu cầu hệ thống xử lý trong một đơn vị thời gian, biểu thị bằng yêu cầu/giây (RPS) hoặc giao dịch/giây (TPS) [3]. Thông lượng được đo ở nhiều cấp độ: hệ thống, dịch vụ và endpoint. Các mẫu giao tiếp khác nhau ảnh hưởng đáng kể đến thông lượng - mẫu đồng bộ thường có thông lượng thấp hơn, mẫu bất đồng bộ có thể đạt thông lượng cao hơn nhờ xử lý song song.

Error Rate (Tỷ lệ lỗi) là tỷ lệ phần trăm yêu cầu thất bại so với tổng số yêu cầu [5]. Tỷ lệ lỗi bị ảnh hưởng bởi lỗi mạng, lỗi dịch vụ, timeout hoặc lỗi logic nghiệp vụ [4]. Mỗi loại lỗi (mạng, timeout, dịch vụ, logic) cần được phân loại và xử lý riêng biệt. Các mẫu giao tiếp khác nhau có cách tiếp cận khác nhau đối với xử lý lỗi, từ HTTP status codes đến dead-letter queues.

Resource Utilization (Sử dụng tài nguyên) đề cập đến lượng tài nguyên hệ thống (CPU, bộ nhớ, băng thông mạng) được sử dụng. Khóa luận giám sát sử dụng tài nguyên cho từng microservice và toàn hệ thống. Các mẫu giao tiếp đồng bộ thường có yêu cầu CPU/bộ nhớ thấp hơn nhưng nhiều kết nối mạng, mẫu bất đồng bộ có thể yêu cầu CPU/bộ nhớ cao hơn nhưng sử dụng mạng hiệu quả hơn.

Scalability (Khả năng mở rộng) đo lường khả năng xử lý tải tăng khi thêm tài nguyên. Khả năng mở rộng theo chiều ngang (thêm instance) thường được ưu tiên hơn chiều dọc (thêm tài nguyên cho instance hiện có). Các mẫu bất đồng bộ thường có khả năng mở rộng tốt hơn do tạo ít phụ thuộc trực tiếp giữa dịch vụ.

Consistency (Tính nhất quán) là khả năng duy trì trạng thái dữ liệu đồng bộ giữa các dịch vụ. Khóa luận đánh giá mức độ nhất quán dữ liệu đạt được bởi các mẫu giao tiếp khác nhau, từ tính nhất quán mạnh (strong consistency) đến nhất quán cuối cùng (eventual consistency).

### 2.3.3. Phương pháp đo lường

Khóa luận áp dụng nhiều phương pháp bổ sung nhau để thu thập dữ liệu hiệu năng [5]. Load Testing (Kiểm thử tải) mô phỏng điều kiện tải thực tế và đánh giá hiệu năng dưới áp lực [3]. Các kịch bản kiểm thử như kiểm tra tăng dần, chịu tải, phá vỡ và độ bền được thiết kế để mô phỏng trường hợp thực tế (tạo đơn hàng, kiểm tra tồn kho, xử lý thanh toán, gửi thông báo).

Benchmarking (Đánh giá) so sánh hiệu năng của các cấu hình hệ thống khác nhau trong điều kiện tiêu chuẩn [4]. Benchmark được tiến hành cho mỗi mẫu giao tiếp với các trường hợp thử nghiệm giống nhau, từ 10 đến 100 người dùng đồng thời. Các metric thu thập bao gồm thời gian phản hồi, thông lượng, tỷ lệ lỗi và sử dụng tài nguyên.



Profiling (Lập hồ sơ) phân tích chi tiết tài nguyên và thời gian thực thi của các thành phần. Trong Node.js, profiling thực hiện bằng công cụ như Node.js Profiler hoặc clinic.js. Khóa luận sử dụng profiling để phân tích thời gian cho serialization/deserialization, xử lý mạng, logic nghiệp vụ và tương tác database.

Distributed Tracing (Theo dõi phân tán) theo dõi yêu cầu qua nhiều dịch vụ, xác định điểm nghẽn và mối quan hệ phụ thuộc. OpenTelemetry được tích hợp với NestJS thông qua interceptors và middleware. Mỗi trace đại diện cho một yêu cầu và gồm nhiều spans (hoạt động đơn lẻ như HTTP request, database query).

Metrics Collection (Thu thập số liệu) thu thập và phân tích chỉ số hiệu năng theo thời gian. Khóa luận thu thập HTTP metrics, microservice metrics, database metrics, message broker metrics và system metrics, lưu trữ trong time-series database để phân tích xu hướng và thiết lập cảnh báo.

### 2.3.4. Công cụ đo lường hiệu năng

Để thực hiện các phương pháp trên, khóa luận triển khai bộ công cụ toàn diện [6]. K6, công cụ kiểm thử tải mã nguồn mở dựa trên JavaScript, tạo tải và đo lường hiệu năng [3]. K6 cho phép viết kịch bản phức tạp mô phỏng hành vi thực tế, hỗ trợ HTTP, WebSocket và gRPC, với khả năng mở rộng và tùy chỉnh cao.

Prometheus, hệ thống giám sát mã nguồn mở, thu thập và lưu trữ số liệu hiệu năng từ microservices [4]. Prometheus sử dụng mô hình pull để truy vấn định kỳ các mục tiêu được cấu hình [5], cung cấp ngôn ngữ truy vấn PromQL và hệ thống cảnh báo mạnh mẽ. Các microservices được cấu hình để hiển thị endpoint metrics (/metrics) mà Prometheus truy vấn mỗi 15 giây.

Kết hợp các công nghệ triển khai và công cụ đo lường này tạo môi trường toàn diện để đánh giá và so sánh các mẫu giao tiếp. Thông qua thu thập và phân tích dữ liệu từ nhiều góc độ, khóa luận cung cấp cái nhìn sâu sắc về ưu nhược điểm của mỗi mẫu và đưa ra khuyến nghị dựa trên bằng chứng cho việc lựa chọn mẫu phù hợp trong từng tình huống.

## 2.4. Tổng kết

Chương này đã cung cấp một cái nhìn tổng quan về kiến trúc vi dịch vụ và vai trò quan trọng của giao tiếp trong kiến trúc này. Chúng ta đã thảo luận về các đặc điểm chính của vi dịch vụ, so sánh với kiến trúc nguyên khối, và xem xét các lợi ích cũng như thách thức.

Về mặt đặc điểm, vi dịch vụ là một kiến trúc phân tán, trong đó mỗi dịch vụ tự trị, tập trung vào một chức năng nghiệp vụ cụ thể, quản lý dữ liệu riêng của nó, được thiết kế để xử lý lỗi, và có thể phát triển độc lập. So với kiến trúc nguyên khối, microservice cung cấp khả năng mở rộng có mục tiêu, phát triển nhanh hơn, tính linh hoạt công nghệ, khả năng chịu lỗi tốt hơn, và khả năng bảo trì và hiểu biết tốt hơn.

Tuy nhiên, microservice cũng đặt ra một số thách thức, bao gồm độ phức tạp phân tán, giao tiếp giữa các dịch vụ, quản lý dữ liệu, vận hành và giám sát, và kiểm thử. Để giải quyết những thách thức này, một số nguyên tắc thiết kế nên được tuân thủ, bao gồm Single Responsibility Principle, Domain-Driven Design, API First, tự động hóa, Monitoring và Observability, và Fault Tolerance.

Trong phần về giao tiếp, chúng ta đã khám phá vai trò quan trọng của giao tiếp trong kiến trúc vi dịch vụ, bao gồm tạo điều kiện cho sự hợp tác giữa các dịch vụ, đảm bảo tính nhất quán dữ liệu, hỗ trợ khả năng chịu lỗi, và cho phép tính mở rộng. Chúng ta cũng đã thảo luận về các thuộc tính quan trọng của giao tiếp microservice, bao gồm độ tin cậy, độ trễ, khả năng mở rộng, cách ly lỗi, tính nhất quán, định dạng dữ liệu, khả năng tương tác, và bảo mật.

Hai mô hình giao tiếp cơ bản trong microservices là đồng bộ và bất đồng bộ. Trong giao tiếp đồng bộ, người gửi đợi phản hồi từ người nhận, trong khi trong giao tiếp bất đồng bộ, người gửi không đợi phản hồi. Ngoài ra, các vi dịch vụ cũng giao tiếp theo các kiểu tương tác khác nhau, bao gồm one-to-one, one-to-many, many-to-one, và many-to-many.

Chúng ta đã đi sâu vào các mẫu giao tiếp cụ thể trong vi dịch vụ, bao gồm Request-Response, Event-Driven, Publish-Subscribe, Point-to-Point Messaging, và Asynchronous Request-Response. Mỗi mẫu có ưu và nhược điểm riêng, thích hợp cho các tình huống và yêu cầu khác nhau.

Trong phần về công nghệ triển khai, chúng ta đã khám phá các công nghệ hiện đại được sử dụng để xây dựng hệ thống vi dịch vụ. NestJS được giới thiệu như một framework mạnh mẽ dựa trên Node.js và TypeScript, cung cấp kiến trúc mô-đun và hỗ trợ tích hợp cho nhiều phương thức giao tiếp. TypeScript mang lại lợi thế với hệ thống kiểu dữ liệu tĩnh, giúp phát hiện lỗi sớm và cải thiện khả năng bảo trì code. Về phần cơ sở dữ liệu, TypeORM được sử dụng như một ORM hiện đại kết hợp với PostgreSQL để lưu trữ dữ liệu một cách hiệu quả.

Các công nghệ giao tiếp được trình bày bao gồm HTTP/REST với Axios cho giao tiếp đồng bộ, RabbitMQ cho các mẫu Point-to-Point và Asynchronous Request-Response, và Apache Kafka cho các mẫu Publish/Subscribe và Event-Driven. Mỗi công nghệ có đặc điểm và ưu điểm riêng, phù hợp với các mẫu giao tiếp tương ứng.

Trong phần về đo lường hiệu năng, chúng ta đã thảo luận chi tiết về các thông số đo lường chính, bao gồm Latency, Throughput, Error Rate, Resource Utilization, Scalability và Consistency. Mỗi thông số được phân tích theo nhiều khía cạnh và ý nghĩa của chúng trong context của vi dịch vụ, cùng với sự ảnh hưởng của các mẫu giao tiếp khác nhau đến các thông số này.

Chúng ta cũng đã khám phá chi tiết các phương pháp đo lường, bao gồm Load Testing với các phương pháp như kiểm thử tăng dần, kiểm thử chịu tải, kiểm thử phá vỡ và kiểm thử độ bền. Benchmarking được trình bày như một phương pháp so sánh hiệu năng dưới các điều kiện tiêu chuẩn. Profiling giúp phân tích chi tiết về tài nguyên sử dụng và thời

gian thực thi. Distributed Tracing theo dõi yêu cầu xuyên suốt các dịch vụ, và Metrics Collection thu thập và phân tích các chỉ số hiệu năng theo thời gian.

Về công cụ đo lường, K6 được giới thiệu như một công cụ kiểm thử tải mạnh mẽ, cho phép viết các tập lệnh kiểm thử phức tạp bằng JavaScript. Prometheus là hệ thống giám sát mở rộng cao và đáng tin cậy, sử dụng mô hình pull để thu thập số liệu và cung cấp ngôn ngữ truy vấn PromQL mạnh mẽ.

Các khái niệm và hiểu biết từ chương này sẽ làm nền tảng cho các chương tiếp theo, nơi chúng ta sẽ đi sâu vào việc phân tích chi tiết các mẫu giao tiếp cụ thể trong kiến trúc vi dịch vụ. Chúng ta sẽ phân loại các mẫu này theo tiêu chí đồng bộ/bất đồng bộ và one-to-one/one-to-many, phân tích ưu và nhược điểm của từng mẫu, và cung cấp hướng dẫn cho việc lựa chọn mẫu phù hợp cho các tình huống cụ thể.

Việc hiểu rõ các khái niệm cơ bản, công nghệ triển khai và phương pháp đo lường hiệu năng sẽ giúp chúng ta đánh giá tốt hơn hiệu quả của các mẫu giao tiếp trong các kịch bản thực tế. Các công nghệ như NestJS, TypeORM, RabbitMQ và Kafka, cùng với các công cụ đo lường như K6 và Prometheus, sẽ được ứng dụng trong các phần tiếp theo để triển khai và đánh giá các mẫu giao tiếp trong môi trường thực tế.

Tóm lại, chương này đã cung cấp một cái nhìn toàn diện về kiến trúc vi dịch vụ, vai trò quan trọng của giao tiếp, các công nghệ triển khai hiện đại và phương pháp đo lường hiệu năng. Những kiến thức này sẽ là nền tảng vững chắc cho các phân tích chi tiết hơn trong các chương tiếp theo, giúp chúng ta hiểu rõ hơn về cách triển khai và đánh giá các mẫu giao tiếp trong kiến trúc microservice.

## Chương 3.

# Phân tích các mẫu giao tiếp

### 3.1. Cách phân loại các mẫu giao tiếp

Trong kiến trúc vi dịch vụ, giao tiếp giữa các dịch vụ đóng vai trò quan trọng trong việc đảm bảo hiệu suất, khả năng mở rộng và độ tin cậy của hệ thống [4]. Việc thiết kế giao tiếp giữa các dịch vụ là một trong những thách thức quan trọng nhất, ảnh hưởng trực tiếp đến kiến trúc tổng thể và hiệu quả hoạt động của hệ thống. Để hiểu rõ và áp dụng hiệu quả các mẫu giao tiếp, việc phân loại chúng một cách có hệ thống là cần thiết.

Các mẫu giao tiếp trong vi dịch vụ có thể được phân loại dựa trên hai tiêu chí chính: phương thức giao tiếp (communication mode) và phạm vi giao tiếp (communication scope). Phương thức giao tiếp xác định cách thức tương tác giữa các dịch vụ, có thể là đồng bộ (synchronous) hoặc bất đồng bộ (asynchronous). Phạm vi giao tiếp xác định số lượng người nhận trong một lần giao tiếp, phổ biến nhất là one-to-one và one-to-many. Bảng 3.1 dưới đây tổng hợp các mẫu giao tiếp phổ biến dựa trên sự kết hợp của hai tiêu chí này.

**Bảng 3.1. So sánh các kiểu tương tác trong giao tiếp vi dịch vụ**

	one-to-one	one-to-many
Synchronous	Request/Response	—
Asynchronous	Asynchronous Request/Response One-way Notifications	Publish/Subscribe Publish/Async responses

#### 3.1.1. Tiêu chí phân loại theo communication mode (sync/async)

Tiêu chí đầu tiên và cơ bản nhất trong việc phân loại các mẫu giao tiếp là dựa trên mode giao tiếp: đồng bộ (synchronous) và bất đồng bộ (asynchronous). Sự phân biệt này liên quan đến cách các dịch vụ tương tác và đợi phản hồi từ nhau.

Trong giao tiếp đồng bộ, dịch vụ gửi yêu cầu chặn (block) quá trình xử lý của nó và đợi cho đến khi nhận được phản hồi từ dịch vụ nhận. Mô hình này tạo ra một sự phụ thuộc trực tiếp về thời gian giữa người gửi và người nhận, đồng thời đặt ra yêu cầu cả hai dịch vụ phải đồng thời hoạt động để hoàn thành giao tiếp [5]. Giao tiếp đồng bộ giống như một "cuộc gọi điện thoại" - người gọi phải đợi người nhận trả lời và hoàn thành cuộc trò chuyện trước khi có thể tiếp tục các hoạt động khác.

Các giao thức đồng bộ phổ biến trong vi dịch vụ bao gồm HTTP/REST, gRPC và SOAP. Giao tiếp đồng bộ thường được triển khai thông qua các API endpoint, với dịch vụ gọi gửi yêu cầu HTTP và đợi phản hồi. Mặc dù giao tiếp đồng bộ có ưu điểm về tính đơn giản và dễ hiểu, nó cũng tạo ra coupling chặt chẽ giữa các dịch vụ và có thể dẫn đến hiệu suất kém trong môi trường phân tán.

Ngược lại, trong giao tiếp bất đồng bộ, dịch vụ gửi không chặn quá trình xử lý của nó khi đợi phản hồi. Thay vào đó, nó tiếp tục thực hiện các tác vụ khác và xử lý phản hồi (nếu cần) khi phản hồi đó đến. Giao tiếp bất đồng bộ giống như "gửi email" - người gửi không cần đợi người nhận đọc và trả lời ngay lập tức.

Các công nghệ bất đồng bộ phổ biến bao gồm message brokers như RabbitMQ, Apache Kafka và Amazon SQS. Các công nghệ này cho phép các dịch vụ giao tiếp thông qua việc gửi và nhận tin nhắn mà không cần đồng thời hoạt động. Giao tiếp bất đồng bộ thúc đẩy sự tách rời (decoupling) giữa các dịch vụ, cải thiện khả năng chịu lỗi và mở rộng, nhưng cũng làm tăng độ phức tạp của hệ thống và khó khăn trong việc theo dõi luồng yêu cầu.

Trong một đánh giá hệ thống về các mẫu giao tiếp và triển khai trong kiến trúc vi dịch vụ, các mẫu đồng bộ như Request-Response thường có thông lượng thấp hơn do tính chất tuần tự của chúng, trong khi các mẫu bất đồng bộ như Publish/Subscribe và Event-Driven có thể đạt thông lượng cao hơn do khả năng xử lý song song [6].

### **3.1.2. Tiêu chí phân loại theo communication scope (one-to-one/one-to-many)**

Tiêu chí phân loại thứ hai là dựa trên phạm vi giao tiếp, tức là số lượng người nhận tham gia vào quá trình nhận thông điệp. Trong phạm vi khóa luận này, chúng ta tập trung vào hai loại chính là one-to-one và one-to-many, đây là hai mô hình giao tiếp phổ biến nhất trong kiến trúc vi dịch vụ.

Giao tiếp one-to-one là mô hình trong đó một dịch vụ gửi thông điệp đến chính xác một dịch vụ khác và chỉ dịch vụ đó nhận được thông điệp. Mô hình này phù hợp cho các tương tác yêu cầu-phản hồi trực tiếp giữa hai dịch vụ, chẳng hạn như khi một dịch vụ cần truy vấn dữ liệu từ dịch vụ khác [4]. Trong mô hình này, người gửi biết rõ người nhận và thường mong đợi một phản hồi. Các mẫu giao tiếp one-to-one phổ biến bao gồm Request-Response (đồng bộ), Asynchronous Request-Response và One-way Notifications (bất đồng bộ).

Giao tiếp one-to-many là mô hình trong đó một dịch vụ gửi thông điệp đến nhiều dịch vụ khác cùng một lúc. Mô hình này thích hợp cho việc phát tán thông tin hoặc thông báo sự kiện trong hệ thống. Ví dụ, khi một dịch vụ muốn thông báo về sự thay đổi trạng thái mà có thể ảnh hưởng đến nhiều dịch vụ khác. Trong mô hình này, người gửi thường không biết hoặc không quan tâm đến việc ai sẽ nhận thông điệp của mình. Các mẫu giao tiếp one-to-many điển hình bao gồm Publish/Subscribe và Publish/Async responses, tất cả đều là các mẫu bất đồng bộ.

Sự kết hợp giữa hai tiêu chí phân loại - communication mode (sync/async) và communication scope (one-to-one/one-to-many) - tạo ra một ma trận phân loại các mẫu giao tiếp. Trong ma trận này, có một điểm đáng chú ý là trong thực tế, hầu như không có mẫu giao tiếp đồng bộ one-to-many được sử dụng rộng rãi. Điều này hoàn toàn hợp lý vì việc một dịch vụ đồng thời gửi yêu cầu đến nhiều dịch vụ khác và đợi tất cả phản hồi sẽ tạo ra bottleneck về hiệu suất và tăng khả năng lỗi cascade. Do đó, khi cần giao tiếp one-to-many, các kiến trúc vi dịch vụ hiện đại gần như luôn ưu tiên sử dụng các mẫu bất đồng bộ.

