HỌC VIỆN CÔNG NGHỆ BƯU CHÍNH VIỄN THÔNG

PHẠM TUẤN PHONG

PHÁT HIỆN VÀ KHẮC PHỤC LỖI TRONG HỆ THỐNG MICROSERVICES

Chuyên ngành: Hệ thống thông tin
Mã số: 8.48.01.04

ĐỀ ÁN TỐT NGHIỆP THẠC SĨ KỸ THUẬT
(HỆ THỐNG THÔNG TIN)
(Theo định hướng ứng dụng)

NGƯỜI HƯỚNG DẪN KHOA HỌC
TS. NGUYỄN DUY PHƯƠNG

HÀ NỘI - 2025

LỜI CAM ĐOAN

Tôi xin cam đoan đây là công trình nghiên cứu của riêng tôi.

Các số liệu, kết quả nêu trong đề án tốt nghiệp là trung thực và chưa từng được ai công bố trong bất kỳ công trình nào khác.

Tác giả

Phạm Tuấn Phong

LỜI CẢM ƠN

Sau thời gian tập trung nghiên cứu, học viên đã đạt được những kết quả nhất định trong nghiên cứu của mình. Những kết quả đạt được đó không những từ sự cố gắng, nỗ lực của học viên, mà còn có sự hỗ trợ và giúp đỡ của các Thầy hướng dẫn. Học viên xin được bày tỏ lòng biết ơn sâu sắc đến **TS. Nguyễn Duy Phương** đã tận tình hướng dẫn, trang bị phương án nghiên cứu, kiến thức khoa học để học viên hoàn thành các nội dung của đề án.

Học viên trân trọng cảm ơn các Thầy, Cô giáo và Lãnh đạo Học viện Công nghệ Bưu chính Viễn thông đã động viên và tạo điều kiện thuận lợi cho học viên trong thời gian làm đề án.

Cuối cùng, học viên chân thành cảm ơn gia đình đã luôn là hậu phương, hỗ trợ và khích lệ tinh thần cho học viên hoàn thành mục tiêu nghiên cứu của mình.

Hà Nội, tháng 05 năm 2025

MỤC LỤC

DANH MỤC TỪ VIẾT TẮT

Viết tắt	Tiếng Anh	Tiếng Việt
APM	Application Performance Monitoring	Giám sát hiệu năng ứng dụng
AWS	Amazon Web Services	Dịch vụ điện toán đám mây của Amazon
CI/CD	Continuous Integration / Continuous Deployment	Tích hợp liên tục / Triển khai liên tục
CPU	Central Processing Unit	Bộ xử lý trung tâm (vi xử lý)
DNS	Domain Name System	Hệ thống phân giải tên miền
EFS	Elastic File System	Hệ thống tệp linh hoạt (thuộc AWS)
ELK Stack	Elasticsearch, Logstash, Kibana	Bộ công cụ thu thập, xử lý và trực quan hóa log
gRPC	Google Remote Procedure Call	Giao thức gọi thủ tục từ xa hiệu năng cao của Google
HTTP	Hypertext Transfer Protocol	Giao thức truyền tải siêu văn bản
IAM	Identity and Access Management	Quản lý danh tính và truy cập (thuộc AWS)
RAM	Random Access Memory	Bộ nhớ truy cập ngẫu nhiên
RBAC	Role-Based Access Control	Kiểm soát truy cập theo vai trò
REST	Representational State Transfer	Kiến trúc giao tiếp web
S3	Simple Storage Service	Dịch vụ lưu trữ đối tượng (thuộc AWS)
SNS	Simple Notification Service	Dịch vụ gửi thông báo (thuộc AWS)
SQS	Simple Queue Service	Dịch vụ hàng đợi (thuộc AWS)
VPC	Virtual Private Cloud	Ảo hóa đám mây riêng (thuộc AWS)

DANH MỤC HÌNH ẢNH

MỞ ĐẦU

1. Lý do chọn đề tài

Trong kỷ nguyên chuyển đổi số mạnh mẽ hiện nay, kiến trúc microservice đang ngày càng trở thành tiêu chuẩn trong việc phát triển và triển khai các hệ thống phần mềm quy mô lớn. Với khả năng phân tách hệ thống thành các dịch vụ nhỏ, độc lập, microservice mang lại nhiều lợi ích vượt trội về khả năng mở rộng, phát triển linh hoạt, triển khai liên tục và dễ bảo trì [2], [3]. Tuy nhiên, cùng với sự phân tán và tăng trưởng nhanh chóng của hệ thống, bài toán giám sát, phát hiện và xử lý lỗi dịch vụ trở nên phức tạp và cấp thiết hơn bao giờ hết.