Phạm vi giao tiếp có ảnh hưởng lớn đến sự phức tạp, khả năng mở rộng và khả năng bảo trì của hệ thống. Các mẫu giao tiếp one-to-many thường phức tạp hơn để triển khai và quản lý so với one-to-one, nhưng cung cấp khả năng tách rời (decoupling) tốt hơn và khả năng mở rộng cao hơn. Ví dụ, với mẫu Publish/Subscribe, việc thêm người đăng ký (subscriber) mới không đòi hỏi bất kỳ thay đổi nào từ phía nhà xuất bản (publisher), điều này tạo ra sự linh hoạt cao trong việc mở rộng hệ thống.

### **3.1.3. Các yếu tố ảnh hưởng đến việc lựa chọn pattern**

Việc lựa chọn mẫu giao tiếp phù hợp cho một hệ thống vi dịch vụ phụ thuộc vào nhiều yếu tố. Hiểu rõ các yếu tố này sẽ giúp kiến trúc sư và nhà phát triển đưa ra quyết định đúng đắn, cân bằng giữa hiệu suất, độ tin cậy, độ phức tạp và khả năng mở rộng.

Yêu cầu về độ trễ (latency) là một trong những yếu tố quan trọng nhất ảnh hưởng đến việc lựa chọn mẫu giao tiếp. Các ứng dụng đòi hỏi thời gian phản hồi nhanh, như các ứng dụng giao diện người dùng, thường ưu tiên giao tiếp đồng bộ [4]. Trong khi đó, các quy trình xử lý nền không cần phản hồi tức thì có thể hưởng lợi từ giao tiếp bất đồng bộ. Giao tiếp đồng bộ đơn giản hơn để hiểu và triển khai, nhưng có thể dẫn đến độ trễ cao khi có nhiều cuộc gọi nối tiếp. Giao tiếp bất đồng bộ giảm thiểu độ trễ cảm nhận được bằng cách cho phép máy khách tiếp tục hoạt động trong khi yêu cầu được xử lý, nhưng làm tăng độ phức tạp của hệ thống [5].

Yêu cầu về khả năng mở rộng (scalability) cũng là một yếu tố quyết định. Các hệ thống cần xử lý khối lượng lớn yêu cầu thường ưu tiên các mẫu giao tiếp bất đồng bộ, đặc biệt là các mẫu như Publish/Subscribe và Event-Driven. Các mẫu bất đồng bộ như Publish/Subscribe và Event-Driven thường có khả năng mở rộng tốt hơn do chúng tách rời các producer và consumer, cho phép chúng mở rộng độc lập [6].

Độ tin cậy và khả năng chịu lỗi là các yếu tố không thể bỏ qua. Các mẫu giao tiếp bất đồng bộ thường cung cấp khả năng chịu lỗi tốt hơn so với các mẫu đồng bộ, vì chúng có thể xử lý các tình huống như dịch vụ không khả dụng tạm thời [8]. Các mẫu như Point-to-Point Messaging với đảm bảo giao hàng và Asynchronous Request-Response với cơ chế retry tích hợp có thể cải thiện đáng kể độ tin cậy của hệ thống.

Độ phức tạp triển khai và bảo trì cũng là một yếu tố quan trọng. Giao tiếp đồng bộ thường đơn giản hơn để triển khai và gỡ lỗi, trong khi giao tiếp bất đồng bộ đòi hỏi cơ sở hạ tầng phức tạp hơn (như message brokers) và logic xử lý phức tạp hơn để đảm bảo tin cậy [5].

Tính nhất quán dữ liệu là một yếu tố đặc biệt quan trọng trong các hệ thống phân tán. Việc duy trì tính nhất quán dữ liệu giữa các dịch vụ có thể là một thách thức, đặc biệt với giao tiếp bất đồng bộ [4]. Các mẫu như Saga và Event Sourcing đã được phát triển để giải quyết vấn đề này bằng cách cung cấp các cơ chế để quản lý giao dịch phân tán và đảm bảo tính nhất quán cuối cùng.

Yêu cầu nghiệp vụ cụ thể cũng đóng vai trò quan trọng trong việc lựa chọn mẫu giao tiếp. Ví dụ, các quy trình nghiệp vụ đòi hỏi phản hồi ngay lập tức, như xử lý thanh toán trực tuyến, có thể yêu cầu giao tiếp đồng bộ. Trong khi đó, các quy trình như phân tích dữ liệu hoặc gửi email thông báo có thể phù hợp hơn với giao tiếp bất đồng bộ.

Cuối cùng, tính linh hoạt của thiết kế và khả năng phát triển trong tương lai cũng cần được cân nhắc. Tầm quan trọng của việc thiết kế hệ thống có thể thích nghi với các yêu cầu thay đổi không thể phủ nhận [5]. Các mẫu giao tiếp bất đồng bộ, đặc biệt là Event-Driven và Publish/Subscribe, thường cung cấp tính linh hoạt cao hơn bằng cách cho phép thêm dịch vụ mới mà không cần thay đổi các dịch vụ hiện có.

Việc hiểu và cân nhắc các yếu tố này là rất quan trọng để lựa chọn mẫu giao tiếp phù hợp cho một hệ thống vi dịch vụ cụ thể. Thông thường, một hệ thống vi dịch vụ hiệu quả sẽ kết hợp nhiều mẫu giao tiếp khác nhau để đáp ứng các yêu cầu khác nhau của các phần khác nhau trong hệ thống.

## **3.2. Synchronous Communication Patterns (one-to-one)**

Giao tiếp đồng bộ là hình thức tương tác phổ biến nhất giữa các microservice, đặc biệt trong các mô hình one-to-one. Đây là mô hình được ứng dụng rộng rãi nhờ tính đơn giản và dễ triển khai của nó [5]. Trong phần này, chúng ta sẽ tìm hiểu sâu về Request/Response - mẫu giao tiếp đồng bộ cơ bản nhất trong kiến trúc microservices, cùng các công nghệ triển khai phổ biến và những ứng dụng thực tế của nó.

### **3.2.1. Request/Response Pattern**

Request/Response là mẫu giao tiếp đồng bộ cơ bản nhất, trong đó một microservice (client) gửi yêu cầu đến một microservice khác (server) và chờ đợi cho đến khi nhận được phản hồi. Quá trình giao tiếp này tạo ra một sự ràng buộc tạm thời giữa hai dịch vụ, buộc client phải chờ đợi server xử lý và phản hồi trước khi tiếp tục các hoạt động khác.

Mẫu Request/Response tuân theo một quy trình cơ bản: client microservice xác định yêu cầu cần thực hiện và endpoint của server microservice; gửi yêu cầu đến server microservice, thường thông qua một giao thức như HTTP; server microservice nhận yêu cầu, xử lý nó và chuẩn bị phản hồi; server gửi phản hồi trở lại client microservice; và cuối cùng client nhận phản hồi và tiếp tục quá trình xử lý của mình. Trong suốt quá trình này, luồng thực thi của client microservice bị chặn cho đến khi nhận được phản hồi từ server microservice, đây chính là đặc điểm định nghĩa tính đồng bộ của mẫu giao tiếp này [5].

Các công nghệ triển khai Request/Response phổ biến nhất bao gồm REST, gRPC, GraphQL, và SOAP. Trong đó, REST (Representational State Transfer) là phương pháp phổ biến nhất do tính đơn giản và khả năng tương thích rộng rãi của nó.

#### REST (Representational State Transfer)

REST là một kiến trúc giao tiếp dựa trên HTTP, sử dụng các phương thức HTTP chuẩn (GET, POST, PUT, DELETE) để thực hiện các thao tác CRUD (Create, Read, Update, Delete) trên tài nguyên. REST đã trở thành tiêu chuẩn de facto cho việc xây dựng API web và giao tiếp giữa các dịch vụ trên Internet.

Trong kiến trúc microservices, REST API thường được sử dụng để triển khai mẫu Request/Response bởi những ưu điểm về tính đơn giản và dễ hiểu, tính stateless (mỗi yêu cầu chứa đầy đủ thông tin cần thiết, không phụ thuộc vào yêu cầu trước đó), khả năng mở rộng tốt (nhờ tính chất stateless, REST API có thể mở rộng dễ dàng bằng cách thêm nhiều instance server), và tính tương thích cao (REST có thể được triển khai trên nhiều nền tảng và ngôn ngữ lập trình khác nhau).

Một tương tác REST điển hình trong kiến trúc microservices có thể bao gồm một yêu cầu GET đến một endpoint như `/api/products/123` để lấy thông tin về một sản phẩm cụ thể. Server sẽ phản hồi với mã trạng thái HTTP (như 200 OK cho thành công) và dữ liệu sản phẩm, thường ở định dạng JSON.

Mặc dù REST là phương pháp triển khai phổ biến nhất cho Request/Response, nó cũng có một số hạn chế như vấn đề overloading nghiêm trọng (REST thường dẫn đến overfetching hoặc underfetching), thiếu kiểm soát contract (REST không có cơ chế tích hợp để định nghĩa và kiểm tra contract giữa client và server), và hiệu suất không tối ưu (HTTP là một giao thức text-based, không hiệu quả bằng các giao thức binary).

#### gRPC (Google Remote Procedure Call)

gRPC là một framework RPC hiệu suất cao được phát triển bởi Google, sử dụng Protocol Buffers làm ngôn ngữ định nghĩa interface và HTTP/2 làm giao thức truyền tải. gRPC cung cấp một cách hiệu quả hơn để triển khai mẫu Request/Response trong kiến trúc microservices [10].

So với REST, gRPC có nhiều ưu điểm đáng kể về hiệu suất (gRPC sử dụng Protocol Buffers, một định dạng nhị phân hiệu quả hơn JSON hoặc XML), service contract rõ ràng (gRPC sử dụng .proto files để định nghĩa service interface, giúp tạo ra client và server code tự động), hỗ trợ streaming (gRPC hỗ trợ cả streaming đơn hướng và hai chiều), và hỗ trợ đa ngôn ngữ (gRPC tạo code cho nhiều ngôn ngữ lập trình từ cùng một định nghĩa).

Trong gRPC, dịch vụ được định nghĩa trong file .proto với các messages (cấu trúc dữ liệu) và services (các phương thức RPC). Từ file này, gRPC sẽ tự động tạo ra code client và server cho nhiều ngôn ngữ lập trình khác nhau. Quá trình giao tiếp diễn ra thông qua HTTP/2, cung cấp nhiều tính năng như multiplexing, header compression, và full-duplex communication.

Mặc dù gRPC mang lại nhiều lợi ích, nó cũng có một số hạn chế như độ phức tạp cao hơn (gRPC đòi hỏi hiểu biết về Protocol Buffers và HTTP/2), thiếu hỗ trợ trình duyệt



(hầu hết các trình duyệt web không hỗ trợ trực tiếp gRPC), và khó khăn trong debug (do sử dụng định dạng nhị phân, việc debug gRPC phức tạp hơn so với REST).

### GraphQL

GraphQL là một ngôn ngữ truy vấn và thao tác dữ liệu được phát triển bởi Facebook. Trong kiến trúc microservices, GraphQL thường được sử dụng như một lớp API gateway để tổng hợp dữ liệu từ nhiều microservices và cho phép client chỉ định chính xác dữ liệu họ cần [11].

GraphQL có một số ưu điểm đáng chú ý so với REST và gRPC như tránh được overfetching và underfetching (client có thể chỉ định chính xác dữ liệu cần thiết), sử dụng một endpoint duy nhất cho tất cả các truy vấn, khả năng introspection (GraphQL API tự mô tả, cho phép tạo tài liệu tự động), và hỗ trợ tốt cho frontend (các thư viện client như Apollo và Relay cung cấp tích hợp mạnh mẽ với các framework frontend).

Trong GraphQL, client gửi một truy vấn (query) xác định chính xác dữ liệu mà họ cần. Ví dụ, thay vì gọi nhiều endpoint REST khác nhau để lấy thông tin về sản phẩm, đánh giá và người dùng, client có thể gửi một truy vấn GraphQL duy nhất chỉ định tất cả thông tin cần thiết. Server GraphQL chịu trách nhiệm thu thập dữ liệu từ các nguồn khác nhau (có thể là các microservices khác) và trả về chính xác những gì client yêu cầu.

Mặc dù GraphQL mang lại nhiều lợi ích, nó cũng có những thách thức như độ phức tạp cao (triển khai GraphQL API đòi hỏi kiến thức chuyên sâu và nhiều boilerplate code), caching phức tạp hơn (do sử dụng một endpoint duy nhất, caching với GraphQL phức tạp hơn so với REST), và vấn đề N+1 truy vấn (GraphQL có thể dẫn đến vấn đề N+1 truy vấn nếu không được triển khai cẩn thận).

### 3.2.2. Ưu điểm và Hạn chế của Synchronous Communication

Giao tiếp đồng bộ trong kiến trúc microservices có nhiều ưu điểm và hạn chế đáng chú ý. Hiểu rõ những điểm mạnh và điểm yếu này là chìa khóa để lựa chọn mẫu giao tiếp phù hợp cho từng trường hợp sử dụng cụ thể.

Về ưu điểm, giao tiếp đồng bộ mang lại tính đơn giản và trực quan, làm cho nó dễ hiểu và triển khai, đặc biệt là với REST. Client gửi yêu cầu và nhận phản hồi theo một cách rõ ràng và trực tiếp, không đòi hỏi hiểu biết về các khái niệm phức tạp như message broker hay event handling. Ưu điểm thứ hai là phản hồi tức thì, client nhận được phản hồi ngay lập tức sau khi server xử lý yêu cầu, điều này quan trọng cho các tương tác yêu cầu tốc độ như truy vấn dữ liệu trong giao diện người dùng. Thứ ba, giao tiếp đồng bộ cho phép xử lý lỗi một cách dễ dàng, lỗi được phát hiện và xử lý ngay lập tức trong quy trình giao tiếp, giúp client có thể phản hồi lập tức cho người dùng hoặc thử lại yêu cầu. Ngoài ra, giao tiếp đồng bộ còn mang lại tính nhất quán cao, client biết ngay lập tức kết quả của yêu cầu, tạo điều kiện cho tính nhất quán dữ liệu giữa các dịch vụ. Cuối cùng, giao tiếp đồng bộ dễ dàng debug, luồng yêu cầu-phản hồi rõ ràng và dễ theo dõi, giúp nhà phát triển nhanh chóng xác định và sửa lỗi.

Tuy nhiên, giao tiếp đồng bộ cũng có những hạn chế đáng kể. Đầu tiên là coupling chặt chẽ, client và server phải đồng thời hoạt động để giao tiếp thành công, nếu server

không khả dụng, client sẽ không thể tiếp tục hoạt động. Hạn chế thứ hai là hiệu suất kém trong môi trường phân tán, độ trễ mạng có thể ảnh hưởng đáng kể đến hiệu suất, đặc biệt là khi có nhiều cuộc gọi nối tiếp. Giao tiếp đồng bộ cũng thiếu khả năng mở rộng, nó có thể tạo thành bottleneck khi số lượng yêu cầu tăng cao, hạn chế khả năng mở rộng của hệ thống. Một hạn chế nghiêm trọng khác là khả năng gây ra lỗi cascade, khi một service gặp sự cố, các service gọi đến nó sẽ bị chặn, và lỗi có thể lan truyền trong toàn hệ thống, gây ra hiệu ứng domino. Cuối cùng, giao tiếp đồng bộ khiến tài nguyên bị chặn, client phải chờ đợi phản hồi trước khi tiếp tục, dẫn đến lãng phí tài nguyên và giảm throughput.

Để giảm thiểu những hạn chế này, nhiều mẫu thiết kế và kỹ thuật đã được phát triển và áp dụng trong giao tiếp đồng bộ microservices. Ví dụ, để giải quyết vấn đề lỗi cascade, các kỹ thuật như circuit breaker pattern có thể được sử dụng. Để cải thiện hiệu suất, các kỹ thuật như caching, connection pooling, và batch processing có thể được áp dụng.

### **3.2.3. Use cases phù hợp cho Synchronous Communication**

Giao tiếp đồng bộ đặc biệt phù hợp cho một số trường hợp sử dụng cụ thể trong kiến trúc microservices. Hiểu rõ những trường hợp này giúp kiến trúc sư và nhà phát triển đưa ra quyết định đúng đắn về việc khi nào nên sử dụng giao tiếp đồng bộ.

Trường hợp sử dụng đầu tiên và quan trọng nhất là khi yêu cầu phản hồi tức thì. Trong nhiều tình huống, client cần phản hồi ngay lập tức từ server để tiếp tục xử lý hoặc cung cấp phản hồi cho người dùng. Ví dụ, khi người dùng thực hiện thanh toán, hệ thống cần kiểm tra ngay lập tức tính khả dụng của sản phẩm và xác thực thanh toán trước khi hoàn tất đơn hàng. Giao tiếp đồng bộ là lựa chọn tự nhiên cho những tình huống này do khả năng cung cấp phản hồi tức thì.

Trường hợp thứ hai là truy vấn dữ liệu đơn giản. Các hoạt động CRUD (Create, Read, Update, Delete) cơ bản mà không yêu cầu xử lý phức tạp thường phù hợp với giao tiếp đồng bộ. Ví dụ, khi ứng dụng cần lấy thông tin chi tiết về một sản phẩm để hiển thị cho người dùng, một cuộc gọi API REST đơn giản là đủ và hiệu quả.

Trường hợp thứ ba là khi tính nhất quán dữ liệu quan trọng. Trong một số quy trình nghiệp vụ, client cần biết ngay lập tức liệu hoạt động đã thành công hay không để đảm bảo tính nhất quán dữ liệu. Ví dụ, khi cập nhật thông tin quan trọng như số dư tài khoản hoặc trạng thái đặt phòng, hệ thống cần đảm bảo cập nhật đã được thực hiện thành công trước khi tiếp tục các hoạt động khác.

Trường hợp thứ tư là tương tác người dùng trực tiếp. Khi hành động của người dùng trực tiếp kích hoạt yêu cầu, và người dùng đang chờ kết quả, giao tiếp đồng bộ thường là lựa chọn tốt nhất. Điều này đảm bảo người dùng nhận được phản hồi ngay lập tức, cải thiện trải nghiệm người dùng. Ví dụ, khi người dùng nhấp vào nút "Thêm vào giỏ hàng", hệ thống cần xác nhận ngay lập tức rằng sản phẩm đã được thêm thành công.

Trường hợp cuối cùng là quy trình nghiệp vụ đơn giản. Các quy trình không yêu cầu xử lý phức tạp hoặc nhiều bước thường phù hợp với giao tiếp đồng bộ. Quy trình càng đơn giản, lợi ích của giao tiếp bất đồng bộ càng ít, và chi phí phức tạp bổ sung càng khó được biện minh.

Fowler [8] lưu ý rằng giao tiếp đồng bộ thường được ưu tiên trong các hệ thống ban đầu và những hệ thống có quy mô nhỏ, sau đó có thể phát triển thành mô hình bất đồng bộ khi hệ thống phát triển hoặc yêu cầu về khả năng mở rộng tăng lên. Điều này phản ánh sự đánh đổi giữa tính đơn giản của giao tiếp đồng bộ và khả năng mở rộng của giao tiếp bất đồng bộ.

### 3.2.4. Case studies

Để hiểu rõ hơn về cách giao tiếp đồng bộ được triển khai và sử dụng trong thế giới thực, chúng ta sẽ xem xét một số trường hợp nghiên cứu từ các công ty nổi tiếng đã thành công với kiến trúc microservices.

Netflix là một ví dụ điển hình về việc sử dụng giao tiếp đồng bộ trong kiến trúc microservices quy mô lớn. Netflix sử dụng Zuul làm API Gateway để quản lý các yêu cầu từ nhiều client khác nhau đến hàng trăm microservices backend [6]. Zuul cung cấp một điểm vào duy nhất cho tất cả các client, thực hiện các chức năng như routing, filtering, và load balancing. Bên cạnh Zuul, Netflix cũng sử dụng Hystrix để triển khai Circuit Breaker pattern, bảo vệ hệ thống khỏi lỗi cascade và cải thiện khả năng phục hồi. Netflix đã phát triển một hệ sinh thái các công cụ để hỗ trợ kiến trúc microservices của họ, bao gồm Eureka (service discovery), Ribbon (client-side load balancing), và Feign (declarative HTTP client). Tất cả các công cụ này đều được thiết kế để hoạt động hiệu quả trong môi trường đám mây và xử lý các thách thức của giao tiếp đồng bộ giữa các microservices.

Amazon là một ví dụ khác về việc sử dụng giao tiếp đồng bộ trong kiến trúc microservices quy mô lớn. Amazon sử dụng AWS API Gateway để quản lý và định tuyến các yêu cầu API đến các microservices backend khác nhau [7]. AWS API Gateway cung cấp nhiều tính năng như authentication, authorization, rate limiting, và caching. Nó cũng tích hợp với các dịch vụ AWS khác như Lambda, DynamoDB, và SNS, cho phép xây dựng kiến trúc serverless kết hợp với microservices. Bên cạnh đó, Amazon cũng sử dụng các service mesh như AWS App Mesh để quản lý giao tiếp giữa các microservices, cung cấp các tính năng như service discovery, traffic routing, và observation.

Uber là một ví dụ về công ty đã chuyển đổi từ REST sang gRPC để cải thiện hiệu suất giao tiếp giữa các microservices [12]. Kiến trúc của Uber bao gồm hàng nghìn microservices với hàng triệu yêu cầu mỗi giây. Uber đã phát triển một framework gọi là Ringpop, sử dụng gRPC để giao tiếp giữa các dịch vụ. Ringpop sử dụng giao thức gossip để phân phối thông tin về trạng thái của các node trong hệ thống và đưa ra quyết định về việc định tuyến yêu cầu. Uber cũng đã phát triển một thư viện gọi là TChannel, một giao thức RPC có độ tin cậy cao dựa trên Thrift, trước khi chuyển sang gRPC. Việc chuyển đổi từ REST sang gRPC đã giúp Uber cải thiện đáng kể hiệu suất và giảm kích thước của các tin nhắn trao đổi giữa các dịch vụ.

Những trường hợp nghiên cứu này minh họa cách các công ty lớn triển khai và điều chỉnh giao tiếp đồng bộ để đáp ứng nhu cầu cụ thể của họ. Chúng cũng cho thấy rằng không có một giải pháp "one-size-fits-all" cho giao tiếp microservices - mỗi công ty cần đánh giá nhu cầu của mình và lựa chọn công nghệ phù hợp nhất.

### 3.2.5. Tương lai của Synchronous Communication trong Microservices

Dù giao tiếp bất đồng bộ ngày càng phổ biến trong kiến trúc microservices, giao tiếp đồng bộ vẫn sẽ tiếp tục đóng vai trò quan trọng, đặc biệt là đối với các tương tác yêu cầu phản hồi tức thì.

Một xu hướng đáng chú ý trong tương lai của giao tiếp đồng bộ là Service Mesh. Công nghệ service mesh như Istio, Linkerd, và Consul Connect đang ngày càng được áp dụng để quản lý giao tiếp giữa các microservices, cung cấp các tính năng như mã hóa TLS, authentication, authorization, và observability. Service mesh hoạt động bằng cách triển khai một proxy bên cạnh mỗi instance của microservice, tạo thành một lớp infrastructure layer quản lý tất cả giao tiếp giữa các dịch vụ. Điều này cho phép tách biệt logic nghiệp vụ khỏi logic giao tiếp mạng, làm cho hệ thống dễ quản lý và bảo trì hơn.

WebAssembly (Wasm) cũng đang trở thành một công nghệ quan trọng trong tương lai của giao tiếp đồng bộ. WebAssembly đang được áp dụng trong API Gateway và service mesh để cải thiện hiệu suất và linh hoạt. Ví dụ, Envoy proxy (được sử dụng trong Istio) hỗ trợ các filter được viết bằng WebAssembly, cho phép mở rộng proxy một cách an toàn và hiệu quả mà không cần triển khai lại.

Serverless API Gateway là một xu hướng khác, với các giải pháp như AWS Lambda + API Gateway hoặc Azure Functions + API Management đang trở nên phổ biến, giúp giảm chi phí và cải thiện khả năng mở rộng. Trong mô hình này, API Gateway định tuyến yêu cầu đến các hàm serverless, chỉ được kích hoạt khi cần thiết, giúp giảm chi phí khi không có lưu lượng truy cập.

GraphQL Federation cũng đang ngày càng được áp dụng, với Apollo Federation và những công nghệ tương tự cho phép chia GraphQL schema thành nhiều microservices, mỗi dịch vụ sở hữu một phần của schema tổng thể. Điều này cho phép các đội phát triển độc lập quản lý các phần khác nhau của API mà không ảnh hưởng đến nhau.

Cuối cùng, các giao thức mới như HTTP/3 và QUIC hứa hẹn cải thiện hiệu suất giao tiếp đồng bộ, đặc biệt trong điều kiện mạng không ổn định. HTTP/3 dựa trên QUIC, một giao thức truyền tải mới phát triển bởi Google, cung cấp những cải tiến đáng kể về độ trễ và độ tin cậy so với TCP.

### 3.2.6. Kết luận

Giao tiếp đồng bộ one-to-one, đặc biệt là mẫu Request/Response, vẫn là một phần không thể thiếu trong kiến trúc microservices hiện đại. Mặc dù có những hạn chế về coupling và khả năng mở rộng, giao tiếp đồng bộ vẫn cung cấp nhiều lợi ích quan trọng như đơn giản, trực quan, và phản hồi tức thì.

Việc lựa chọn công nghệ triển khai phù hợp (REST, gRPC, GraphQL) phụ thuộc vào nhu cầu cụ thể của dự án, bao gồm yêu cầu về hiệu suất, khả năng mở rộng, và độ phức tạp. REST vẫn là lựa chọn phổ biến nhất do tính đơn giản và khả năng tương thích rộng rãi, nhưng gRPC và GraphQL đang ngày càng được áp dụng cho các trường hợp sử dụng cụ thể yêu cầu hiệu suất cao hoặc truy vấn dữ liệu phức tạp.

Trong thực tế, một hệ thống microservices hiệu quả thường kết hợp cả giao tiếp đồng bộ và bất đồng bộ, sử dụng mẫu phù hợp cho từng trường hợp sử dụng cụ thể. Hiểu rõ ưu điểm và hạn chế của mỗi mẫu giao tiếp là chìa khóa để thiết kế một kiến trúc microservices thành công.

Với sự phát triển liên tục của công nghệ và sự xuất hiện của các mẫu thiết kế mới, giao tiếp đồng bộ trong kiến trúc microservices sẽ tiếp tục phát triển và cải thiện, cung cấp nhiều tùy chọn hơn cho các kiến trúc sư và nhà phát triển để xây dựng các hệ thống phân tán mạnh mẽ và hiệu quả.

### **3.3. Asynchronous Communication Patterns (one-to-one)**

Giao tiếp bất đồng bộ one-to-one là một phương pháp quan trọng trong kiến trúc microservices, tạo nên sự tách rời (decoupling) giữa các dịch vụ và cải thiện khả năng mở rộng của hệ thống. Không giống như giao tiếp đồng bộ, trong giao tiếp bất đồng bộ one-to-one, microservice gửi tin nhắn không cần đợi phản hồi tức thì từ microservice nhận. Phần này sẽ tìm hiểu chi tiết về các mẫu giao tiếp bất đồng bộ one-to-one phổ biến, cùng với các công nghệ triển khai và các trường hợp sử dụng thực tế.

#### **3.3.1. Cơ chế hoạt động**

Giao tiếp bất đồng bộ one-to-one hoạt động theo mô hình khác biệt so với giao tiếp đồng bộ, với đặc điểm chính là sự phi chặn (non-blocking) trong quá trình truyền tải tin nhắn. Thay vì chờ đợi phản hồi, sender tiếp tục thực hiện các hoạt động khác sau khi gửi tin nhắn, và receiver xử lý tin nhắn khi có khả năng. Cơ chế này thường được triển khai thông qua một thành phần trung gian, có thể là message broker hoặc queue, đóng vai trò lưu trữ tạm thời tin nhắn khi receiver chưa sẵn sàng nhận [9].

Quá trình giao tiếp bất đồng bộ one-to-one thường tuân theo các bước sau: sender chuẩn bị tin nhắn với đầy đủ thông tin cần thiết; sender gửi tin nhắn đến hệ thống trung gian (message broker hoặc queue) mà không đợi phản hồi; sender tiếp tục xử lý các tác vụ khác; tin nhắn được lưu trữ trong hệ thống trung gian cho đến khi receiver sẵn sàng; receiver lấy tin nhắn từ hệ thống trung gian khi có thể; receiver xử lý tin nhắn theo logic nghiệp vụ của nó; tùy thuộc vào mẫu cụ thể, receiver có thể hoặc không gửi phản hồi.

Một khía cạnh quan trọng của giao tiếp bất đồng bộ là cơ chế delivery guarantee. Có ba mức độ đảm bảo phổ biến: at-most-once (tin nhắn có thể bị mất nhưng không bao giờ được gửi hơn một lần), at-least-once (tin nhắn được đảm bảo gửi ít nhất một lần, nhưng có thể trùng lặp), và exactly-once (tin nhắn được đảm bảo gửi chính xác một lần). Mức độ đảm bảo nào được sử dụng phụ thuộc vào yêu cầu nghiệp vụ và khả năng của hệ thống messaging [5].

Một cơ chế khác cần xem xét là message ordering. Trong một số trường hợp, thứ tự xử lý tin nhắn là quan trọng (ví dụ, tin nhắn cập nhật giá trị phải được xử lý sau tin nhắn tạo giá trị). Một số message broker đảm bảo thứ tự tin nhắn trong một partition hoặc queue cụ thể, trong khi những broker khác có thể không cung cấp đảm bảo này.

### 3.3.2. One-way Notifications Pattern

Mẫu One-way Notifications là hình thức đơn giản nhất của giao tiếp bất đồng bộ one-to-one. Trong mẫu này, một microservice gửi tin nhắn tới một microservice khác mà không mong đợi phản hồi. Đây là mẫu hoàn toàn "fire-and-forget", trong đó sender không quan tâm đến việc tin nhắn được xử lý như thế nào hoặc khi nào [9].

Mẫu One-way Notifications đặc biệt hữu ích cho các thông báo sự kiện không yêu cầu phản hồi, chẳng hạn như cập nhật trạng thái, ghi nhật ký, hoặc theo dõi hoạt động. Chẳng hạn, khi một đơn hàng được cập nhật, dịch vụ đơn hàng có thể gửi thông báo cho dịch vụ thông báo mà không cần đợi phản hồi.

Triển khai mẫu One-way Notifications thường đơn giản hơn so với các mẫu bất đồng bộ khác, vì không cần cơ chế tương quan giữa yêu cầu và phản hồi. Sender chỉ cần đặt tin nhắn vào hàng đợi hoặc chủ đề, và receiver xử lý nó khi sẵn sàng.

Các công nghệ phổ biến cho One-way Notifications bao gồm RabbitMQ, Apache Kafka, Amazon SQS, và Google Cloud Pub/Sub. Ví dụ, trong RabbitMQ, sender có thể xuất bản tin nhắn tới một exchange, và RabbitMQ định tuyến tin nhắn tới hàng đợi phù hợp dựa trên routing key. Receiver đăng ký hàng đợi và xử lý tin nhắn khi chúng đến. Apache Kafka hoạt động tương tự, với sender xuất bản tin nhắn tới chủ đề, và receiver đăng ký chủ đề để nhận tin nhắn.

Theo Newman [5], một khía cạnh quan trọng của One-way Notifications là xử lý lỗi. Vì sender không đợi phản hồi, việc xử lý lỗi phải được xử lý khác với giao tiếp đồng bộ. Các tiếp cận phổ biến bao gồm sử dụng Dead Letter Queues (DLQ) để lưu trữ tin nhắn không thể xử lý, các cơ chế retry để thử lại tin nhắn lỗi, và các cơ chế logging và monitoring để phát hiện và giải quyết vấn đề.

Richardson [4] đề xuất một số best practices khi triển khai One-way Notifications, bao gồm sử dụng message versioning để hỗ trợ backward và forward compatibility, triển khai idempotent message handling để tránh xử lý lặp lại tin nhắn, và sử dụng message acknowledgements để đảm bảo tin nhắn được xử lý thành công.

### 3.3.3. Message Queue Pattern

Message Queue là một mẫu cơ bản trong giao tiếp bất đồng bộ, trong đó các tin nhắn được gửi tới một hàng đợi trung gian, nơi chúng được lưu trữ cho đến khi được xử lý bởi consumer. Mẫu này hỗ trợ cả One-way Notifications và các mẫu bất đồng bộ khác, và cung cấp nhiều lợi ích như decoupling, buffering, và reliable delivery [9].

Trong kiến trúc Message Queue, thường có ba thành phần chính: Producer đặt tin nhắn vào hàng đợi; Queue lưu trữ tin nhắn cho đến khi chúng được xử lý; và Consumer lấy tin nhắn từ hàng đợi và xử lý chúng. Một tính năng quan trọng của Message Queue là nó cho phép asynchronous consumption, có nghĩa là tin nhắn không cần được xử lý ngay lập tức khi chúng đến. Điều này giúp xử lý các peak load và bảo vệ các service khi chúng đang quá tải.

Các công nghệ Message Queue phổ biến bao gồm RabbitMQ, Apache ActiveMQ, Amazon SQS, và Microsoft Azure Service Bus. Mỗi công nghệ có những đặc điểm

và ưu điểm riêng. Ví dụ, RabbitMQ hỗ trợ nhiều mẫu messaging như work queues, publish/subscribe, routing, và topics. Nó cũng cung cấp các tính năng như message acknowledgement, durability, và fair dispatch. Amazon SQS là một dịch vụ hàng đợi tin nhắn được quản lý, cung cấp các tính năng như at-least-once delivery, message retention, và visibility timeout.

Một khía cạnh quan trọng của Message Queue là khả năng mở rộng. Có hai cách chính để mở rộng hệ thống Message Queue: horizontal scaling (thêm nhiều consumer để xử lý nhiều tin nhắn hơn song song) và partitioning (chia hàng đợi thành nhiều partition, mỗi partition được xử lý bởi một consumer).

Hohpe và Woolf [9] mô tả nhiều mẫu messaging chi tiết hơn liên quan đến Message Queue, bao gồm Competing Consumers (nhiều consumer cạnh tranh để xử lý tin nhắn từ một hàng đợi), Message Dispatcher (định tuyến tin nhắn tới consumer cụ thể dựa trên một số tiêu chí), và Priority Queue (tin nhắn ưu tiên cao được xử lý trước).

Trong kiến trúc microservices, Message Queue thường được sử dụng để xử lý các tác vụ nặng hoặc tốn thời gian bất đồng bộ. Ví dụ, khi một người dùng đăng ký, service đăng ký có thể đặt một tin nhắn vào hàng đợi để gửi email xác nhận, trong khi vẫn phản hồi ngay lập tức cho người dùng.

### **3.3.4. Ưu điểm và Hạn chế của Asynchronous Communication (one-to-one)**

Giao tiếp bất đồng bộ one-to-one có nhiều ưu điểm và hạn chế đáng chú ý so với giao tiếp đồng bộ, hiểu rõ những điểm mạnh và điểm yếu này là chìa khóa để lựa chọn mẫu giao tiếp phù hợp.

Về ưu điểm, decoupling là lợi ích lớn nhất của giao tiếp bất đồng bộ. Các service giao tiếp không cần biết về nhau hoặc thậm chí hoạt động đồng thời. Sender có thể gửi tin nhắn ngay cả khi receiver đang ngoại tuyến, và receiver có thể xử lý tin nhắn khi sẵn sàng. Điều này làm giảm đáng kể coupling giữa các service. Ưu điểm thứ hai là khả năng phục hồi cải thiện. Giao tiếp bất đồng bộ có thể chịu được lỗi service tạm thời, vì tin nhắn có thể lưu trữ trong hàng đợi cho đến khi receiver khả dụng. Điều này giúp hệ thống duy trì hoạt động ngay cả khi một số thành phần gặp sự cố. Thứ ba, giao tiếp bất đồng bộ cung cấp khả năng mở rộng tốt hơn, cho phép mở rộng linh hoạt hơn bằng cách thêm nhiều receiver để xử lý nhiều tin nhắn hơn song song, mà không cần thay đổi sender. Ngoài ra, bất đồng bộ giúp cải thiện khả năng phản hồi và hiệu suất, vì sender không cần đợi receiver xử lý yêu cầu, do đó giảm thời gian chờ và cải thiện trải nghiệm người dùng. Giao tiếp bất đồng bộ cũng cung cấp buffering và smoothing, hàng đợi tin nhắn có thể hấp thụ đỉnh tải và bảo vệ service khỏi quá tải. Cuối cùng, giao tiếp bất đồng bộ cải thiện sử dụng tài nguyên, vì service không chặn tài nguyên chờ phản hồi, cho phép sử dụng tài nguyên hiệu quả hơn.

Tuy nhiên, giao tiếp bất đồng bộ cũng có những hạn chế đáng kể. Đầu tiên là độ phức tạp gia tăng. Giao tiếp bất đồng bộ thường đòi hỏi cơ sở hạ tầng bổ sung (như message broker) và logic phức tạp hơn để xử lý tin nhắn, quản lý lỗi, và đảm bảo tin cậy, làm tăng độ phức tạp của hệ thống. Hạn chế thứ hai là tính nhất quán yếu hơn. Bất đồng bộ thường

dẫn đến eventual consistency thay vì strong consistency, nghĩa là có độ trễ trước khi tất cả các service phản ánh cùng một trạng thái. Điều này có thể là một thách thức đối với các hệ thống yêu cầu tính nhất quán cao. Thứ ba, debugging và tracing có thể phức tạp hơn trong giao tiếp bất đồng bộ, vì luồng tin nhắn không trực quan như yêu cầu-phản hồi đồng bộ. Theo dõi tin nhắn qua nhiều service và hàng đợi có thể là một thách thức. Giao tiếp bất đồng bộ cũng gặp phải vấn đề về độ tin cậy và mất tin nhắn. Mặc dù nhiều message broker cung cấp các cơ chế đảm bảo độ tin cậy như message acknowledgement và persistent messaging, nhưng vẫn có nguy cơ mất tin nhắn hoặc trùng lặp tin nhắn. Cuối cùng, độ trễ end-to-end cao hơn. Mặc dù giao tiếp bất đồng bộ cải thiện độ phản hồi của sender, độ trễ end-to-end (từ khi gửi tin nhắn đến khi hoàn thành xử lý) thường cao hơn so với giao tiếp đồng bộ.

Để giảm thiểu những hạn chế này, nhiều kỹ thuật và mẫu thiết kế đã được phát triển, Ví dụ, để giải quyết vấn đề tính nhất quán yếu, mẫu thiết kế như Saga và Event Sourcing có thể được sử dụng để quản lý giao dịch phân tán và duy trì tính nhất quán trong hệ thống bất đồng bộ. Để cải thiện khả năng debugging và tracing, các công cụ như Zipkin và Jaeger có thể được sử dụng để theo dõi tin nhắn qua hệ thống phân tán. Và để cải thiện độ tin cậy, các message broker cung cấp các tính năng như message acknowledgement, dead letter queues, và persistent messaging.

### **3.3.5. Use cases phù hợp cho Asynchronous Communication (one-to-one)**

Giao tiếp bất đồng bộ one-to-one đặc biệt phù hợp cho một số trường hợp sử dụng cụ thể trong kiến trúc microservices. Hiểu rõ những trường hợp này giúp kiến trúc sư và nhà phát triển đưa ra quyết định đúng đắn về việc khi nào nên sử dụng giao tiếp bất đồng bộ.

Trường hợp sử dụng đầu tiên và quan trọng nhất là xử lý background. Các tác vụ tốn thời gian hoặc tài nguyên không cần phản hồi ngay lập tức là ứng viên lý tưởng cho giao tiếp bất đồng bộ. Ví dụ, khi người dùng tải lên video, việc xử lý video (như chuyển đổi định dạng, tạo thumbnail) có thể được thực hiện bất đồng bộ trong background, trong khi người dùng nhận được phản hồi ngay lập tức rằng quá trình tải lên đã hoàn tất.