Trong một hệ thống microservice, lỗi không còn là vấn đề cục bộ. Một sự cố nhỏ ở một dịch vụ có thể dẫn đến hiệu ứng dây chuyền, làm gián đoạn toàn bộ hệ thống. Thực tế cho thấy, các lỗi như sự cố gián đoạn hạ tầng (container stop), phản hồi phía ứng dụng (Application-level HTTP error), hoặc lỗi vượt ngưỡng thời gian phản hồi do quá tải mạng – nếu không được phát hiện và xử lý kịp thời – có thể gây mất ổn định, giảm chất lượng dịch vụ và ảnh hưởng đến trải nghiệm người dùng cuối [1], [7]. Do đó, việc xây dựng một giải pháp phát hiện lỗi sớm và phản ứng nhanh chóng, thậm chí có khả năng tự khôi phục (self-healing), đang trở thành một nhu cầu thiết yếu trong các doanh nghiệp vận hành theo mô hình microservice [6].

Trên thực tế, có nhiều giải pháp giám sát lỗi đang được triển khai, từ các công cụ mã nguồn mở như Prometheus, ELK Stack, đến các nền tảng thương mại như Datadog, New Relic [5], [12]. Tuy nhiên, các giải pháp này thường yêu cầu cấu hình phức tạp, tiêu tốn tài nguyên, và chi phí cao nếu áp dụng cho hệ thống lớn. Trong khi đó, với sự phát triển mạnh mẽ của các dịch vụ đám mây, đặc biệt là Amazon Web Services (AWS), các doanh nghiệp hiện nay đã có thể tiếp cận những công cụ nhẹ, linh hoạt, chi phí thấp và dễ tích hợp để xây dựng hệ thống giám sát và khôi phục lỗi tự động [9], [10], [11].

Đồ án này được kỳ vọng không chỉ góp phần giải quyết một bài toán kỹ thuật quan trọng trong phát triển hệ thống hiện đại, mà còn mở ra định hướng cho các

nghiên cứu và ứng dụng sâu hơn trong lĩnh vực giám sát thông minh và vận hành hệ thống tự động.

2. Mục tiêu của đề án

Mục tiêu của đề án là xây dựng một giải pháp phát hiện và phục hồi lỗi trong hệ thống microservice, sử dụng các dịch vụ AWS như SNS và SQS để gửi cảnh báo và xử lý lỗi tự động. Hệ thống cần phát hiện được các lỗi phổ biến như dịch vụ ngừng đột ngột, lỗi HTTP và timeout. Đồng thời, đề án đánh giá hiệu quả giải pháp và so sánh với các phương pháp giám sát truyền thống. Giải pháp hướng tới tính đơn giản, dễ triển khai và phù hợp với môi trường doanh nghiệp thực tế.

3. Phạm vi, đối tượng và phương pháp nghiên cứu

Phạm vi nghiên cứu tập trung vào việc phát hiện và phục hồi ba loại lỗi thường gặp trong hệ thống microservice: dịch vụ bị dừng đột ngột, lỗi phản hồi HTTP và lỗi timeout.

Đối tượng nghiên cứu là kiến trúc microservice triển khai bằng Docker, các dịch vụ AWS như SNS, SQS và các cơ chế phản ứng lỗi tự động thông qua Lambda hoặc script nội bộ.

Phương pháp nghiên cứu bao gồm khảo sát tài liệu, xây dựng mô hình, mô phỏng lỗi thực tế trong môi trường cục bộ (LocalStack), triển khai thử nghiệm, và so sánh với các phương pháp giám sát khác để đánh giá hiệu quả hệ thống.