Trường hợp thứ hai là xử lý số lượng lớn. Khi một service cần xử lý số lượng lớn yêu cầu, giao tiếp bất đồng bộ cho phép điều tiết tải thông qua buffering và sử dụng nhiều receiver để xử lý song song. Ví dụ, hệ thống xử lý log hoặc các ứng dụng phân tích dữ liệu lớn có thể sử dụng giao tiếp bất đồng bộ để thu thập và xử lý dữ liệu từ nhiều nguồn.

Trường hợp thứ ba là cải thiện phản hồi người dùng. Trong các tình huống mà phản hồi ngay lập tức cho người dùng là quan trọng, nhưng xử lý hoàn chỉnh có thể hoàn thành sau, giao tiếp bất đồng bộ cho phép hệ thống phản hồi ngay lập tức trong khi tiếp tục xử lý trong background. Ví dụ, khi đặt đơn hàng trực tuyến, hệ thống có thể xác nhận đơn hàng ngay lập tức và xử lý thanh toán, gửi email xác nhận, và cập nhật kho bất đồng bộ.

Trường hợp thứ tư là long-running processes. Các quy trình nghiệp vụ kéo dài, liên quan đến nhiều bước hoặc tương tác với nhiều service hay hệ thống bên ngoài, phù hợp với giao tiếp bất đồng bộ. Ví dụ, quy trình onboarding khách hàng có thể yêu cầu nhiều



kiểm tra (như xác minh danh tính, kiểm tra tín dụng) có thể mất nhiều thời gian, trong khi khách hàng không nên phải đợi quá trình này hoàn tất.

Trường hợp cuối cùng là integration với hệ thống bên ngoài. Khi giao tiếp với hệ thống bên ngoài không trong quyền kiểm soát của bạn, giao tiếp bất đồng bộ cung cấp sự cách ly và khả năng phục hồi. Nếu hệ thống bên ngoài chậm hoặc không khả dụng, hệ thống của bạn có thể tiếp tục hoạt động và xử lý yêu cầu khi hệ thống bên ngoài trở lại.

Bên cạnh đó, các hệ thống cần khả năng chịu lỗi cao và lỏng lẻo về tính nhất quán dữ liệu thường phù hợp với giao tiếp bất đồng bộ [16]. Tiếp cận bất đồng bộ cho phép hệ thống tiếp tục hoạt động ngay cả khi một số thành phần gặp sự cố, mặc dù có thể dẫn đến tính nhất quán yếu hơn (eventual consistency).

### 3.3.6. Case studies

Để hiểu rõ hơn về cách giao tiếp bất đồng bộ one-to-one được triển khai và sử dụng trong thế giới thực, chúng ta sẽ xem xét một số trường hợp nghiên cứu từ các công ty nổi tiếng.

LinkedIn là một ví dụ nổi bật về việc sử dụng giao tiếp bất đồng bộ one-to-one trong kiến trúc microservices quy mô lớn. Họ đã phát triển và sử dụng Apache Kafka, một nền tảng streaming phân tán, để xử lý hàng nghìn tỉ tin nhắn mỗi ngày [13]. LinkedIn sử dụng Kafka cho nhiều mục đích, bao gồm activity tracking, xử lý sự kiện real-time, và đồng bộ hóa dữ liệu giữa các hệ thống khác nhau. Ví dụ, khi một người dùng thực hiện hành động như cập nhật profile hoặc kết nối với người dùng khác, hành động này được ghi lại như một sự kiện và được gửi tới Kafka. Các service khác nhau, như service đề xuất kết nối hoặc service thông báo, sau đó xử lý sự kiện này để cập nhật dữ liệu và cung cấp chức năng của chúng.

PayPal cũng là một ví dụ đáng chú ý về việc sử dụng giao tiếp bất đồng bộ one-to-one trong một hệ thống xử lý thanh toán cần độ tin cậy cao. PayPal đã chuyển đổi từ một kiến trúc monolithic sang microservices, sử dụng Apache Kafka làm platform tin nhắn chính [14]. Trong kiến trúc này, các service như xử lý thanh toán, phát hiện gian lận, và thông báo người dùng giao tiếp bất đồng bộ thông qua Kafka. Ví dụ, khi một giao dịch được khởi tạo, một sự kiện được gửi tới Kafka và được xử lý bởi nhiều service khác nhau, bao gồm service xác thực, service phát hiện gian lận, và service xử lý giao dịch. Mỗi service thực hiện chức năng cụ thể của nó và có thể tạo ra các sự kiện khác, tạo thành một luồng xử lý transaction.

Uber cũng sử dụng giao tiếp bất đồng bộ one-to-one rộng rãi trong kiến trúc microservices của họ. Uber đã xây dựng một platform gọi là Cadence, một service orchestration engine cho phép viết workflow code như code đồng bộ, nhưng thực thi chúng bất đồng bộ và phân tán [15]. Cadence được sử dụng cho nhiều use-case trong Uber, bao gồm matching riders với drivers, thanh toán, và onboarding drivers. Ví dụ, khi một yêu cầu chuyến đi được tạo, Cadence khởi tạo một workflow để tìm driver phù hợp, đặt lịch chuyến đi, xử lý thanh toán, và cập nhật trạng thái của rider và driver. Mỗi bước

trong workflow này được thực hiện bất đồng bộ, cho phép quá trình tiếp tục ngay cả khi một số service tạm thời không khả dụng.

Những trường hợp nghiên cứu này minh họa cách các công ty lớn triển khai giao tiếp bất đồng bộ one-to-one để xây dựng hệ thống phân tán có khả năng mở rộng, linh hoạt, và đáng tin cậy. Chúng cũng cho thấy rằng giao tiếp bất đồng bộ đặc biệt hữu ích cho các hệ thống xử lý số lượng lớn giao dịch hoặc sự kiện, và cho các quy trình nghiệp vụ phức tạp liên quan đến nhiều service.

### **3.3.7. Kết luận**

Giao tiếp bất đồng bộ one-to-one đóng vai trò quan trọng trong các kiến trúc microservices hiện đại, cung cấp decoupling, khả năng mở rộng, và khả năng phục hồi cao hơn so với giao tiếp đồng bộ. Các mẫu như One-way Notifications và Message Queue cung cấp các cách khác nhau để triển khai giao tiếp bất đồng bộ, mỗi mẫu với những ưu điểm và use case phù hợp riêng.

Mặc dù giao tiếp bất đồng bộ có nhiều lợi ích, nó cũng mang đến các thách thức về độ phức tạp, debugging, và tính nhất quán dữ liệu. Tuy nhiên, với các công nghệ và mẫu thiết kế phù hợp, những thách thức này có thể được giải quyết, cho phép xây dựng các hệ thống microservices mạnh mẽ và có khả năng mở rộng.

Trong thực tế, một hệ thống microservices hiệu quả thường kết hợp cả giao tiếp đồng bộ và bất đồng bộ, sử dụng mỗi loại cho những trường hợp sử dụng phù hợp. Hiểu rõ các mẫu giao tiếp khác nhau và khi nào nên sử dụng chúng là chìa khóa để thiết kế một kiến trúc microservices thành công.

Với sự phát triển của các công nghệ như serverless computing, stream processing, và reactive programming, tương lai của giao tiếp bất đồng bộ one-to-one trong microservices còn nhiều hứa hẹn. Những tiến bộ này sẽ tiếp tục mở rộng khả năng và tính hiệu quả của giao tiếp bất đồng bộ, cho phép xây dựng các hệ thống phân tán ngày càng phức tạp và có khả năng mở rộng.

## **3.4. Asynchronous Communication Patterns (one-to-many)**

Giao tiếp bất đồng bộ one-to-many là mô hình quan trọng trong kiến trúc microservice, cho phép một dịch vụ gửi thông điệp đến nhiều dịch vụ nhận khác nhau cùng một lúc. Không giống như mô hình one-to-one, trong giao tiếp one-to-many, một thông điệp được phân phối đến nhiều người nhận mà không cần biết trước danh tính hoặc số lượng người nhận. Mô hình này đặc biệt hữu ích trong các hệ thống phân tán quy mô lớn, nơi cần thông báo nhiều thành phần về các sự kiện quan trọng [5]. Phần này sẽ tìm hiểu chi tiết về các mẫu giao tiếp bất đồng bộ one-to-many phổ biến, cùng với các công nghệ triển khai và các trường hợp sử dụng thực tế.

### 3.4.1. Cơ chế hoạt động

Giao tiếp bất đồng bộ one-to-many hoạt động theo nguyên tắc phân phối thông điệp từ một nguồn đến nhiều đích mà không yêu cầu sự tương tác trực tiếp giữa chúng. Thay vì thiết lập kết nối trực tiếp với từng người nhận, người gửi phát hành thông điệp đến một kênh trung gian, và thông điệp này được phân phối đến tất cả các dịch vụ đã đăng ký với kênh đó [9].

Quá trình giao tiếp bất đồng bộ one-to-many thường tuân theo các bước sau: nhà xuất bản (publisher) chuẩn bị thông điệp; nhà xuất bản gửi thông điệp đến kênh hoặc topic trung gian; hệ thống trung gian nhận thông điệp và lưu trữ tạm thời; hệ thống trung gian xác định tất cả người đăng ký (subscribers) cho kênh hoặc topic đó; hệ thống trung gian phân phối thông điệp đến tất cả người đăng ký phù hợp; các người đăng ký nhận và xử lý thông điệp độc lập với nhau.

Một yếu tố quan trọng trong giao tiếp one-to-many là khả năng mở rộng động số lượng người nhận. Các dịch vụ mới có thể đăng ký nhận thông điệp mà không cần thay đổi logic của nhà xuất bản, tạo nên sự linh hoạt cao cho hệ thống. Đồng thời, nhiều hệ thống trung gian còn hỗ trợ các cơ chế lọc thông điệp, cho phép người đăng ký chỉ nhận những thông điệp phù hợp với tiêu chí cụ thể [4].

Cũng như trong giao tiếp one-to-one, các cơ chế đảm bảo độ tin cậy của thông điệp như at-most-once, at-least-once hoặc exactly-once đều có thể được áp dụng tùy theo yêu cầu của hệ thống. Tuy nhiên, do tính chất phân phối đến nhiều người nhận, việc đảm bảo thông điệp được nhận bởi tất cả người đăng ký có thể phức tạp hơn [6].

### 3.4.2. Publish/Subscribe Pattern

Publish/Subscribe (Pub/Sub) là mẫu giao tiếp không đồng bộ one-to-many phổ biến nhất trong kiến trúc microservice. Trong mô hình này, nhà xuất bản (publisher) không gửi thông điệp trực tiếp đến người nhận cụ thể, mà thay vào đó phát hành (publish) thông điệp đến một kênh (topic hoặc channel). Các dịch vụ quan tâm đến loại thông điệp này đăng ký (subscribe) vào kênh đó và nhận tất cả thông điệp được phát hành [9].

Wolff [7] phân biệt hai biến thể chính của mô hình Pub/Sub: Topic-based Pub/Sub, trong đó người đăng ký đăng ký vào các kênh cụ thể và nhận tất cả thông điệp được phát hành đến kênh đó; và Content-based Pub/Sub, trong đó người đăng ký định nghĩa các tiêu chí lọc và chỉ nhận thông điệp phù hợp với tiêu chí đã định.

Các công nghệ phổ biến để triển khai Pub/Sub bao gồm Apache Kafka, RabbitMQ, Google Cloud Pub/Sub, Amazon SNS và Redis Pub/Sub. Ví dụ, Apache Kafka tổ chức thông điệp thành các topic và lưu trữ chúng trong log phân tán, cho phép người đăng ký xử lý thông điệp theo tốc độ riêng của họ và thậm chí phát lại thông điệp cũ. RabbitMQ sử dụng mô hình exchange-queue, trong đó các exchange nhận thông điệp từ nhà xuất bản và định tuyến chúng đến các queue dựa trên các quy tắc khác nhau, và người đăng ký nhận thông điệp từ queue [3].

Pub/Sub đặc biệt hữu ích cho các trường hợp như cập nhật trạng thái và sự kiện hệ thống, phân tích và giám sát, đồng bộ hóa dữ liệu giữa các dịch vụ, và IoT. Chẳng hạn, khi

một đơn hàng được cập nhật, dịch vụ đơn hàng có thể phát hành sự kiện "order\_updated" đến một topic, và nhiều dịch vụ khác nhau như thanh toán, thông báo, và kho hàng có thể đăng ký topic này để cập nhật trạng thái tương ứng của chúng [5].

Uber sử dụng mô hình Pub/Sub để xây dựng nền tảng sự kiện quy mô lớn, xử lý hàng trăm tỷ sự kiện mỗi ngày. Kiến trúc của họ sử dụng Apache Kafka làm nền tảng chính, cho phép các dịch vụ phát hành và đăng ký các sự kiện mà không cần biết về nhau. Khi một chuyến đi được tạo, sự kiện "trip\_created" được phát hành và nhiều dịch vụ đăng ký sự kiện này để thực hiện các chức năng khác nhau như thông báo cho tài xế, tính toán giá thành, và thu thập dữ liệu phân tích [12].

### 3.4.3. Event Sourcing Pattern

Event Sourcing là một mẫu kiến trúc trong đó thay vì lưu trữ trạng thái hiện tại của đối tượng, hệ thống lưu trữ chuỗi các sự kiện mô tả những thay đổi đã xảy ra với đối tượng đó theo thời gian. Trạng thái hiện tại được tạo ra bằng cách phát lại các sự kiện này [16].

Khi kết hợp với kiến trúc microservice, Event Sourcing thường được triển khai như một mô hình giao tiếp không đồng bộ one-to-many, trong đó các sự kiện được lưu trữ trong Event Store và nhiều dịch vụ có thể đăng ký để nhận thông báo về các sự kiện mới. Richardson [4] chỉ ra rằng Event Sourcing thường được sử dụng kết hợp với Command Query Responsibility Segregation (CQRS) để tách biệt hoạt động đọc và ghi dữ liệu.

Event Sourcing có nhiều ưu điểm trong kiến trúc microservice, bao gồm khả năng lưu trữ lịch sử hoàn chỉnh của tất cả thay đổi (hữu ích cho kiểm toán và phân tích), khả năng phục hồi trạng thái hệ thống tại bất kỳ thời điểm nào, và khả năng phát triển song song các dịch vụ dựa trên luồng sự kiện chung.

Tuy nhiên, mẫu này cũng có những thách thức như độ phức tạp cao trong triển khai, vấn đề về schema evolution khi cấu trúc sự kiện thay đổi, và chi phí lưu trữ lớn do phải lưu trữ tất cả sự kiện. Event Sourcing phù hợp cho các hệ thống tài chính, quản lý đơn hàng, hệ thống quản lý nội dung, và các ứng dụng yêu cầu khả năng kiểm toán cao [4].

PayPal đã áp dụng mô hình Event Sourcing kết hợp với kiến trúc microservice để xây dựng nền tảng thanh toán mới. Mỗi giao dịch thanh toán được mô hình hóa như một chuỗi các sự kiện (payment\_initiated, payment\_authorized, payment\_captured, v.v.), và nhiều dịch vụ khác nhau đăng ký để nhận thông báo về các sự kiện mới để thực hiện các chức năng như phát hiện gian lận, thông báo cho người dùng, và cập nhật báo cáo tài chính [14].

### 3.4.4. Ưu điểm và Hạn chế của Asynchronous Communication (one-to-many)

Giao tiếp bất đồng bộ one-to-many mang lại nhiều ưu điểm đáng kể so với các mô hình giao tiếp khác trong kiến trúc microservice, nhưng cũng đi kèm với một số hạn chế cần được xem xét kỹ lưỡng.

Về ưu điểm, đầu tiên phải kể đến sự tách rời cao (high decoupling) giữa nhà xuất bản và người đăng ký. Nhà xuất bản không cần biết danh tính hoặc số lượng người đăng ký, cho phép thêm hoặc xóa người đăng ký mà không ảnh hưởng đến nhà xuất bản. Điều này tạo ra hệ thống linh hoạt, có thể mở rộng dễ dàng [5]. Thứ hai, mô hình này hỗ trợ khả năng mở rộng động (dynamic scalability). Có thể thêm người đăng ký mới bất kỳ lúc nào để xử lý tải tăng lên hoặc cung cấp chức năng mới mà không cần thay đổi nhà xuất bản. Ưu điểm thứ ba là khả năng xử lý đồng thời (parallel processing). Nhiều người đăng ký có thể xử lý cùng một thông điệp song song, tăng hiệu suất tổng thể của hệ thống [7]. Giao tiếp one-to-many cũng mang lại khả năng chịu lỗi cao (fault tolerance). Lỗi trong một người đăng ký không ảnh hưởng đến nhà xuất bản hoặc các người đăng ký khác, tăng tính ổn định của hệ thống [4]. Cuối cùng, mô hình này rất phù hợp cho xử lý sự kiện phân tán (distributed event processing). Cho phép xây dựng hệ thống phản ứng với các sự kiện một cách linh hoạt và mở rộng [9].

Tuy nhiên, giao tiếp bất đồng bộ one-to-many cũng có những hạn chế đáng kể. Đầu tiên là độ phức tạp cao hơn. Triển khai mô hình này thường đòi hỏi cơ sở hạ tầng bổ sung (như message broker hoặc hệ thống streaming) và logic phức tạp hơn để quản lý việc phát hành và đăng ký [5]. Thứ hai, debugging và tracing khó khăn hơn. Theo dõi luồng thông điệp qua nhiều dịch vụ và trong môi trường phân tán có thể rất phức tạp, đặc biệt khi xử lý các lỗi [7]. Giao tiếp one-to-many cũng dẫn đến tính nhất quán yếu hơn (eventual consistency). Các dịch vụ khác nhau có thể xử lý thông điệp với tốc độ khác nhau, dẫn đến trạng thái hệ thống không đồng nhất trong một khoảng thời gian [4]. Ngoài ra, mô hình này có thể gặp vấn đề về quản lý thứ tự thông điệp. Đảm bảo thông điệp được xử lý theo đúng thứ tự có thể khó khăn, đặc biệt trong môi trường phân tán [6]. Cuối cùng, giao tiếp one-to-many có thể dẫn đến độ trễ end-to-end cao hơn. Mặc dù nhà xuất bản có thể phản hồi nhanh, thời gian để tất cả người đăng ký hoàn thành xử lý có thể đáng kể [3].

Để giảm thiểu những hạn chế này, nhiều kỹ thuật và mẫu thiết kế đã được phát triển. Chẳng hạn, để cải thiện khả năng debugging và tracing, các công cụ như Zipkin và Jaeger có thể được sử dụng để theo dõi thông điệp qua hệ thống phân tán. Để quản lý thứ tự thông điệp, các nền tảng như Kafka hỗ trợ partitioning và ordering guarantees. Và để giải quyết vấn đề tính nhất quán yếu, các kỹ thuật như snapshot, event versioning, và materialized views có thể được áp dụng.

### **3.4.5. Use cases phù hợp cho Asynchronous Communication (one-to-many)**

Giao tiếp bất đồng bộ one-to-many đặc biệt phù hợp cho một số trường hợp sử dụng cụ thể trong kiến trúc microservice. Hiểu rõ những trường hợp này giúp kiến trúc sư và nhà phát triển đưa ra quyết định đúng đắn về việc khi nào nên sử dụng mô hình giao tiếp này.

Trường hợp sử dụng đầu tiên và phổ biến nhất là thông báo sự kiện hệ thống (system event notification). Khi một sự kiện quan trọng xảy ra (như tạo đơn hàng, cập nhật hồ sơ người dùng, hoặc thay đổi trạng thái), nhiều dịch vụ khác nhau có thể cần biết về sự

kiện đó. Ví dụ, khi một đơn hàng được tạo, dịch vụ thanh toán, dịch vụ thông báo, dịch vụ kho hàng, và dịch vụ phân tích đều cần được thông báo [5].

Trường hợp thứ hai là tích hợp dữ liệu cross-service. Khi dữ liệu được tạo hoặc cập nhật trong một dịch vụ, các dịch vụ khác có thể cần sao chép hoặc chuyển đổi dữ liệu đó cho mục đích riêng của chúng. Ví dụ, khi dịch vụ người dùng cập nhật thông tin người dùng, dịch vụ tìm kiếm và dịch vụ đề xuất cần cập nhật chỉ mục của chúng [4].

Trường hợp thứ ba là xử lý dữ liệu thời gian thực (real-time data processing). Đối với các hệ thống cần phân tích hoặc phản ứng với dữ liệu thời gian thực, mô hình one-to-many cho phép nhiều dịch vụ xử lý song song cùng một luồng dữ liệu. Ví dụ, dữ liệu click stream của người dùng có thể được phân tích bởi nhiều dịch vụ để phát hiện hành vi bất thường, cập nhật mô hình ML, và cung cấp phân tích theo thời gian thực [13].

Trường hợp thứ tư là logging và monitoring phân tán. Các hệ thống phân tán cần thu thập log và metric từ nhiều dịch vụ cho mục đích giám sát và phân tích. Mô hình one-to-many cho phép các dịch vụ phát hành log và metric đến một kênh trung tâm, nơi chúng có thể được tiêu thụ bởi nhiều dịch vụ như lưu trữ log, cảnh báo, và phân tích [6].

Trường hợp thứ năm là cung cấp feed dữ liệu cho các hệ thống bên ngoài. Khi nhiều hệ thống bên ngoài (như ứng dụng đối tác, dịch vụ phân tích, hoặc hệ thống báo cáo) cần truy cập vào cùng một luồng dữ liệu, mô hình one-to-many cho phép phân phối dữ liệu một cách hiệu quả mà không tạo thêm gánh nặng cho hệ thống nguồn [12].

Ngoài ra, các use case liên quan đến Internet of Things (IoT) cũng rất phù hợp với giao tiếp one-to-many. Đối với các hệ thống IoT, nơi dữ liệu từ nhiều thiết bị cần được thu thập và xử lý bởi nhiều dịch vụ, mô hình one-to-many cung cấp cách hiệu quả để phân phối dữ liệu đến tất cả các dịch vụ quan tâm [10].

Cuối cùng, các hệ thống cần khả năng chịu lỗi cao và tách rời mạnh mẽ giữa các thành phần cũng rất phù hợp với mô hình one-to-many. Tiếp cận này cho phép các dịch vụ hoạt động độc lập với nhau, làm tăng tính ổn định và độ tin cậy của hệ thống tổng thể [16].

### 3.4.6. Kết luận

Giao tiếp bất đồng bộ one-to-many đóng vai trò quan trọng trong các kiến trúc microservice hiện đại, cung cấp khả năng phân phối thông điệp từ một nguồn đến nhiều đích một cách hiệu quả. Các mẫu như Publish/Subscribe, Event Sourcing, Message Broker với Exchange Routing, và Streaming Platform cung cấp các cách khác nhau để triển khai giao tiếp one-to-many, mỗi mẫu với những ưu điểm và trường hợp sử dụng phù hợp riêng.

Mô hình này mang lại nhiều lợi ích như sự tách rời cao, khả năng mở rộng động, xử lý đồng thời, và khả năng chịu lỗi tốt. Tuy nhiên, nó cũng đặt ra những thách thức về độ phức tạp, debugging, tính nhất quán dữ liệu, và quản lý thứ tự thông điệp. Với các công nghệ và mẫu thiết kế phù hợp, những thách thức này có thể được giải quyết, cho phép xây dựng các hệ thống microservice mạnh mẽ và có khả năng mở rộng.

Các trường hợp sử dụng điển hình cho giao tiếp one-to-many bao gồm thông báo sự kiện hệ thống, tích hợp dữ liệu cross-service, xử lý dữ liệu thời gian thực, logging và monitoring phân tán, và IoT. Nhiều công ty công nghệ hàng đầu như Uber, LinkedIn, PayPal, và Netflix đã triển khai thành công các mô hình giao tiếp one-to-many trong kiến trúc microservice của họ để xử lý khối lượng dữ liệu và sự kiện khổng lồ.

Trong thực tế, một hệ thống microservice hiệu quả thường kết hợp cả giao tiếp đồng bộ, bất đồng bộ one-to-one, và bất đồng bộ one-to-many, sử dụng mỗi loại cho những trường hợp sử dụng phù hợp. Hiểu rõ các mẫu giao tiếp khác nhau và khi nào nên sử dụng chúng là chìa khóa để thiết kế một kiến trúc microservice thành công và có khả năng mở rộng.

Với sự phát triển của các công nghệ như serverless computing, edge computing, và 5G, tương lai của giao tiếp bất đồng bộ one-to-many trong microservice còn nhiều hứa hẹn. Những tiến bộ này sẽ tiếp tục mở rộng khả năng và hiệu quả của giao tiếp one-to-many, cho phép xây dựng các hệ thống phân tán ngày càng phức tạp và có khả năng mở rộng.

Với sự phát triển của các công nghệ như serverless computing, edge computing, và 5G, tương lai của giao tiếp bất đồng bộ one-to-many trong microservice còn nhiều hứa hẹn. Những tiến bộ này sẽ tiếp tục mở rộng khả năng và hiệu quả của giao tiếp one-to-many, cho phép xây dựng các hệ thống phân tán ngày càng phức tạp và có khả năng mở rộng.

### 3.5. Tổng kết

Trong chương này, chúng ta đã tìm hiểu chi tiết về các mẫu giao tiếp trong kiến trúc microservices. Đầu tiên, chúng ta đã phân loại các mẫu giao tiếp theo hai tiêu chí chính: communication mode (synchronous/asynchronous) và communication scope (one-to-one/one-to-many).

Với giao tiếp đồng bộ (synchronous) one-to-one, chúng ta tập trung vào mẫu Request/Response, được triển khai thông qua các công nghệ phổ biến như REST, gRPC và GraphQL. Mẫu này có ưu điểm là đơn giản, trực quan và phản hồi tức thì, nhưng hạn chế ở khả năng mở rộng và độ tin cậy.

Trong giao tiếp bất đồng bộ (asynchronous) one-to-one, chúng ta đã xem xét các mẫu như One-way Notifications và Message Queue. Các mẫu này cung cấp decoupling tốt hơn, khả năng chịu lỗi cao và khả năng mở rộng tốt, nhưng lại phức tạp hơn trong triển khai và debug.

Đối với giao tiếp bất đồng bộ one-to-many, chúng ta đã tìm hiểu các mẫu như Publish/Subscribe, Event Sourcing và Message Broker với Exchange Routing. Các mẫu này cung cấp decoupling cao nhất và khả năng mở rộng tốt nhất, đặc biệt phù hợp cho việc phát tán thông tin và xử lý sự kiện.

Mỗi mẫu giao tiếp đều có những use cases phù hợp riêng và việc lựa chọn mẫu giao tiếp phù hợp phụ thuộc vào nhiều yếu tố như yêu cầu về độ trễ, tính nhất quán dữ liệu, khả năng mở rộng và độ phức tạp triển khai.

Trong thực tế, một hệ thống microservices hiệu quả thường kết hợp nhiều mẫu giao tiếp khác nhau, sử dụng mỗi mẫu cho những trường hợp phù hợp nhất. Hiểu rõ các mẫu giao tiếp và tiêu chí lựa chọn là nền tảng quan trọng để thiết kế một kiến trúc microservices thành công.

Ở chương tiếp theo, chúng ta sẽ tiến hành triển khai thử nghiệm các mẫu giao tiếp đã học trong một hệ thống microservices thực tế. Chúng ta sẽ xây dựng một ứng dụng mẫu với nhiều microservices khác nhau, triển khai các mẫu giao tiếp khác nhau, và đánh giá hiệu suất cũng như tính phù hợp của từng mẫu trong các tình huống cụ thể.



## Chương 4.

# Triển khai thử nghiệm

### 4.1. Mô tả bài toán và yêu cầu

#### 4.1.1. Giới thiệu bài toán

Trong dự án này, một hệ thống thương mại điện tử dựa trên kiến trúc microservice được triển khai nhằm đánh giá các mẫu giao tiếp khác nhau giữa các dịch vụ. Hệ thống được thiết kế để mô phỏng các tương tác thực tế giữa các microservice trong một ứng dụng thương mại điện tử, bao gồm quản lý đơn hàng, kiểm tra tồn kho, xử lý thanh toán và thông báo khách hàng. Mỗi quy trình nghiệp vụ này yêu cầu sự phối hợp giữa nhiều dịch vụ và đòi hỏi các mẫu giao tiếp khác nhau tùy thuộc vào tính chất của tác vụ.

Bài toán tập trung vào việc đánh giá hiệu suất và độ tin cậy của các mẫu giao tiếp trong bốn kịch bản nghiệp vụ chính. Kịch bản thứ nhất là kiểm tra và cập nhật tồn kho khi khách hàng đặt hàng, hệ thống cần kiểm tra tồn kho và cập nhật số lượng sản phẩm khả dụng. Kịch bản thứ hai liên quan đến xử lý thanh toán sau khi xác nhận đơn hàng, hệ thống cần xử lý thanh toán thông qua dịch vụ thanh toán. Kịch bản thứ ba là thông báo kết quả đơn hàng, sau khi hoàn tất đơn hàng, hệ thống cần gửi thông báo qua nhiều kênh khác nhau như email, tin nhắn và cập nhật phân tích. Cuối cùng là kịch bản ghi nhận hoạt động người dùng, hệ thống ghi lại hoạt động của người dùng để phục vụ cho phân tích dữ liệu và phát hiện gian lận.

Cho mỗi kịch bản, các mẫu giao tiếp khác nhau được triển khai và so sánh, bao gồm giao tiếp đồng bộ (Synchronous), giao tiếp bất đồng bộ dạng một-một (Asynchronous One-to-One) và giao tiếp bất đồng bộ dạng một-nhiều (Asynchronous One-to-Many).

#### 4.1.2. Yêu cầu hệ thống

Dự án này được phát triển với mục tiêu đánh giá các mẫu giao tiếp microservice dựa trên các yêu cầu cụ thể. Về yêu cầu chức năng, hệ thống phải hỗ trợ tạo và quản lý đơn hàng, bao gồm thêm sản phẩm vào đơn hàng và xử lý đơn hàng. Hệ thống cần thực hiện kiểm tra và cập nhật tồn kho khi có đơn hàng mới, đồng thời xử lý thanh toán cho đơn hàng và cập nhật trạng thái thanh toán. Ngoài ra, hệ thống phải gửi thông báo đến khách hàng thông qua nhiều kênh khác nhau như email và thông báo đẩy, đồng thời ghi lại hoạt động của người dùng cho mục đích phân tích và phát hiện gian lận.

Về yêu cầu phi chức năng, hiệu suất là yếu tố quan trọng với thời gian phản hồi thấp cho các giao dịch quan trọng, đặc biệt là kiểm tra tồn kho và xử lý đơn hàng. Độ tin cậy đòi hỏi hệ thống phải đảm bảo tính nhất quán dữ liệu giữa các dịch vụ, đặc biệt là đối với tồn kho và trạng thái đơn hàng. Khả năng chịu lỗi yêu cầu hệ thống phải tiếp tục hoạt động ngay cả khi một hoặc nhiều dịch vụ không khả dụng. Khả năng mở rộng đòi hỏi

các dịch vụ phải có thể mở rộng độc lập để đáp ứng nhu cầu tăng đột biến. Cuối cùng, tính linh hoạt yêu cầu kiến trúc phải cho phép thêm hoặc thay đổi dịch vụ mà không ảnh hưởng đến toàn bộ hệ thống.

#### **4.1.3. Kiến trúc tổng thể hệ thống**

Hệ thống được thiết kế theo kiến trúc microservice, trong đó mỗi dịch vụ chịu trách nhiệm cho một chức năng nghiệp vụ cụ thể. Kiến trúc tổng thể của hệ thống bao gồm tám dịch vụ chính, mỗi dịch vụ đóng vai trò riêng biệt trong quy trình xử lý đơn hàng và tương tác với người dùng.

Dịch vụ Order Service đóng vai trò trung tâm, xử lý việc tạo và quản lý đơn hàng, đồng thời điều phối luồng xử lý giữa các dịch vụ khác nhau. Dịch vụ này tương tác trực tiếp với Inventory Service để kiểm tra và cập nhật tồn kho, và với Payment Service để xử lý thanh toán cho đơn hàng. Inventory Service quản lý tồn kho sản phẩm, hỗ trợ kiểm tra và cập nhật số lượng tồn kho khi có đơn hàng mới. Payment Service xử lý thanh toán cho đơn hàng và cập nhật trạng thái thanh toán, đảm bảo giao dịch tài chính được thực hiện an toàn và đáng tin cậy.

Sau khi đơn hàng được xử lý, thông tin được chuyển đến ba dịch vụ khác nhau để thông báo cho khách hàng. Email Service gửi email thông báo đến khách hàng, trong khi Notification Service gửi thông báo đầy trực tiếp đến các thiết bị của khách hàng. Analytics Service thu thập và phân tích dữ liệu từ các hoạt động của người dùng và đơn hàng, cung cấp thông tin chi tiết về hiệu suất kinh doanh và hành vi người dùng.

Hai dịch vụ còn lại phục vụ cho việc ghi nhận và phân tích hoạt động người dùng. Fraud Service phát hiện các hoạt động đáng ngờ và ngăn chặn gian lận, bảo vệ hệ thống khỏi các hoạt động độc hại. Activity Service ghi lại hoạt động người dùng và định tuyến sự kiện đến các dịch vụ phù hợp, đóng vai trò quan trọng trong việc thu thập dữ liệu cho phân tích và phát hiện gian lận.

Kiến trúc này cho phép các dịch vụ hoạt động độc lập, đồng thời cung cấp khả năng mở rộng và linh hoạt cao. Mỗi dịch vụ có thể được phát triển, triển khai và mở rộng độc lập, giúp tăng cường khả năng chịu lỗi và hiệu suất của hệ thống.

#### **4.1.4. Các mẫu giao tiếp đánh giá**

Trong dự án này, ba mẫu giao tiếp chính trong kiến trúc microservice được đánh giá. Mẫu giao tiếp đầu tiên là giao tiếp đồng bộ (Synchronous Communication), sử dụng REST API, trong đó dịch vụ gọi gửi yêu cầu và chờ phản hồi từ dịch vụ được gọi. Mẫu này được triển khai thông qua giao thức HTTP và thường được sử dụng cho các tương tác yêu cầu phản hồi nhanh. Đây là mẫu đơn giản nhất và dễ triển khai, nhưng có thể gây ra vấn đề về hiệu suất và khả năng chịu lỗi khi hệ thống mở rộng.

Mẫu giao tiếp thứ hai là giao tiếp bất đồng bộ dạng một-một (Asynchronous One-to-One Communication), sử dụng RabbitMQ message queue. Trong mẫu này, dịch vụ gửi đặt thông điệp vào hàng đợi và tiếp tục xử lý mà không cần chờ phản hồi, trong khi dịch

vụ nhận sẽ xử lý thông điệp khi sẵn sàng. Mẫu này được sử dụng cho các tác vụ không yêu cầu phản hồi ngay lập tức, giúp cải thiện hiệu suất và khả năng chịu lỗi của hệ thống.

Mẫu giao tiếp thứ ba là giao tiếp bất đồng bộ dạng một-nhiều (Asynchronous One-to-Many Communication), sử dụng Kafka event streaming. Trong mẫu này, dịch vụ xuất bản sự kiện cho nhiều dịch vụ đăng ký, cho phép nhiều dịch vụ xử lý cùng một sự kiện độc lập với nhau. Mẫu này đặc biệt hữu ích khi một sự kiện cần được xử lý bởi nhiều dịch vụ độc lập, giúp tăng cường tính linh hoạt và khả năng mở rộng của hệ thống.

Việc đánh giá các mẫu giao tiếp này sẽ giúp xác định mẫu phù hợp nhất cho từng kịch bản nghiệp vụ dựa trên các tiêu chí như hiệu suất, độ tin cậy, khả năng chịu lỗi và khả năng mở rộng.

## 4.2. Cài đặt và triển khai

### 4.2.1. Cấu trúc dự án

Dự án được tổ chức theo kiến trúc monorepo, cho phép quản lý nhiều microservice trong một repository duy nhất. Cấu trúc này tạo điều kiện thuận lợi cho việc chia sẻ mã nguồn và tài nguyên giữa các dịch vụ, đồng thời đơn giản hóa quá trình triển khai và kiểm thử. Cấu trúc thư mục chính của dự án bao gồm thư mục apps chứa mã nguồn cho từng microservice riêng biệt, thư mục libs chứa các module dùng chung, và thư mục test-script chứa các kịch bản kiểm thử hiệu suất.

Trong thư mục apps, mỗi dịch vụ được tổ chức theo cấu trúc tiêu chuẩn của NestJS, bao gồm các module, controller, service và entity. Điều này giúp duy trì tính nhất quán và khả năng bảo trì trên toàn dự án. Thư mục libs chứa module common chia sẻ các định nghĩa, interface và utility dùng chung giữa các microservice, giúp tránh việc trùng lặp mã và đảm bảo tính nhất quán trong toàn hệ thống.

Mỗi microservice được đóng gói thành một container Docker riêng biệt, cho phép triển khai độc lập và cô lập. File Dockerfile cho mỗi service được thiết kế để tối ưu hóa kích thước image và thời gian xây dựng. Tập Docker Compose phối hợp việc triển khai tất cả các microservice cùng với các dịch vụ cơ sở hạ tầng như cơ sở dữ liệu, RabbitMQ và Kafka.

### 4.2.2. Triển khai các microservice chính

Order Service đóng vai trò trung tâm trong hệ thống, xử lý việc tạo và quản lý đơn hàng, đồng thời điều phối luồng xử lý giữa các dịch vụ khác. Service này được triển khai với các endpoint RESTful cho các hoạt động CRUD đơn hàng, cùng với các handlers cho các sự kiện từ các dịch vụ khác.