4. Bố cục đề án

Nội dung đề án được trình bày trong ba chương, bao gồm:

Chương 1: Giới thiệu và cơ sở lý luận, Trình bày bối cảnh nghiên cứu, lý do chọn đề tài, mục tiêu, phạm vi và phương pháp thực hiện. Đồng thời tổng hợp cơ sở lý thuyết về kiến trúc microservice, AWS, các loại lỗi phổ biến trong hệ thống phân tán.

Chương 2: Xây dựng hệ thống phát hiện và khắc phục lỗi trên AWS, Phân tích, thiết kế yêu cầu, mô tả kiến trúc hệ thống đề xuất, quy trình phát hiện và xử lý lỗi sử dụng SNS, SQS và script tự động.

Chương 3: Mô phỏng, kết luận và đánh giá, Trình bày quá trình mô phỏng lỗi, triển khai thử nghiệm và mô phỏng. Tổng kết toàn bộ kết quả đạt được, phân tích ưu điểm – hạn chế của hệ thống, so sánh với các phương pháp khác, ứng dụng thực tiễn, định hướng, khả năng tích hợp và đưa ra các định hướng phát triển tiếp theo.

Chương 1 - GIỚI THIỆU VÀ CƠ SỞ LÝ LUẬN

Trong Chương 1, đề án trình bày bối cảnh, động cơ, mục tiêu và phạm vi nghiên cứu, lý do chọn đề tài, phương pháp thực hiện. Đồng thời tổng hợp cơ sở lý thuyết về kiến trúc microservice, AWS, các loại lỗi phổ biến trong hệ thống phân tán.

1.1 Bối cảnh và động cơ nghiên cứu

Microservices là một kiến trúc phần mềm trong đó hệ thống được chia thành nhiều dịch vụ nhỏ, độc lập, mỗi dịch vụ đảm nhiệm một chức năng riêng biệt và có thể triển khai một cách riêng lẻ. Các dịch vụ này thường giao tiếp với nhau thông qua các giao thức mạng như HTTP/REST hoặc gRPC [6], [7].. Kiến trúc này giúp tăng tính linh hoạt, dễ bảo trì và mở rộng hệ thống.

Trong những năm gần đây, kiến trúc microservices đã trở thành một xu hướng chủ đạo trong phát triển hệ thống phần mềm quy mô lớn. Thay vì xây dựng một hệ thống nguyên khối (monolithic), microservices chia nhỏ hệ thống thành nhiều dịch vụ nhỏ, độc lập, mỗi dịch vụ đảm nhiệm một chức năng cụ thể. Mỗi service có thể được phát triển, triển khai và mở rộng riêng biệt, giúp tăng tính linh hoạt, khả năng mở rộng và khả năng phục hồi [6], [8].

Trong mô hình truyền thống, toàn bộ ứng dụng được triển khai như một khối duy nhất. Khi có lỗi xảy ra ở một module, toàn bộ hệ thống có thể bị ảnh hưởng. Ngoài ra, việc triển khai lại cả hệ thống chỉ để cập nhật một chức năng nhỏ cũng tiêu tốn thời gian và tài nguyên đáng kể. Trái lại, microservices cho phép triển khai từng phần của ứng dụng mà không ảnh hưởng đến phần còn lại. Tính linh hoạt này đã khiến nhiều doanh nghiệp chuyển đổi sang mô hình microservices nhằm tối ưu hóa quá trình phát triển phần mềm [7].

Tuy nhiên, cùng với những lợi ích đó, microservices cũng đặt ra những thách thức mới, đặc biệt trong vấn đề quản lý, giám sát và xử lý lỗi. Do các service hoạt động độc lập và giao tiếp qua mạng (thường là HTTP hoặc gRPC), nên nếu một service ngừng hoạt động hoặc phản hồi chậm, điều này có thể gây ảnh hưởng dây chuyền tới các thành phần khác trong hệ thống. Do đó, việc giám sát trạng thái

service, phát hiện lỗi sớm và có khả năng tự động khắc phục lỗi là những yêu cầu sống còn để đảm bảo hệ thống vận hành liên tục [8].