Để xử lý đơn hàng, khóa luận đã triển khai một thực thể Order với các trường như id, productId, customerId, quantity, status, paymentId, paymentStatus, và paymentError. Mỗi đơn hàng đi qua nhiều trạng thái khác nhau, bao gồm pending, payment\_pending, paid, payment\_failed, và completed, phản ánh quy trình xử lý đơn hàng đầy đủ.

```

1 @Entity()
2 export class Order {
3     @PrimaryGeneratedColumn('uuid')
4     id: string;
5
6     @Column()
7     productId: string;
8
9     @Column({ nullable: true })
10    customerId: string;
11
12    @Column()
13    quantity: number;
14
15    @Column()
16    status: string; // pending, payment_pending, paid, payment_failed,
17                    completed
18
19    @Column({ nullable: true })
20    paymentId: string;
21
22    @Column({ nullable: true })
23    paymentStatus: string;
24
25    @Column({ nullable: true })
26    paymentError: string;
27
28    @CreateDateColumn()
29    createdAt: Date;
30
31    @UpdateDateColumn()
32    updatedAt: Date;
33 }

```

Order Service cung cấp hai phương thức khác nhau để tạo đơn hàng: đồng bộ và bất đồng bộ. Trong phương thức đồng bộ (createOrderSync), service kiểm tra tồn kho bằng cách gửi yêu cầu HTTP đến Inventory Service và đợi phản hồi trước khi tiếp tục. Nếu tồn kho đủ, service cập nhật tồn kho và tạo đơn hàng mới với trạng thái confirmed.

```

1 async createOrderSync(data: { productId: string; quantity: number })
2 {
3     this.validateQuantity(data.quantity);
4
5     try {
6         return await this.orderRepository.manager.transaction(async (
7             manager) => {
8             const response = await firstValueFrom(
9                 this.httpService
10                    .get<InventoryResponse>(

```

```

9         `${this.inventoryBaseUrl}/inventory/check/${data.
productId}`,
10         { ...HTTP_CONFIG },
11     )
12     .pipe(/* handling error */),
13 );
14
15     if (response.data.quantity < data.quantity) {
16         throw new BadRequestException('Insufficient inventory');
17     }
18
19     const order = manager.create(Order, {
20         productId: data.productId,
21         quantity: data.quantity,
22         status: 'created',
23     });
24
25     try {
26         await firstValueFrom(
27             this.httpService
28                 .post<InventoryResponse>(
29                     `${this.inventoryBaseUrl}/inventory/update`,
30                     {
31                         productId: data.productId,
32                         quantity: data.quantity,
33                         orderId: order.id,
34                     },
35                     { ...HTTP_CONFIG },
36                 )
37                 .pipe(retry(3)),
38         );
39         order.status = 'confirmed';
40     } catch (error) {
41         order.status = 'failed';
42         throw new ServiceUnavailableException('Failed to update
inventory');
43     }
44     await manager.save(Order, order);
45
46     return order;
47 });
48 } catch (error) {
49     this.logger.error(`Order creation failed: ${(error as Error).
message}`);
50 }
51 }

```

Trong phương thức bất đồng bộ (createOrderAsync), service gửi một thông điệp đến Inventory Service thông qua RabbitMQ và không đợi phản hồi ngay lập tức. Order được tạo với trạng thái pending và sẽ được cập nhật khi nhận được phản hồi từ Inventory Service.

```
1  async createOrderAsync(data: { productId: string; quantity: number
    }) {
2      this.validateQuantity(data.quantity);
3      const order = this.orderRepository.create({
4          productId: data.productId,
5          quantity: data.quantity,
6          status: 'pending',
7      });
8
9      await this.orderRepository.save(order);
10
11     try {
12         void firstValueFrom(
13             this.inventoryClient
14                 .send<InventoryResponse>('check_update_inventory', {
15                     productId: data.productId,
16                     quantity: data.quantity,
17                 })
18                 .pipe(/* handling error */),
19         )
20         .then(response => {
21             if (!response.isAvailable) {
22                 order.status = 'failed';
23                 void this.orderRepository.save(order);
24                 throw new BadRequestException('Insufficient inventory');
25             } else {
26                 order.status = 'confirmed';
27                 void this.orderRepository.save(order);
28             }
29         })
30         .catch(error => {
31             this.logger.error(`Async order failed: ${error}`);
32             order.status = 'failed';
33             void this.orderRepository.save;
34         });
35     } return order;
36 } catch (error) {
37     this.logger.error(`Async order failed: ${error}`);
38     order.status = 'failed';
39     await this.orderRepository.save(order);
40     return order;
41 }
42 }
```

Order Service cũng triển khai streaming API để client có thể theo dõi trạng thái đơn hàng theo thời gian thực. API này sử dụng Server-Sent Events (SSE) để gửi cập nhật cho client khi trạng thái đơn hàng thay đổi.

Inventory Service quản lý tồn kho sản phẩm, cung cấp khả năng kiểm tra và cập nhật số lượng tồn kho. Service này cung cấp cả endpoint RESTful đồng bộ và handler bất đồng bộ cho RabbitMQ.

Thực thể Inventory được định nghĩa với các trường như id, productId, quantity, và isAvailable. Trường isAvailable được sử dụng để đánh dấu nhanh xem sản phẩm có sẵn để đặt hàng hay không.

```
1 @Entity()
2 export class Inventory {
3   @PrimaryGeneratedColumn('uuid')
4   id: string;
5
6   @Column({ name: 'product_id' })
7   productId: string;
8
9   @Column()
10  quantity: number;
11
12  @Column({ default: true, name: 'is_available' })
13  isAvailable: boolean;
14 }
```

Trong giao tiếp đồng bộ, Inventory Service cung cấp endpoint /inventory/check/productId để kiểm tra tồn kho và endpoint /inventory/update để cập nhật tồn kho. Các endpoint này được gọi trực tiếp từ Order Service thông qua HTTP.

```
1 @Get('check/:productId')
2 async checkInventorySync(@Param('productId') productId: string) {
3   return this.inventoryService.checkInventory({ productId, quantity:
4     1 });
5 }
6
7 @Post('update')
8 async updateInventorySync(@Body() updateInventoryDto:
9   UpdateInventoryDto) {
10   return this.inventoryService.updateInventory(updateInventoryDto);
11 }
```

Trong giao tiếp bất đồng bộ, Inventory Service xử lý thông điệp từ RabbitMQ thông qua handler handleCheckInventory. Handler này thực hiện cả việc kiểm tra và cập nhật tồn kho trong một giao dịch.

```
1 @MessagePattern('check_update_inventory')
2 async handleCheckInventory(data: CheckInventoryDto) {
```

```

3   const checkResult = await this.inventoryService.checkInventory(
      data);
4   if (checkResult.isAvailable) {
5       return this.inventoryService.updateInventory(data);
6   }
7   return checkResult;
8 }

```

Payment Service Payment Service xử lý thanh toán cho đơn hàng và cập nhật trạng thái thanh toán. Service này mô phỏng tương tác với một cổng thanh toán bên ngoài, với thời gian xử lý từ 3 đến 5 giây và tỷ lệ thành công 90%.

Thực thể Payment được định nghĩa với các trường như id, orderId, quantity, currency, status, transactionId, và errorMessage. Trường status có thể là pending, processing, completed, hoặc failed, phản ánh trạng thái xử lý thanh toán.

```

1  @Entity()
2  export class Payment {
3      @PrimaryGeneratedColumn('uuid')
4      id: string;
5
6      @Column()
7      orderId: string;
8
9      @Column()
10     quantity: number;
11
12     @Column({ default: 'USD' })
13     currency: string;
14
15     @Column()
16     status: string; // pending, processing, completed, failed
17
18     @Column({ nullable: true })
19     transactionId: string;
20
21     @Column({ nullable: true })
22     errorMessage: string;
23
24     @CreateDateColumn()
25     createdAt: Date;
26
27     @UpdateDateColumn()
28     updatedAt: Date;
29 }

```

Payment Service cung cấp hai phương thức để xử lý thanh toán: đồng bộ và bất đồng bộ. Trong phương thức đồng bộ, client gửi yêu cầu đến endpoint /payment/process và



đội phản hồi. Trong phương thức bất đồng bộ, client gửi yêu cầu đến RabbitMQ và nhận phản hồi thông qua callback.

```
1 @Post('process')
2 async processPaymentSync(
3   @Body() processPaymentDto: ProcessPaymentDto,
4 ): Promise<PaymentResponseDto> {
5   return this.paymentService.processPayment(processPaymentDto, true)
6   ;
7 }
8 @MessagePattern(PAYMENT_PATTERNS.PROCESS_PAYMENT)
9 async processPaymentAsync(
10   processPaymentDto: ProcessPaymentDto,
11 ): Promise<PaymentResponseDto> {
12   return this.paymentService.processPayment(processPaymentDto, false
13   );
14 }
```

Việc xử lý thanh toán được thực hiện trong processPayment, mô phỏng tương tác với cổng thanh toán bên ngoài. Phương thức này tạo một bản ghi thanh toán, gọi mô phỏng cổng thanh toán, cập nhật trạng thái thanh toán, và gửi kết quả về client (trong trường hợp đồng bộ) hoặc gửi callback đến Order Service (trong trường hợp bất đồng bộ).

```
1 async processPayment(
2   processPaymentDto: ProcessPaymentDto,
3   isSync: boolean,
4 ): Promise<PaymentResponseDto> {
5   const payment = this.paymentRepository.create({
6     orderId: processPaymentDto.orderId,
7     quantity: processPaymentDto.quantity,
8     currency: processPaymentDto.currency,
9     status: 'pending',
10   });
11
12   await this.paymentRepository.save(payment);
13
14   try {
15     const result = await this.processWithExternalGateway(payment);
16
17     payment.status = result.success ? 'completed' : 'failed';
18     payment.transactionId = result.transactionId;
19     payment.errorMessage = result.message;
20
21     await this.paymentRepository.save(payment);
22
23     const response = {
24       success: result.success,
```

```

25     transactionId: result.transactionId,
26     message: result.message,
27     paymentId: payment.id,
28     status: payment.status,
29 };
30
31 if (!isSync) {
32     try {
33         await firstValueFrom(
34             this.orderClient.emit(PAYMENT_PATTERNS.PAYMENT_CALLBACK, {
35                 orderId: payment.orderId,
36                 payload: response,
37             }),
38         );
39     } catch (error) {
40         console.error(
41             `Failed to send callback for order ${payment.orderId}: ${
42                 error
43             }`,
44         );
45     }
46
47     return response;
48 } catch (error) {
49     payment.status = 'failed';
50     payment.errorMessage = (error as Error).message;
51     await this.paymentRepository.save(payment);
52
53     const response = {
54         success: false,
55         message: 'Payment processing failed',
56         paymentId: payment.id,
57         status: 'failed',
58     };
59
60     if (!isSync) {
61         await firstValueFrom(
62             this.orderClient.emit(PAYMENT_PATTERNS.PAYMENT_CALLBACK, {
63                 orderId: payment.orderId,
64                 payload: response,
65             }),
66         );
67     }
68
69     return response;
70 }

```

Notification Services Nhóm các dịch vụ thông báo bao gồm Email Service, Notification Service và Analytics Service, đều nhận sự kiện từ Order Service và xử lý chúng theo cách riêng.

Email Service gửi email thông báo đến khách hàng, mô phỏng tương tác với một dịch vụ email bên ngoài như SendGrid hoặc AWS SES. Service này cung cấp cả endpoint RESTful cho giao tiếp đồng bộ và handler cho Kafka cho giao tiếp bất đồng bộ.

```
1 @Post()
2 async sendEmail(
3   @Body()
4   emailData: {
5     orderId: string;
6     customerId: string;
7     subject: string;
8     body: string;
9   },
10 ) {
11   this.logger.log(
12     `Received sync email request for order: ${emailData.orderId}`,
13   );
14   // Simulate processing time
15   await new Promise((resolve) => setTimeout(resolve, 500));
16   return this.emailService.sendEmail(emailData);
17 }
18
19 @EventPattern('order_confirmed')
20 async handleOrderConfirmed(data: {
21   orderId: string;
22   customerId: string;
23   status: string;
24   productId: string;
25   quantity: number;
26   timestamp: string;
27 }) {
28   this.logger.log(`Received async event for order: ${data.orderId}`)
29   ;
30
31   return this.emailService.sendEmail({
32     orderId: data.orderId,
33     customerId: data.customerId,
34     subject: `Order Confirmation: #${data.orderId}`,
35     body: `Thank you for your order #${data.orderId}...`,
36   });
37 }
```

Tương tự, Notification Service gửi thông báo đến khách hàng, trong khi Analytics Service ghi lại và phân tích sự kiện đơn hàng. Cả hai service này cũng cung cấp cả endpoint RESTful và handler Kafka.

### 4.2.3. Triển khai các mẫu giao tiếp

Giao tiếp đồng bộ (REST API) Giao tiếp đồng bộ được triển khai thông qua REST API, sử dụng module `HttpModule` của NestJS. Trong mô hình này, một service gửi yêu cầu HTTP đến service khác và đợi phản hồi trước khi tiếp tục xử lý.

Ví dụ, khi xử lý đơn hàng đồng bộ, Order Service gửi yêu cầu đến Inventory Service để kiểm tra tồn kho, đợi phản hồi, sau đó gửi yêu cầu khác để cập nhật tồn kho. Quá trình này đảm bảo tính nhất quán dữ liệu, nhưng có thể gây ra hiện tượng nghẽn cổ chai và điểm thất bại duy nhất.

```
1 const response = await firstValueFrom(  
2   this.httpService  
3     .get<InventoryResponse>(  
4       `${this.inventoryBaseUrl}/inventory/check/${data.productId}`,  
5       { ...HTTP_CONFIG },  
6     )  
7     .pipe(/* error handling */),  
8 );  
9  
10 if (response.data.quantity < data.quantity) {  
11   throw new BadRequestException('Insufficient inventory');  
12 }  
13  
14 // Update inventory  
15 await firstValueFrom(  
16   this.httpService  
17     .post<InventoryResponse>(  
18       `${this.inventoryBaseUrl}/inventory/update`,  
19       {  
20         productId: data.productId,  
21         quantity: data.quantity,  
22         orderId: order.id,  
23       },  
24       { ...HTTP_CONFIG },  
25     )  
26     .pipe(retry(3)),  
27 );
```

Để cải thiện khả năng chịu lỗi, dự án đã triển khai Circuit Breaker pattern sử dụng thư viện `opossum`. Pattern này ngăn chặn các yêu cầu đến service không khả dụng, giảm thiểu tác động của lỗi dịch vụ.

```
1 createBreaker(name: string, options: CircuitBreaker.Options = {}) {  
2   if (!this.breakers.has(name)) {  
3     const defaultOptions: CircuitBreaker.Options = {  
4       timeout: 5000,  
5       errorThresholdPercentage: 50,  
6       resetTimeout: 10000,  
7       rollingCountTimeout: 10000,
```

```

8     rollingCountBuckets: 10,
9     ...options,
10  };
11
12  const breaker = new CircuitBreaker(async function
13    T,
14    Args extends unknown[],
15    >(fn: (...args: Args) => Promise<T> | T, ...args: Args): Promise
16    <T> {
17      return fn(...args) as Promise<T>;
18    }, defaultOptions);
19
20  breaker?.on('open', () => {
21    this.logger.warn(`Circuit Breaker '${name}' is open`);
22  });
23
24  // Other event handlers...
25
26  this.breakers.set(name, breaker);
27
28  return this.breakers.get(name);
29 }
30
31 async fire<T, Args extends unknown[]>(<
32   name: string,
33   fn: (...args: Args) => Promise<T> | T,
34   ...args: Args
35 >): Promise<T> {
36   const breaker = this.createBreaker(name);
37   return (await breaker.fire(fn, ...args)) as T;
38 }

```

Giao tiếp bất đồng bộ dạng một-một (RabbitMQ) Giao tiếp bất đồng bộ dạng một-một được triển khai sử dụng RabbitMQ thông qua module ClientsModule của NestJS với transport Transport.RMQ. Trong mô hình này, một service gửi thông điệp đến một hàng đợi, và một service khác tiêu thụ thông điệp từ hàng đợi đó.

Ví dụ, khi xử lý đơn hàng bất đồng bộ, Order Service gửi thông điệp đến Inventory Service để kiểm tra và cập nhật tồn kho. Order Service không đợi phản hồi ngay lập tức, mà tiếp tục xử lý. Khi Inventory Service hoàn thành xử lý, nó gửi phản hồi thông qua một hàng đợi khác.

```

1 // Order Service
2 void firstValueFrom(
3   this.inventoryClient
4   .send<InventoryResponse>('check_update_inventory', {
5     productId: data.productId,
6     quantity: data.quantity,

```

```

7     })
8     .pipe(/* error handling */),
9 )
10 .then(response => {
11     if (!response.isAvailable) {
12         order.status = 'failed';
13         void this.orderRepository.save(order);
14         throw new BadRequestException('Insufficient inventory');
15     } else {
16         order.status = 'confirmed';
17         void this.orderRepository.save(order);
18     }
19 })
20 .catch(error => {
21     this.logger.error(`Async order failed: ${error}`);
22     order.status = 'failed';
23     void this.orderRepository.save;
24 });
25
26 // Inventory Service
27 @MessagePattern('check_update_inventory')
28 async handleCheckInventory(data: CheckInventoryDto) {
29     const checkResult = await this.inventoryService.checkInventory(
30         data);
31     if (checkResult.isAvailable) {
32         return this.inventoryService.updateInventory(data);
33     }
34     return checkResult;
35 }

```

Cấu hình RabbitMQ được đặt trong module của mỗi service, chỉ định URL kết nối, tên hàng đợi và các tùy chọn khác.

```

1 ClientsModule.register([
2     {
3         name: 'INVENTORY_SERVICE',
4         transport: Transport.RMQ,
5         options: {
6             urls: [
7                 process.env.RABBITMQ_URL || 'amqp://guest:guest@localhost:5672',
8             ],
9             queue: 'inventory_queue',
10            queueOptions: {
11                durable: true,
12            },
13        },
14    },
15    // Other clients...

```

16 `]),`

Giao tiếp bất đồng bộ dạng một-nhiều (Kafka) Giao tiếp bất đồng bộ dạng một-nhiều được triển khai sử dụng Kafka thông qua module `ClientsModule` của NestJS với transport `Transport.KAFKA`. Trong mô hình này, một service xuất bản sự kiện lên một topic Kafka, và nhiều service đăng ký nhận sự kiện từ topic đó.

Ví dụ, khi một đơn hàng được xác nhận, Order Service xuất bản sự kiện `order_confirmed` lên Kafka. Email Service, Notification Service và Analytics Service đều đăng ký nhận sự kiện này và xử lý nó độc lập.

```
1 // Order Service
2 async notifyServicesAsync(order: Order) {
3   const startTime = Date.now();
4
5   try {
6     // Publish single event to Kafka
7     await firstValueFrom(
8       this.kafkaClient
9         .emit('order_confirmed', {
10           orderId: order.id,
11           customerId: order.customerId,
12           status: order.status,
13           productId: order.productId,
14           quantity: order.quantity,
15           timestamp: new Date().toISOString(),
16         })
17       .pipe(/* error handling */),
18     );
19
20     return {
21       success: true,
22       time: Date.now() - startTime,
23     };
24   } catch (error) {
25     return {
26       success: false,
27       time: Date.now() - startTime,
28       error: error as string,
29     };
30   }
31 }
32
33 // Email Service
34 @EventPattern('order_confirmed')
35 async handleOrderConfirmed(data: {
36   orderId: string;
37   customerId: string;
38   status: string;
```

```

39  productId: string;
40  quantity: number;
41  timestamp: string;
42 }) {
43  this.logger.log(`Received async event for order: ${data.orderId}`)
    ;
44
45  return this.emailService.sendEmail({
46    orderId: data.orderId,
47    customerId: data.customerId,
48    subject: `Order Confirmation: #${data.orderId}`,
49    body: `Thank you for your order...`,
50  });
51 }

```

Cấu hình Kafka được đặt trong module của mỗi service, chỉ định thông tin broker, client ID và các tùy chọn khác.

```

1  ClientsModule.register([
2    {
3      name: 'KAFKA_SERVICE',
4      transport: Transport.KAFKA,
5      options: {
6        client: {
7          clientId: 'order',
8          brokers: [process.env.KAFKA_BROKERS || 'localhost:9092'],
9        },
10       consumer: {
11         groupId: 'order-consumer',
12         allowAutoTopicCreation: true,
13         sessionTimeout: 30000,
14         maxInFlightRequests: 100,
15       },
16       producer: {
17         allowAutoTopicCreation: true,
18       },
19     },
20   ],
21   // Other clients...
22 ],

```

#### 4.2.4. Thiết lập môi trường kiểm thử

Để đánh giá một cách khách quan và toàn diện hiệu suất của các mẫu giao tiếp trong kiến trúc microservice, nghiên cứu đã thiết lập một môi trường kiểm thử chuyên biệt. Môi trường này được xây dựng dựa trên các tiêu chuẩn và phương pháp luận trong lĩnh vực đánh giá hiệu suất hệ thống phân tán. sử dụng công cụ k6, một nền tảng kiểm thử



hiệu suất mã nguồn mở được cộng đồng công nghệ đánh giá cao, để thực hiện các bài kiểm thử.

Khóa luận đã phát triển các script kiểm thử riêng cho từng kịch bản nghiệp vụ, mỗi script đo lường hiệu suất của cả giao tiếp đồng bộ và bất đồng bộ. Các chỉ số được đo lường bao gồm thời gian phản hồi, thông lượng, tỷ lệ lỗi, và sử dụng tài nguyên.

```
1 export const options = {
2   scenarios: {
3     sync_test: {
4       executor: 'ramping-arrival-rate',
5       startRate: 1,
6       timeUnit: '1s',
7       preAllocatedVUs: 5,
8       maxVUs: 10,
9       stages: [
10        { duration: '30s', target: 5 },
11        { duration: '1m', target: 5 },
12        { duration: '30s', target: 0 },
13      ],
14       exec: 'syncTest',
15     },
16     async_test: {
17       executor: 'ramping-arrival-rate',
18       startRate: 1,
19       timeUnit: '1s',
20       preAllocatedVUs: 5,
21       maxVUs: 10,
22       stages: [
23        { duration: '30s', target: 5 },
24        { duration: '1m', target: 5 },
25        { duration: '30s', target: 0 },
26      ],
27       exec: 'asyncTest',
28       startTime: '2m30s'
29     },
30     // Other scenarios...
31   },
32   thresholds: {
33     http_req_duration: ['p(95)<3000', 'p(99)<5000'],
34     http_req_failed: ['rate<0.01'],
35     // Other thresholds...
36   },
37 };

```

Bài đánh giá cũng đã thiết lập giám sát hệ thống sử dụng Prometheus và Grafana để thu thập và hiển thị dữ liệu hiệu suất theo thời gian thực. Điều này cho phép bài đánh giá phân tích hành vi hệ thống dưới các mức tải khác nhau và xác định các điểm nghẽn có thể có.

```

1 function getSystemMetrics(serviceName) {
2   try {
3     const cpuResponse = http.get(
4       `${MONITORING_URL}/api/v1/query?query=
process_cpu_seconds_total{service="${serviceName}"}`,
5       { headers: { Accept: "application/json" }, timeout: "2s" }
6     );
7
8     const memoryResponse = http.get(
9       `${MONITORING_URL}/api/v1/query?query=
process_resident_memory_bytes{service="${serviceName}"}`,
10      { headers: { Accept: "application/json" }, timeout: "2s" }
11    );
12
13    // Process and return metrics...
14  } catch (e) {
15    console.log(`Error fetching metrics: ${e}`);
16  }
17
18  return { cpu: 0, memory: 0 };
19 }

```

Việc thiết lập này cho phép bài đánh giá thu thập dữ liệu toàn diện về hiệu suất của các mẫu giao tiếp khác nhau trong các kịch bản thực tế, cung cấp cơ sở vững chắc cho việc đánh giá và so sánh.

## 4.3. Kết quả triển khai

### 4.3.1. Phương pháp đánh giá

Việc đánh giá các mẫu giao tiếp trong hệ thống microservice được thực hiện dựa trên bộ tiêu chí cụ thể, thiết kế để đo lường hiệu suất, độ tin cậy và khả năng mở rộng của từng phương pháp. Các kịch bản kiểm thử được thiết kế tương ứng với các tình huống thực tế phổ biến mà hệ thống microservice thường gặp phải.

Tiêu chí đánh giá chính bao gồm độ trễ (thời gian phản hồi trung bình, P95 và thời gian xử lý end-to-end), thông lượng (số lượng yêu cầu xử lý được trong một đơn vị thời gian), sử dụng tài nguyên (CPU và bộ nhớ), tính nhất quán dữ liệu và khả năng chịu lỗi của hệ thống. Các công cụ được sử dụng trong quá trình đánh giá bao gồm k6 để mô phỏng lưu lượng người dùng và đo lường các chỉ số hiệu suất, Prometheus để thu thập và lưu trữ số liệu về hiệu suất hệ thống, và Grafana để trực quan hóa các số liệu thu thập.

Các kịch bản kiểm thử được thiết kế đặc biệt để đánh giá hiệu suất của các mẫu giao tiếp trong bốn tình huống phổ biến: kiểm tra và cập nhật tồn kho, xử lý thanh toán, thông báo kết quả đơn hàng, và ghi nhận hoạt động người dùng. Mỗi kịch bản đều so sánh các mẫu giao tiếp khác nhau trong cùng một ngữ cảnh để đánh giá ưu nhược điểm của từng phương pháp.

### 4.3.2. Kết quả đánh giá Order-Inventory

Kết quả đo lường hiệu suất giữa phương pháp đồng bộ (REST) và bất đồng bộ (Message Queue) trong quá trình kiểm tra và cập nhật tồn kho cho thấy phương pháp bất đồng bộ có thời gian phản hồi ban đầu nhanh hơn đáng kể (72%), với thời gian phản hồi trung bình chỉ 2.91ms so với 10.38ms của phương pháp đồng bộ. Tuy nhiên, về tổng thời gian xử lý end-to-end, phương pháp đồng bộ nhanh hơn 28% (10.38ms so với 14.39ms).

**Bảng 4.1. Kết quả đo lường hiệu suất Order-Inventory**

Tiêu chí	Synchronous (REST)	Asynchronous (MQ)	Khác biệt
Average Response Time	10.38ms	2.91ms	Async nhanh hơn 72%
95th Percentile Response Time	18.85ms	4.27ms	Async nhanh hơn 77%
End-to-End Processing Time	10.38ms	14.39ms	Sync nhanh hơn 28%
Throughput	106.64 req/s	89.64 msg/s	Sync cao hơn 16%
CPU Usage	0.0022%	0.00036%	Async ít hơn 84%
Memory Usage	142.59MB	117.78MB	Async ít hơn 17%

Về mặt thông lượng, phương pháp đồng bộ cho kết quả cao hơn một chút (106.64 req/s so với 89.64 msg/s), nhưng lại tiêu tốn nhiều tài nguyên CPU hơn đáng kể (cao hơn 84%). Phương pháp bất đồng bộ cũng sử dụng ít bộ nhớ hơn (117.78MB so với 142.59MB).

Đánh giá tính nhất quán dữ liệu cho thấy phương pháp bất đồng bộ có tỷ lệ nhất quán dữ liệu cao hơn (97.2% so với 93.9%) và ít lỗi hơn (28 so với 61 trường hợp), mặc dù có độ trễ dữ liệu nhỏ (khoảng 12-15ms). Cả hai phương pháp đều cho thấy sự suy giảm hiệu suất khi tải tăng lên, nhưng phương pháp bất đồng bộ duy trì tỷ lệ nhất quán cao hơn ở tất cả các mức tải, từ 10 đến 100 người dùng đồng thời.

### 4.3.3. Kết quả đánh giá Order-Payment

Trong kịch bản xử lý thanh toán, sự khác biệt về hiệu suất giữa hai phương pháp càng trở nên rõ rệt. Phương pháp bất đồng bộ có thời gian phản hồi ban đầu nhanh hơn 99.8% (2.25ms so với 1508.14ms) và thông lượng cao hơn 52 lần (90.04 msg/s so với

**Bảng 4.2. Kết quả đánh giá tính nhất quán dữ liệu**

Tiêu chí	Synchronous	Asynchronous	Khác biệt
Data Consistency Rate	93.9%	97.2%	Async tốt hơn 3.3%
Failed Requests	61	28	Async ít lỗi hơn 54%
Data Lag	0ms	12.09ms	Sync nhanh hơn
Eventual Consistency Time (P95)	0ms	15ms	Sync nhanh hơn

1.68 req/s). Về tổng thời gian xử lý end-to-end, phương pháp đồng bộ nhanh hơn một chút (4%), đạt 1508.14ms so với 1571.71ms của phương pháp bất đồng bộ.

Phương pháp đồng bộ sử dụng ít CPU hơn 41.1%, trong khi phương pháp bất đồng bộ sử dụng ít bộ nhớ hơn 4.4%. Đáng chú ý là cả hai phương pháp đều có tỷ lệ lỗi tương đương, lần lượt là 4.71% và 5.05%.

Đối với xử lý thanh toán thời gian dài, phương pháp bất đồng bộ vẫn duy trì thời gian phản hồi ban đầu rất nhanh (2.77ms so với 3773.11ms của phương pháp đồng bộ) và có thời gian xử lý end-to-end ngắn hơn 7.1%. Cả hai phương pháp đều xử lý được các trường hợp timeout, với phương pháp đồng bộ ghi nhận tỷ lệ timeout 0%, nhưng phương pháp bất đồng bộ có thời gian xử lý nhỏ hơn ở tất cả các mức tải, từ 10 đến tải tối đa.

#### **4.3.4. Kết quả đánh giá Order-Notification**

Trong kịch bản thông báo kết quả đơn hàng, mô hình Pub/Sub thể hiện hiệu suất vượt trội so với phương pháp gọi đồng bộ tuần tự. Thời gian broadcast trung bình của mô hình Pub/Sub chỉ là 11.53ms, nhanh hơn 97.8% so với 520.55ms của phương pháp gọi đồng bộ tuần tự. Thời gian xử lý mỗi service cũng nhanh hơn đáng kể, chỉ 9.57ms so với 350.40ms.

Mô hình Pub/Sub cũng sử dụng ít tài nguyên hơn, với mức tiêu thụ CPU thấp hơn 73.5% và bộ nhớ thấp hơn nhẹ 0.7%. Cả hai phương pháp đều đạt tỷ lệ thành công 100% trong các bài kiểm thử.

Về khả năng chịu lỗi khi một service thất bại, cả hai phương pháp đều đã được cải tiến để không còn hiện tượng lỗi lan truyền giữa các service. Tuy nhiên, mô hình Pub/Sub vẫn cho thấy ưu điểm vượt trội về thời gian phục hồi, chỉ 2.30ms so với 4793.46ms của phương pháp gọi đồng bộ tuần tự, nhanh hơn 99.95%. Tỷ lệ phục hồi thành công của mô hình Pub/Sub cũng cao hơn, đạt 100% so với 74.5% của phương pháp gọi đồng bộ tuần tự.

**Bảng 4.3. Kết quả đo lường hiệu suất Order-Payment**

Tiêu chí	Synchronous	Asynchronous	Khác biệt
Average Response Time	1508.14ms	2.25ms	Async nhanh hơn 99.8%
95th Percentile Response Time	2856.27ms	4.77ms	Async nhanh hơn 99.8%
End-to-End Processing Time	1508.14ms	1571.71ms	Sync nhanh hơn 4.0%
Throughput	1.68 req/s	90.04 msg/s	Async cao hơn 5259%
CPU Usage	0.0083%	0.0141%	Sync ít hơn 41.1%
Memory Usage	114.70MB	109.67MB	Async ít hơn 4.4%

#### 4.3.5. Kết quả đánh giá User Activity Logging

Kết quả kiểm thử ghi nhận hoạt động người dùng cho thấy cả Kafka (mô hình một-nhiều) và RabbitMQ (mô hình một-một) đều có hiệu suất tương đương trong việc phân phối dữ liệu. Tổng thời gian phân phối của Kafka là 507.12ms, nhanh hơn nhẹ 0.3% so với 508.56ms của RabbitMQ. Thời gian phản hồi P95 cũng rất tương đồng, 509ms so với 510ms.

Cả hai phương pháp đều đạt thông lượng 1.97 msg/s và tỷ lệ thành công 100%. Tuy nhiên, Kafka sử dụng ít tài nguyên hơn, với mức tiêu thụ CPU thấp hơn 15.8% và bộ nhớ thấp hơn 10.5%.

#### 4.3.6. Đánh giá tổng thể

Từ kết quả đánh giá các mẫu giao tiếp trong bốn kịch bản kiểm thử, có thể rút ra một số nhận xét tổng quát. Đối với Order-Inventory, mặc dù phương pháp đồng bộ có thời gian xử lý end-to-end nhỏ hơn và thông lượng cao hơn, phương pháp bất đồng bộ vẫn ưu việt hơn nhờ thời gian phản hồi nhanh, sử dụng ít tài nguyên và có tỷ lệ nhất quán dữ liệu cao hơn, đặc biệt khi tải tăng cao.

Trong kịch bản Order-Payment, phương pháp bất đồng bộ thể hiện ưu thế vượt trội về thời gian phản hồi ban đầu và thông lượng, đặc biệt quan trọng cho trải nghiệm người dùng. Phương pháp này cũng xử lý tốt hơn các trường hợp thanh toán thời gian dài, duy trì thời gian phản hồi nhanh và hiệu suất tổng thể tốt hơn ở tất cả các mức tải.

**Bảng 4.4. Kết quả đo lường hiệu suất Order-Notification**

Tiêu chí	Multiple Calls	Sync Event Bus	Khác biệt
Average Broadcast Time	520.55ms	11.53ms	Pub/Sub nhanh hơn 97.8%
95th Percentile Broadcast Time	537.90ms	6.10ms	Pub/Sub nhanh hơn 98.9%
Maximum Broadcast Time	723ms	204ms	Pub/Sub nhanh hơn 71.8%
Per-Service Time (avg)	350.40ms	9.57ms	Pub/Sub nhanh hơn 97.3%
CPU Usage	0.01173%	0.00311%	Pub/Sub ít hơn 73.5%
Memory Usage	22.98GB	22.83GB	Pub/Sub ít hơn 0.7%
Success Rate	100%	100%	Ngang bằng

Đối với Order-Notification, mô hình Pub/Sub vượt trội hơn hẳn về mọi mặt, từ thời gian broadcast, thời gian xử lý mỗi service đến khả năng phục hồi khi có lỗi. Mô hình này cũng sử dụng ít tài nguyên hơn và dễ dàng mở rộng khi thêm các service nhận thông báo mới.

Cuối cùng, trong kịch bản User Activity Logging, cả Kafka và RabbitMQ đều thể hiện hiệu suất tương đương, với Kafka có lợi thế nhỏ về thời gian phản hồi và sử dụng tài nguyên. Kafka phù hợp hơn cho các trường hợp có nhiều consumer, trong khi RabbitMQ có thể phù hợp hơn cho các trường hợp cần đảm bảo giao tiếp một-một chính xác.

Nhìn chung, các mẫu giao tiếp bất đồng bộ (Message Queue, Pub/Sub, Kafka) cho thấy ưu thế về thời gian phản hồi, khả năng chịu lỗi và hiệu quả sử dụng tài nguyên trong hầu hết các kịch bản. Tuy nhiên, các mẫu giao tiếp đồng bộ vẫn có những ưu điểm nhất định về thời gian xử lý end-to-end và tính nhất quán dữ liệu tức thời, có thể phù hợp cho các tác vụ yêu cầu phản hồi nhanh và đảm bảo dữ liệu nhất quán ngay lập tức.

**Bảng 4.5. Kết quả đo lường hiệu suất User Activity Logging**

<b>Tiêu chí</b>	<b>Kafka (One-to-Many)</b>	<b>RabbitMQ (One-to-One)</b>	<b>Khác biệt</b>
Total Distribution Time	507.12ms	508.56ms	Kafka nhanh hơn 0.3%
95th Percentile Response Time	509ms	510ms	Kafka nhanh hơn 0.2%
Throughput	1.97 msg/s	1.97 msg/s	Không có sự khác biệt
CPU Usage	0.016%	0.019%	Kafka ít hơn 15.8%
Memory Usage	282.44MB	315.72MB	Kafka ít hơn 10.5%
Success Rate	100%	100%	Không có sự khác biệt

# Chương 5.

## Đánh giá và thảo luận

### 5.1. Phương pháp và tiêu chí đánh giá

#### 5.1.1. Tổng quan phương pháp đánh giá

Phương pháp đánh giá trong dự án này được thiết kế để cung cấp một cái nhìn toàn diện về hiệu suất và độ tin cậy của các mẫu giao tiếp khác nhau trong kiến trúc microservice. Quá trình đánh giá tuân theo phương pháp luận khoa học nghiêm ngặt, bao gồm việc thiết lập các tiêu chí đánh giá rõ ràng, triển khai các kịch bản kiểm thử thực tế, và sử dụng các công cụ đo lường hiệu suất chuyên nghiệp.

Dự án xác định bốn kịch bản nghiệp vụ chính trong hệ thống thương mại điện tử: kiểm tra và cập nhật tồn kho, xử lý thanh toán, thông báo kết quả đơn hàng, và ghi nhận hoạt động người dùng. Mỗi kịch bản này đại diện cho một loại tương tác phổ biến trong các hệ thống microservice, với các yêu cầu về hiệu suất và độ tin cậy khác nhau. Việc chọn các kịch bản đa dạng giúp đảm bảo kết quả đánh giá có tính đại diện cao và áp dụng được cho nhiều ngữ cảnh khác nhau.

Dự án triển khai ba mẫu giao tiếp chính: giao tiếp đồng bộ sử dụng REST API, giao tiếp bất đồng bộ dạng một-một sử dụng RabbitMQ, và giao tiếp bất đồng bộ dạng một-nhiều sử dụng Kafka. Mỗi mẫu giao tiếp được triển khai trên cùng một nền tảng hạ tầng và được đánh giá trong cùng các kịch bản, đảm bảo tính công bằng và khách quan của quá trình so sánh.

#### 5.1.2. Tiêu chí đánh giá cụ thể

Các tiêu chí đánh giá được xây dựng dựa trên các yêu cầu phi chức năng quan trọng của hệ thống microservice, phản ánh các khía cạnh chất lượng mà các kiến trúc microservice hiện đại cần đáp ứng. Các tiêu chí này được phân thành bốn nhóm chính: hiệu suất, tài nguyên hệ thống, độ tin cậy và khả năng chịu lỗi.

Về hiệu suất, dự án đánh giá thời gian phản hồi (thời gian từ khi gửi yêu cầu đến khi nhận được phản hồi ban đầu), thời gian xử lý end-to-end (tổng thời gian để hoàn thành toàn bộ quy trình), thông lượng (số lượng yêu cầu xử lý trong một đơn vị thời gian), và thời gian phản hồi phân vị thứ 95 (P95).

Về tài nguyên hệ thống, dự án đánh giá mức sử dụng CPU, bộ nhớ và lưu lượng mạng. Các chỉ số này phản ánh hiệu quả sử dụng tài nguyên của mỗi mẫu giao tiếp, yếu tố quan trọng ảnh hưởng đến chi phí vận hành hệ thống.

Về độ tin cậy, dự án đánh giá tỷ lệ thành công, tỷ lệ lỗi và tỷ lệ nhất quán dữ liệu. Đặc biệt, tỷ lệ nhất quán dữ liệu đo lường khả năng duy trì tính nhất quán giữa các service, yếu tố quan trọng trong các hệ thống phân tán.



Về khả năng chịu lỗi, dự án đánh giá tỷ lệ lan truyền lỗi, thời gian phục hồi và tỷ lệ thành công một phần. Các chỉ số này phản ánh khả năng của hệ thống trong việc duy trì hoạt động khi gặp sự cố.

### **5.1.3. Công cụ và môi trường đánh giá**

Dự án sử dụng một bộ công cụ hiện đại để thực hiện đánh giá. Công cụ k6 được sử dụng để mô phỏng lưu lượng người dùng và đo lường các chỉ số hiệu suất. Prometheus được sử dụng để thu thập và lưu trữ các số liệu về hiệu suất hệ thống. Các công cụ này đều là mã nguồn mở và được sử dụng rộng rãi trong cộng đồng phát triển phần mềm, đảm bảo tính tin cậy và khả năng mở rộng của quá trình đánh giá.

Môi trường kiểm thử được thiết kế để mô phỏng các điều kiện thực tế mà một hệ thống thương mại điện tử có thể gặp phải. Các kịch bản kiểm thử bao gồm các mức tải khác nhau, từ tải nhẹ đến tải nặng, để đánh giá hiệu suất của các mẫu giao tiếp trong nhiều điều kiện khác nhau. Các service được triển khai trên cùng một cấu hình phần cứng, và các bài kiểm thử được thực hiện nhiều lần để đảm bảo tính đại diện thống kê của kết quả.

Kịch bản kiểm thử được thiết kế để tự động thực hiện các tác vụ như tạo đơn hàng, kiểm tra tồn kho, xử lý thanh toán và gửi thông báo. Mỗi kịch bản được thực hiện trên những mẫu giao tiếp thích hợp với kịch bản đấy, và các chỉ số hiệu suất được ghi lại để so sánh.

Việc sử dụng các công cụ và môi trường đánh giá hiện đại giúp đảm bảo tính khách quan và chính xác của kết quả đánh giá. Điều này cho phép dự án đưa ra những kết luận đáng tin cậy về hiệu suất và độ tin cậy của các mẫu giao tiếp trong kiến trúc microservice.