Một thách thức khác là khả năng quan sát hệ thống (observability). Với hàng chục, thậm chí hàng trăm service, việc xác định nguyên nhân gốc rễ của một lỗi trở nên phức tạp hơn nhiều. Các công cụ như centralized logging (ELK stack), distributed tracing (Jaeger, Zipkin), và metrics collection (Prometheus, CloudWatch) trở thành bắt buộc trong hệ thống microservices [4], [9].

Trong các hệ thống microservices, lỗi có thể xảy ra dưới nhiều hình thức: một container bị crash, một cổng dịch vụ đóng sai, giới hạn tài nguyên (CPU, memory) bị vượt quá, kết nối giữa các service bị timeout, hoặc các vấn đề về cấu hình bảo mật (IAM, Security Group) [10]. Nếu không được xử lý kịp thời, những lỗi này không chỉ ảnh hưởng đến một service mà có thể gây ra hiệu ứng domino, làm sập toàn bộ hệ thống.

Do đó, nhu cầu phát triển các cơ chế phát hiện lỗi sớm và phản ứng tự động trở nên cấp thiết. Các hệ thống hiện đại không chỉ cần phát hiện ra lỗi nhanh chóng, mà còn phải có khả năng phản hồi ngay lập tức như gửi cảnh báo, tự động khởi động lại container, hoặc tái triển khai service bị lỗi. Vấn đề đặt ra là làm thế nào để xây dựng một cơ chế giám sát chủ động, có khả năng phát hiện lỗi nhanh chóng và phản ứng tự động để khắc phục sự cố? Đây chính là động lực thúc đẩy việc nghiên cứu và triển khai các giải pháp phát hiện và xử lý lỗi trong môi trường microservices [6], [7].

Amazon Web Services (AWS) là nền tảng điện toán đám mây hàng đầu hiện nay, cung cấp nhiều dịch vụ như lưu trữ, tính toán, mạng, cơ sở dữ liệu và các dịch vụ hỗ trợ DevOps [2], [3]. Trong đó, AWS cung cấp các công cụ mạnh mẽ như CloudWatch, SNS và Lambda phục vụ cho việc giám sát và tự động khắc phục lỗi trong hệ thống phân tán [4], [9], [10]. Trong số các nền tảng hỗ trợ triển khai và vận hành hệ thống hiện đại, Amazon Web Services (AWS) được đánh giá là một trong những giải pháp hàng đầu nhờ khả năng cung cấp đầy đủ các công cụ phục vụ giám sát, phát hiện lỗi và tự động phản ứng. Với sự phát triển mạnh mẽ của các hệ thống phân tán, đặc biệt là kiến trúc microservices, việc tích hợp các giải pháp giám sát theo

thời gian thực và tự động hóa xử lý lỗi là yêu cầu thiết yếu. AWS đáp ứng tốt những yêu cầu này thông qua một chuỗi dịch vụ tích hợp chặt chẽ.

Cụ thể, Amazon CloudWatch cung cấp khả năng thu thập log, theo dõi chỉ số hiệu năng và thiết lập các ngưỡng cảnh báo đối với hệ thống. Khi một sự kiện bất thường được phát hiện như response time vượt ngưỡng, lỗi HTTP 5xx hay timeout, CloudWatch có thể kích hoạt hành động tiếp theo bằng cách gửi thông báo qua AWS SNS (Simple Notification Service) [2], [4]. SNS hoạt động theo mô hình publish-subscribe, cho phép truyền tải cảnh báo đến nhiều thành phần khác nhau trong hệ thống. Một trong những thành phần tiêu biểu là AWS Lambda – dịch vụ điện toán serverless có thể thực thi mã lệnh tự động nhằm phản ứng với lỗi, ví dụ như khởi động lại container, ghi log bổ sung, gửi thông báo tới quản trị viên hoặc thậm chí tương tác với các API nội bộ [10].

Với kiến trúc này, AWS không chỉ cung cấp các dịch vụ rời rạc, mà cho phép kết nối các thành phần lại thành một pipeline hoàn chỉnh từ khâu phát hiện – cảnh báo – phản ứng – khắc phục lỗi [3], [4], [9]. Các dịch vụ được vận hành trên hạ tầng đám mây ổn định, có khả năng mở rộng cao và bảo mật tốt, phù hợp cả với môi trường doanh nghiệp lớn và các nhóm phát triển nhỏ.

Nhờ tính tự động hóa, tích hợp dễ dàng và độ tin cậy cao, AWS ngày càng được nhiều tổ chức sử dụng làm nền tảng cho các hệ thống yêu cầu độ sẵn sàng cao và cần giảm thiểu thời gian downtime [10]. Khả năng phản ứng gần như tức thời với lỗi giúp giảm thiểu rủi ro, tăng tính ổn định và hỗ trợ hiệu quả cho các mô hình CI/CD hiện đại.

Xuất phát từ những vấn đề và nhu cầu thực tiễn nêu trên, đồ án này được thực hiện với mục tiêu xây dựng một mô hình phát hiện và khắc phục lỗi trong hệ thống microservices sử dụng các dịch vụ của AWS. Thông qua mô hình, đồ án hướng đến việc thiết kế một pipeline giám sát hoàn chỉnh, phát hiện lỗi theo thời gian thực, gửi cảnh báo đúng ngữ cảnh và thực hiện các hành động phản ứng tự động. Kết quả đạt

được sẽ góp phần nâng cao tính ổn định, khả năng tự phục hồi và độ tin cậy của hệ thống microservices trong thực tế triển khai.

1.2 Mục tiêu và phạm vi nghiên cứu

1.2.1 Mục tiêu

Trong bối cảnh chuyển đổi số mạnh mẽ và xu hướng phát triển phần mềm hiện đại, kiến trúc microservices và nền tảng điện toán đám mây AWS ngày càng đóng vai trò quan trọng trong việc thiết kế, triển khai và vận hành các hệ thống quy mô lớn. Tuy nhiên, để khai thác hiệu quả các công nghệ này, đòi hỏi người phát triển không chỉ nắm vững kiến thức lý thuyết mà còn cần hiểu rõ các vấn đề phát sinh thực tế cũng như giải pháp khắc phục phù hợp.

Xuất phát từ thực tiễn đó, đồ án này được xây dựng với mục tiêu chính là giúp người đọc hiểu sâu và đúng về ba khía cạnh trọng tâm trong thiết kế hệ thống hiện đại:

a. Giúp người đọc hiểu rõ về kiến trúc microservices và nền tảng AWS

Microservices không còn là một khái niệm mới trong lĩnh vực phần mềm, nhưng việc triển khai đúng và vận hành hiệu quả mô hình này vẫn là một thách thức lớn với nhiều cá nhân và tổ chức. Do đó, mục tiêu đầu tiên của đồ án là trình bày một cách rõ ràng, dễ tiếp cận về kiến trúc microservices: từ định nghĩa, đặc điểm, ưu điểm, đến những điểm khác biệt căn bản so với kiến trúc nguyên khối (monolithic).

Đồng thời, đồ án cũng cung cấp cái nhìn tổng quan về nền tảng AWS – một trong những hệ sinh thái điện toán đám mây hàng đầu thế giới. Việc hiểu rõ đặc điểm, vai trò và ứng dụng thực tiễn của AWS sẽ giúp người đọc hình dung được bức tranh tổng thể trong thiết kế, triển khai và vận hành các hệ thống hiện đại.

b. Giúp người đọc nhận diện rõ các lỗi phổ biến trong hệ thống microservices

Một hệ thống microservices, mặc dù có nhiều ưu điểm, lại tiềm ẩn hàng loạt vấn đề kỹ thuật phát sinh trong quá trình giao tiếp giữa các dịch vụ độc lập. Chính vì vậy, mục tiêu thứ hai của đồ án là hệ thống hóa và phân loại rõ các lỗi thường gặp

trong môi trường microservices, từ lỗi dừng dịch vụ, lỗi mạng, lỗi logic đến lỗi dữ liệu đầu ra sai định dạng.