## **5.2. Kết quả đánh giá**

### **5.2.1. Mẫu giao tiếp trong kịch bản Order-Inventory**

Kết quả đánh giá hiệu suất giữa giao tiếp đồng bộ và bất đồng bộ trong kịch bản kiểm tra và cập nhật tồn kho cho thấy những đặc điểm đáng chú ý về cách các mẫu giao tiếp ảnh hưởng đến trải nghiệm người dùng và hiệu quả hệ thống. Giao tiếp bất đồng bộ thể hiện thời gian phản hồi ban đầu nhanh hơn đáng kể, giúp người dùng nhận được phản hồi tức thì khi thực hiện thao tác. Tuy nhiên, giao tiếp đồng bộ lại cho thời gian xử lý end-to-end nhanh hơn, phản ánh đặc tính xử lý trực tiếp, không qua trung gian của phương pháp này.

Về tính nhất quán dữ liệu, một phát hiện thú vị là giao tiếp bất đồng bộ đạt tỷ lệ nhất quán cao hơn, đặc biệt khi hệ thống chịu tải nặng. Điều này có vẻ trái ngược với quan niệm truyền thống rằng giao tiếp đồng bộ đảm bảo nhất quán dữ liệu tốt hơn. Nguyên nhân có thể do cơ chế hàng đợi giúp điều tiết luồng xử lý và tránh tình trạng quá tải, dẫn đến ít lỗi và nhất quán dữ liệu cao hơn.

Về hiệu quả sử dụng tài nguyên, giao tiếp bất đồng bộ thể hiện rõ ràng lợi thế với mức tiêu thụ CPU và bộ nhớ thấp hơn đáng kể. Điều này đặc biệt quan trọng trong môi trường đám mây, nơi chi phí vận hành thường tỷ lệ thuận với tài nguyên tiêu thụ.

### 5.2.2. Mẫu giao tiếp trong kịch bản Order-Payment

Trong kịch bản xử lý thanh toán, sự khác biệt về hiệu suất giữa giao tiếp đồng bộ và bất đồng bộ trở nên rõ rệt hơn. Giao tiếp bất đồng bộ thể hiện thời gian phản hồi ban đầu gần như tức thì, cải thiện đáng kể trải nghiệm người dùng so với phương pháp đồng bộ. Điều này đặc biệt quan trọng trong bối cảnh thanh toán, nơi người dùng mong đợi phản hồi nhanh chóng để biết yêu cầu của họ đã được tiếp nhận.

Thông lượng của giao tiếp bất đồng bộ vượt trội hơn hẳn so với giao tiếp đồng bộ, phản ánh khả năng tiếp nhận nhiều yêu cầu hơn trong cùng một khoảng thời gian. Điều này là do giao tiếp bất đồng bộ không bị chặn bởi thời gian xử lý của mỗi yêu cầu, cho phép hệ thống tiếp tục tiếp nhận yêu cầu mới trong khi các yêu cầu cũ đang được xử lý.

Đối với các trường hợp thanh toán kéo dài, giao tiếp bất đồng bộ vẫn duy trì thời gian phản hồi ban đầu thấp, trong khi giao tiếp đồng bộ buộc người dùng phải đợi đến khi toàn bộ quá trình hoàn tất. Điều này làm cho giao tiếp bất đồng bộ trở thành lựa chọn ưu việt hơn cho các tác vụ có thời gian xử lý dài như thanh toán.

### 5.2.3. Mẫu giao tiếp trong kịch bản Order-Notification

Kết quả đánh giá cho kịch bản thông báo kết quả đơn hàng thể hiện rõ ràng ưu điểm của mô hình Pub/Sub so với phương pháp gọi đồng bộ tuần tự. Mô hình Pub/Sub cung cấp thời gian broadcast và thời gian xử lý mỗi service nhanh hơn đáng kể, giúp thông báo được gửi đến tất cả các kênh nhanh chóng và hiệu quả.

Về khả năng chịu lỗi, mô hình Pub/Sub thể hiện thời gian phục hồi cực kỳ nhanh và tỷ lệ phục hồi thành công cao hơn so với phương pháp gọi đồng bộ tuần tự. Điều này phản ánh kiến trúc lỏng lẻo (loosely coupled) của mô hình Pub/Sub, nơi các subscriber hoạt động độc lập với nhau và với publisher, giúp hệ thống duy trì hoạt động ngay cả khi một số thành phần gặp sự cố.

Hiệu quả sử dụng tài nguyên của mô hình Pub/Sub cũng vượt trội hơn, với mức tiêu thụ CPU thấp hơn đáng kể. Điều này làm cho mô hình Pub/Sub trở thành lựa chọn tiết kiệm chi phí và hiệu quả hơn cho các kịch bản thông báo đa kênh.

### 5.2.4. Mẫu giao tiếp trong kịch bản User Activity Logging

Đối với kịch bản ghi nhận hoạt động người dùng, cả Kafka (mô hình một-nhiều) và RabbitMQ (mô hình một-một) đều thể hiện hiệu suất tương đương về thời gian phân phối và thông lượng. Tuy nhiên, Kafka sử dụng ít tài nguyên hơn và có thể xử lý nhiều consumer hơn một cách hiệu quả, làm cho nó phù hợp hơn cho các kịch bản có nhiều service cần truy cập cùng một dữ liệu.

Kafka cũng cung cấp khả năng lưu trữ dữ liệu dài hạn và phát lại các sự kiện, điều này đặc biệt hữu ích cho các kịch bản phân tích dữ liệu và học máy, nơi dữ liệu lịch sử có giá trị cao. RabbitMQ, mặt khác, phù hợp hơn cho các kịch bản yêu cầu điều hướng thông điệp phức tạp và định tuyến có điều kiện.

### 5.2.5. Ảnh hưởng của tải hệ thống đến các mẫu giao tiếp

Một khía cạnh quan trọng của dự án là đánh giá cách các mẫu giao tiếp hoạt động dưới các mức tải khác nhau. Kết quả cho thấy khi tải tăng, hiệu suất của cả giao tiếp đồng bộ và bất đồng bộ đều giảm, nhưng mức độ giảm khác nhau đáng kể.

Giao tiếp đồng bộ thể hiện sự suy giảm hiệu suất mạnh hơn khi tải tăng, với tỷ lệ nhất quán dữ liệu và thông lượng giảm nhanh. Điều này là do mô hình chặn (blocking model) của giao tiếp đồng bộ, nơi mỗi yêu cầu phải đợi đến khi yêu cầu trước đó hoàn thành. Khi số lượng yêu cầu tăng lên, điều này dẫn đến tình trạng nghẽn cổ chai và suy giảm hiệu suất.

Giao tiếp bất đồng bộ, mặt khác, duy trì hiệu suất ổn định hơn dưới tải nặng, nhờ vào khả năng đệm các yêu cầu và xử lý chúng theo tốc độ phù hợp với tài nguyên có sẵn. Đặc biệt, thời gian phản hồi ban đầu của giao tiếp bất đồng bộ vẫn duy trì ở mức thấp ngay cả khi tải tăng cao, giúp duy trì trải nghiệm người dùng tốt.

Mô hình Pub/Sub cũng thể hiện khả năng mở rộng tốt, với hiệu suất ổn định khi số lượng subscriber tăng lên. Điều này làm cho mô hình Pub/Sub trở thành lựa chọn phù hợp cho các hệ thống cần khả năng mở rộng theo chiều ngang.

## 5.3. Thảo luận

### 5.3.1. Lựa chọn mẫu giao tiếp phù hợp cho từng kịch bản

Dựa trên kết quả đánh giá toàn diện, có thể đưa ra những khuyến nghị về việc lựa chọn mẫu giao tiếp phù hợp cho từng kịch bản nghiệp vụ trong kiến trúc microservice.

Đối với kịch bản kiểm tra và cập nhật tồn kho, giao tiếp bất đồng bộ sử dụng Message Queue (RabbitMQ) được khuyến nghị vì thời gian phản hồi ban đầu nhanh, tỷ lệ nhất quán dữ liệu cao và hiệu quả sử dụng tài nguyên tốt. Mặc dù có thời gian xử lý end-to-end dài hơn một chút, nhưng ưu điểm này được bù đắp bởi khả năng duy trì hiệu suất ổn định dưới tải cao và trải nghiệm người dùng tốt hơn. Độ trễ dữ liệu nhỏ (12-15ms) không đáng kể trong hầu hết các trường hợp sử dụng.

Đối với kịch bản xử lý thanh toán, giao tiếp bất đồng bộ cũng là lựa chọn ưu việt với thời gian phản hồi ban đầu nhanh và thông lượng cao. Trong bối cảnh thanh toán, việc phản hồi nhanh cho người dùng rằng yêu cầu đã được tiếp nhận là rất quan trọng, ngay cả khi quá trình xử lý thực sự có thể kéo dài. Giao tiếp bất đồng bộ cho phép xác nhận ngay lập tức, trong khi vẫn đảm bảo việc xử lý được hoàn thành đúng cách ở phía sau. Đặc biệt với các giao dịch thanh toán thời gian dài, mô hình này giúp giữ trải nghiệm người dùng mượt mà và phản hồi nhanh.

Đối với kịch bản thông báo kết quả đơn hàng, mô hình Pub/Sub sử dụng Kafka là lựa chọn vượt trội với thời gian broadcast nhanh, khả năng phục hồi tốt khi có service gặp lỗi, và hiệu quả sử dụng tài nguyên cao. Mô hình này đặc biệt phù hợp khi một sự kiện cần được xử lý bởi nhiều service độc lập, như gửi email, thông báo đẩy và cập nhật

phân tích. Khả năng mở rộng dễ dàng bằng cách thêm các subscriber mới mà không ảnh hưởng đến publisher là một lợi thế lớn của mô hình này.

Đối với kịch bản ghi nhận hoạt động người dùng, Kafka được khuyến nghị nhẹ so với RabbitMQ vì hiệu quả sử dụng tài nguyên tốt hơn và mô hình một-nhiều phù hợp hơn cho việc phân phối dữ liệu hoạt động đến nhiều service phân tích khác nhau. Khả năng lưu trữ dữ liệu lâu dài và phát lại các sự kiện cũng là những ưu điểm quan trọng trong bối cảnh phân tích dữ liệu.

### 5.3.2. Mô hình tích hợp các mẫu giao tiếp

Trong thực tế, việc kết hợp các mẫu giao tiếp khác nhau thường mang lại kết quả tốt nhất cho hệ thống microservice. Mỗi mẫu giao tiếp có những ưu điểm và nhược điểm riêng, phù hợp với những tình huống cụ thể. Một mô hình tích hợp hiệu quả có thể tận dụng ưu điểm của từng phương pháp để tối ưu hóa hiệu suất tổng thể của hệ thống.

Một chiến lược tích hợp hiệu quả là sử dụng giao tiếp đồng bộ cho các tác vụ yêu cầu phản hồi tức thời và có thời gian xử lý ngắn, như truy vấn thông tin đơn giản. Giao tiếp bất đồng bộ dạng một-một được sử dụng cho các tác vụ có thời gian xử lý dài nhưng cần phản hồi nhanh cho người dùng, như xử lý đơn hàng và thanh toán. Giao tiếp bất đồng bộ dạng một-nhiều được sử dụng cho các tác vụ cần phân phối thông tin đến nhiều service, như thông báo sự kiện và ghi nhận hoạt động.

Trong một hệ thống thương mại điện tử, mô hình tích hợp này có thể được triển khai như sau: REST API được sử dụng cho việc hiển thị thông tin sản phẩm và danh mục, RabbitMQ được sử dụng cho xử lý đơn hàng và thanh toán, và Kafka được sử dụng cho thông báo kết quả đơn hàng và phân tích dữ liệu.

Một yếu tố quan trọng cần xem xét khi thiết kế mô hình tích hợp là độ phức tạp của hệ thống. Việc sử dụng nhiều mẫu giao tiếp khác nhau có thể làm tăng độ phức tạp trong triển khai và bảo trì. Do đó, cần cân nhắc giữa lợi ích hiệu suất và chi phí phức tạp khi quyết định số lượng mẫu giao tiếp cần sử dụng.

### 5.3.3. Tối ưu hóa hiệu suất trong thực tế

Ngoài việc lựa chọn mẫu giao tiếp phù hợp, còn có nhiều chiến lược tối ưu hóa hiệu suất khác có thể áp dụng trong thực tế. Dựa trên kết quả đánh giá, có thể đưa ra một số khuyến nghị sau:

Đối với giao tiếp đồng bộ, việc triển khai Circuit Breaker pattern là rất quan trọng để ngăn chặn lỗi cascade và cải thiện khả năng chịu lỗi của hệ thống. Mẫu này giúp ngăn chặn các yêu cầu đến service không khả dụng, giảm thiểu tác động của lỗi dịch vụ đến toàn bộ hệ thống. Việc sử dụng timeouts hợp lý cũng giúp tránh tình trạng chờ đợi vô hạn khi service gặp sự cố.

Đối với giao tiếp bất đồng bộ, việc điều chỉnh kích thước hàng đợi và số lượng consumer có thể giúp cân bằng giữa thông lượng và độ trễ. Tăng số lượng consumer giúp cải thiện thông lượng, nhưng cũng làm tăng chi phí tài nguyên. Việc triển khai

cơ chế retry với exponential backoff giúp xử lý các lỗi tạm thời, trong khi Dead Letter Queues giúp xử lý các thông điệp không thể xử lý.

Đối với mô hình Pub/Sub, việc phân vùng (partitioning) dữ liệu có thể giúp cải thiện khả năng mở rộng và hiệu suất. Việc chọn số lượng partition phù hợp với số lượng consumer giúp tối ưu hóa cân bằng tải và thông lượng. Việc duy trì kích thước thông điệp nhỏ và sử dụng định dạng nhị phân như Avro hoặc Protobuf thay vì JSON cũng giúp cải thiện hiệu suất.

#### **5.3.4. Xu hướng phát triển và nghiên cứu tiếp theo**

Dựa trên kết quả đánh giá và xu hướng hiện tại trong lĩnh vực, có thể thấy một số hướng phát triển và nghiên cứu tiếp theo đáng chú ý.

Một xu hướng đang phát triển là server-less communication, nơi các hàm serverless được sử dụng để xử lý các sự kiện và thông điệp mà không cần quan tâm đến cơ sở hạ tầng bên dưới. Mô hình này hứa hẹn giảm chi phí vận hành và cải thiện khả năng mở rộng, đặc biệt cho các hệ thống có tải không đều. Tuy nhiên, cần nghiên cứu thêm về hiệu suất và độ tin cậy của mô hình này trong các kịch bản khác nhau.

Một hướng nghiên cứu khác là Service Mesh, một lớp cơ sở hạ tầng chuyên dụng cho giao tiếp service-to-service, cung cấp các tính năng như service discovery, load balancing, encryption, observability, và authentication/authorization. Service Mesh có thể kết hợp các ưu điểm của cả giao tiếp đồng bộ và bất đồng bộ, đồng thời giải quyết một số thách thức về bảo mật và giám sát.

Việc kết hợp các mẫu giao tiếp với các công nghệ stream processing như Apache Flink hoặc Kafka Streams cũng là một hướng nghiên cứu hứa hẹn. Các công nghệ này cho phép xử lý dữ liệu liên tục theo thời gian thực, mở ra khả năng cho các ứng dụng phản ứng nhanh với các sự kiện và thay đổi dữ liệu.

Cuối cùng, việc áp dụng các kỹ thuật học máy và trí tuệ nhân tạo để tự động hóa việc lựa chọn và tối ưu hóa các mẫu giao tiếp dựa trên đặc điểm của tải và yêu cầu nghiệp vụ là một hướng nghiên cứu đầy tiềm năng. Các hệ thống tự thích ứng có thể điều chỉnh mẫu giao tiếp dựa trên các điều kiện hoạt động thực tế, tối ưu hóa hiệu suất và độ tin cậy một cách liên tục.

#### **5.3.5. Các hạn chế của dự án và hướng cải thiện**

Mặc dù dự án đã cung cấp những hiểu biết có giá trị về hiệu suất và độ tin cậy của các mẫu giao tiếp, vẫn có một số hạn chế cần được ghi nhận và cải thiện trong các nghiên cứu tương lai.

Trước hết, các kịch bản kiểm thử được thực hiện trong môi trường kiểm thử có kiểm soát, có thể không hoàn toàn phản ánh các điều kiện hoạt động thực tế. Các yếu tố như độ trễ mạng không đồng đều, sự cố phần cứng, và các vấn đề về bảo mật có thể ảnh hưởng đáng kể đến hiệu suất của các mẫu giao tiếp trong môi trường sản xuất thực tế.

Thứ hai, dự án tập trung chủ yếu vào hiệu suất và độ tin cậy, mà không đi sâu vào các khía cạnh khác như bảo mật, chi phí triển khai và vận hành, và khả năng tích hợp với

các hệ thống hiện có. Những yếu tố này cũng là những cân nhắc quan trọng khi lựa chọn mẫu giao tiếp trong thực tế.

Thứ ba, việc đánh giá chỉ bao gồm ba mẫu giao tiếp chính, trong khi có nhiều mẫu và biến thể khác cũng được sử dụng trong thực tế, như gRPC, GraphQL, và WebSockets. Các mẫu này có thể có những ưu điểm đặc thù trong một số kịch bản cụ thể.

Cuối cùng, dự án không đánh giá hiệu suất của các mẫu giao tiếp trong thời gian dài, có thể làm thiếu sót các vấn đề như rò rỉ bộ nhớ, suy giảm hiệu suất theo thời gian, và khả năng phục hồi sau sự cố lớn.

Để cải thiện, các nghiên cứu tương lai có thể mở rộng phạm vi đánh giá để bao gồm nhiều mẫu giao tiếp hơn, các kịch bản đa dạng hơn, và thời gian kiểm thử dài hơn. Việc thực hiện các bài kiểm thử trong các môi trường đám mây thực tế với các yếu tố nhiễu như bị phát giả, tấn công DDoS, và sự cố phần cứng cũng sẽ cung cấp những hiểu biết có giá trị về độ tin cậy của các mẫu giao tiếp trong điều kiện khắc nghiệt.

## Tài liệu tham khảo

### Tiếng Anh

- [1] International Data Corporation (IDC), “IDC FutureScape: Worldwide IT Industry 2021 Predictions”, *IDC #US46942020*, October 2020.
- [2] Gartner, Inc., “Gartner Identifies Key Trends in PaaS and Platform Architecture for Application Leaders”, *Gartner Press Release*, April 2019.
- [3] Jun Hong, X. et al., “Performance Analysis of RESTful API and RabbitMQ for Microservice Web Application”, *IEEE ICTC*, 2018.
- [4] Richardson, C., *Microservices Patterns*, Manning Publications, 2019.
- [5] Newman, S., *Building Microservices*, O’Reilly Media, 2015.
- [6] Karabey Aksakalli, I., Çelik, T., Can, A. B., & Tekinerdoğan, B., “Deployment and communication patterns in microservice architectures: A systematic literature review”, *Journal of Systems and Software*, Vol. 180, 2021, pp. 111014.
- [7] Wolff, E., *Microservices: Flexible Software Architecture*, Addison-Wesley Professional, 2016.
- [8] Fowler, M., “Microservices”, <https://martinfowler.com/articles/microservices.html>, 2014.
- [9] Hohpe, G., & Woolf, B., *Enterprise Integration Patterns*, Addison-Wesley, 2004.
- [10] Indrasiri, K., & Kuruppu, D., *gRPC: Up and Running: Building Cloud Native Applications with Go and Java for Docker and Kubernetes*, O’Reilly Media, 2020.
- [11] Wittern, E., Cha, A., & Davis, J. C., “GraphQL: A data query language”, *IBM Research*, 2018.
- [12] Beyer, M. et al., “Uber’s Microservice Architecture”, *eng.uber.com*, 2018.
- [13] Goodhope, K. et al., “Building LinkedIn’s Real-time Activity Data Pipeline”, *IEEE Data Eng. Bull.*, Vol. 35, No. 2, 2012.
- [14] Raman, A., “PayPal’s Journey to Microservices: Building the Payments Platform of the Future”, *QCon Conference*, 2016.
- [15] Fateev, M., & Tolstoi, S., “Uber Cadence: Fault Tolerant Actor Framework for Distributed Applications”, *Uber Engineering Blog*, 2017.
- [16] Fowler, M., *Patterns of Enterprise Application Architecture*, Addison-Wesley Professional, 2002.