Việc phân tích kỹ các lỗi không chỉ giúp người đọc có cái nhìn toàn diện và sâu sắc về đặc thù hệ thống microservices mà còn là tiền đề quan trọng để tiếp cận các phương pháp giám sát và phản ứng hiệu quả [1][3][6].

c. Trình bày phương pháp phát hiện và khắc phục lỗi trong microservices dựa trên AWS

Mục tiêu cuối cùng của đồ án là giới thiệu và phân tích cách thức sử dụng nền tảng AWS để phát hiện lỗi và thực hiện các hành động phản ứng trong hệ thống microservices. Thông qua đó, người đọc có thể nắm được quy trình giám sát, gửi cảnh báo và khôi phục dịch vụ một cách tự động – giúp đảm bảo tính ổn định và độ tin cậy cao cho hệ thống.

Đồ án sẽ mô tả tổng quan nguyên lý vận hành, luồng xử lý và mối quan hệ giữa các thành phần trong pipeline phản ứng lỗi, để từ đó hình thành một mô hình có thể áp dụng vào thực tế vận hành.

d. So sánh và chỉ ra sự khác biệt giữa phương pháp sử dụng AWS với các giải pháp phát hiện – khắc phục lỗi khác

Cuối cùng, đồ án hướng tới việc làm rõ điểm khác biệt giữa phương pháp sử dụng nền tảng AWS với các phương pháp truyền thống trong việc giám sát và xử lý lỗi hệ thống. Các phương pháp cũ thường đòi hỏi sự can thiệp thủ công, thiếu tính tự động hóa hoặc khó mở rộng, trong khi AWS cung cấp giải pháp linh hoạt, chủ động và dễ tích hợp với quy trình DevOps hiện đại.

Thông qua các tiêu chí so sánh như khả năng tự động hóa, thời gian phản hồi, tính mở rộng và mức độ dễ triển khai, đồ án giúp người đọc nhận ra đâu là điểm mạnh nổi bật của mô hình đề xuất, từ đó lựa chọn phù hợp hơn cho từng môi trường hệ thống cụ thể [12].

1.2.2 Phạm vi nghiên cứu

Trong khuôn khổ đồ án này, phạm vi nghiên cứu được xác định rõ ràng nhằm đảm bảo tính khả thi trong triển khai, kiểm thử và đánh giá. Thay vì xây dựng một hệ

thống phức tạp quy mô doanh nghiệp, đề án tập trung vào việc mô phỏng một mô hình đơn giản nhưng đầy đủ chức năng, có thể thể hiện rõ quy trình phát hiện – cảnh báo – khắc phục lỗi trong hệ thống microservices dựa trên các dịch vụ của AWS.

Cụ thể, phạm vi nghiên cứu bao gồm các nội dung sau:

a. Xây dựng mô hình hệ thống microservices cơ bản gồm hai dịch vụ chính:
   - Service A đóng vai trò là dịch vụ mục tiêu, có thể chủ động tạo lỗi (như timeout, crash, sai định dạng phản hồi…).
   - Service B là dịch vụ giám sát, định kỳ kiểm tra hoạt động của Service A và phát hiện lỗi khi có dấu hiệu bất thường.

b. Sử dụng các dịch vụ cốt lõi trong AWS để triển khai pipeline phản ứng lỗi, bao gồm:
   - Giám sát trạng thái và log dịch vụ thông qua cơ chế kiểm tra định kỳ và ghi log.
   - Gửi cảnh báo lỗi dưới dạng thông điệp JSON mô phỏng gửi lên SNS.
   - Kích hoạt Lambda function tương ứng để xử lý thông báo lỗi, thực hiện hành động phản ứng như: log, cảnh báo phụ, khởi động lại container...

c. Các tình huống lỗi được mô phỏng trong phạm vi đồ án bao gồm:
   - Lỗi hạ tầng: sự cố gián đoạn hạ tầng (container stop)
   - Lỗi logic ứng dụng: lỗi phản hồi phía ứng dụng (Application-level HTTP error)
   - Lỗi hiệu năng: lỗi vượt ngưỡng thời gian phản hồi

d. Môi trường

Môi trường triển khai được sử dụng là máy tính cá nhân, kết hợp với công cụ LocalStack – nền tảng giả lập AWS – để mô phỏng các dịch vụ CloudWatch, SNS và Lambda. Điều này cho phép kiểm thử toàn bộ pipeline phát hiện và phản ứng lỗi mà không cần tài khoản AWS thật, từ đó tiết kiệm chi phí và đảm bảo khả năng kiểm soát môi trường thử nghiệm.

e. Đồ án không đi sâu vào các vấn đề sau:
   - Không xử lý lỗi bảo mật, phân quyền phức tạp (IAM, RBAC nâng cao).

- Không triển khai giám sát quy mô lớn hoặc liên vùng (multi-region).
- Không tích hợp giao diện trực quan (dashboard) hoặc công cụ APM ngoài AWS như Datadog, Prometheus.
- Không thực hiện mô hình học máy (machine learning) cho phát hiện lỗi bất thường.

Do giới hạn về cơ sở vật chất, hạ tầng phần cứng cũng như quy mô triển khai, việc mô phỏng trong khuôn khổ đồ án không thể tái hiện đầy đủ môi trường vận hành trên nền tảng AWS thật với tất cả các tính năng bảo mật, tự động hóa và tích hợp sâu rộng. Thay vào đó, các thành phần như CloudWatch, SNS và Lambda sẽ được triển khai bằng công cụ giả lập (LocalStack) để mô phỏng hành vi gần giống thực tế [9]. Mặc dù còn nhiều giới hạn về mặt kỹ thuật, đồ án vẫn tập trung làm rõ luồng xử lý và nguyên lý hoạt động của pipeline giám sát – cảnh báo – phản ứng lỗi tự động. Mô hình được xây dựng nhằm giúp người đọc hiểu rõ cách các thành phần phối hợp với nhau trong môi trường microservices hiện đại, đồng thời có thể mở rộng hoặc tích hợp vào hệ thống thực tế nếu có điều kiện triển khai trên nền tảng AWS thật [1][6][9].

1.3 Các lỗi trong hệ thống microservices, phương pháp phát hiện và xử lý của AWS

1.3.1 Các lỗi thường gặp trong microservice

Trong kiến trúc microservices, việc phát sinh lỗi là điều không thể tránh khỏi do đặc thù phân tán, giao tiếp mạng và tính độc lập của từng service. Các lỗi có thể xảy ra ở nhiều lớp khác nhau trong hệ thống như lỗi ứng dụng, lỗi mạng, lỗi cấu hình bảo mật, lỗi khi triển khai hoặc quá tải tài nguyên. Việc phát hiện sớm và xử lý kịp thời các lỗi này là yếu tố then chốt giúp đảm bảo tính ổn định và độ tin cậy của toàn hệ thống [1][5][6].

Một số loại lỗi phổ biến trong hệ thống microservices bao gồm:

STT	Loại lỗi	Mô tả	Nguyên nhân phổ biến
1	Timeout	Dịch vụ không phản hồi trong thời gian cho phép	Tải nặng, deadlock, dịch vụ phụ không phản hồi
2	Lỗi HTTP 500 / 503	Lỗi nội bộ hoặc dịch vụ không sẵn sàng	Bug logic, lỗi truy vấn DB, hết tài nguyên
3	Dịch vụ bị dừng (crash)	Dịch vụ ngừng chạy đột ngột hoặc container bị stop	Lỗi bộ nhớ, CPU quá tải, deploy lỗi
4	Lỗi DNS	Không phân giải được địa chỉ dịch vụ	Sai cấu hình DNS, tên host sai, container chưa khởi động
5	Phản hồi sai định dạng	Dữ liệu trả về không đúng định dạng mong đợi (ví dụ: không JSON)	Lỗi logic, thiếu field, lỗi serialization
6	Kết nối bị từ chối	Không thể thiết lập kết nối TCP đến dịch vụ	Port sai, firewall, dịch vụ chưa chạy
7	Circuit breaker bật	Hệ thống chủ động ngắt kết nối do nhiều lỗi liên tiếp	Ngưỡng lỗi vượt giới hạn, thiết lập circuit breaker
8	Gọi sai endpoint	Gọi nhầm đường dẫn API hoặc endpoint không tồn tại	Thay đổi API, client không cập nhật

9	Phản hồi chậm	Dữ liệu hợp lệ nhưng trả về quá trễ	Tắc nghẽn DB, request chồng chéo, xử lý lâu
10	Lỗi tái phát liên tục	Lỗi vừa khắc phục xong lại xuất hiện lại ngay	Cấu hình sai, lỗi gốc chưa xử lý triệt để

Khi các lỗi này xảy ra, nếu không có cơ chế giám sát và phục hồi tự động, hệ thống sẽ dễ bị gián đoạn kéo dài. Đặc biệt, trong môi trường có nhiều service phụ thuộc lẫn nhau, lỗi từ một service có thể lan rộng theo chuỗi, ảnh hưởng toàn bộ hệ thống.

1.3.2 Nguyên lý phát hiện và khắc phục lỗi của AWS

Amazon Web Services (AWS) cung cấp một bộ công cụ mạnh mẽ giúp tự động phát hiện và khắc phục lỗi trong các hệ thống phân tán sử dụng kiến trúc microservices. Cơ chế này được xây dựng dựa trên nguyên tắc giám sát chủ động, cảnh báo sự kiện và phản ứng tự động, nhằm đảm bảo hệ thống hoạt động liên tục và ổn định [3][4][5].

a. Phát hiện lỗi

AWS sử dụng các thành phần sau để phát hiện lỗi:

- Amazon CloudWatch: Theo dõi log, chỉ số hiệu suất (CPU, memory, response time...) và có thể thiết lập ngưỡng cảnh báo (alarm).
- CloudWatch Logs và CloudWatch Metrics: Giúp phát hiện lỗi ứng dụng (HTTP 500, 503...), lỗi hệ thống (timeout, crash), hoặc hành vi bất thường (spike load, chậm phản hồi).
- Amazon EventBridge hoặc AWS Config: Giúp phát hiện các thay đổi bất thường về cấu hình, trạng thái dịch vụ, hoặc chính sách bảo mật.

b. Gửi cảnh báo

Khi một lỗi được phát hiện, AWS có thể tự động gửi cảnh báo thông qua:

AWS SNS (Simple Notification Service): Gửi thông báo đến email, tin nhắn, dịch vụ xử lý hoặc hệ thống quản lý vận hành.

Các dịch vụ cảnh báo có thể kích hoạt hành động tiếp theo theo mô hình sự kiện (event-driven).

c. Khắc phục lỗi tự động

AWS cung cấp khả năng tự khắc phục lỗi bằng:

AWS Lambda: Một hàm serverless có thể được gọi tự động khi có sự kiện lỗi xảy ra. Lambda có thể:

- Khởi động lại dịch vụ hoặc container lỗi
- Thay đổi cấu hình
- Gửi cảnh báo nâng cao (email, webhook...)
- Ghi log hoặc cập nhật hệ thống giám sát

AWS Auto Scaling: Tự động scale-in/scale-out khi tài nguyên vượt ngưỡng hoặc khi một instance gặp sự cố.

Elastic Load Balancing (ELB): Phân phối lại lưu lượng khi một node không phản hồi.

d. Khả năng mở rộng và tích hợp

- Cơ chế phát hiện và khắc phục lỗi của AWS có thể mở rộng:
- Kết hợp với AWS Systems Manager để tự động chạy playbook xử lý sự cố.
- Tích hợp với DevOps toolchain như Jenkins, GitLab CI, hoặc hệ thống ticket như Jira.

1.4 Định hướng sử dụng dịch vụ AWS trong đồ án

Để thực hiện mô hình phát hiện và khắc phục lỗi trong hệ thống microservices, đồ án sử dụng một số dịch vụ quan trọng trong hệ sinh thái Amazon Web Services (AWS). Các dịch vụ này đóng vai trò chủ chốt trong việc giám sát hệ thống, phát hiện sự cố và tự động phản hồi nhằm đảm bảo tính liên tục và ổn định của hệ thống [3][4][5]. Tuy việc triển khai không thực hiện trực tiếp trên AWS thật, nhưng thông qua công cụ LocalStack, các hành vi, API và quy trình mô phỏng vẫn đảm bảo mức độ tương đồng cao.

a. AWS CloudWatch:

CloudWatch là dịch vụ giám sát và quan sát hệ thống do AWS cung cấp. Nó thu thập log, metric và sự kiện từ các dịch vụ AWS cũng như ứng dụng người dùng [4]. Trong mô hình đồ án, CloudWatch được sử dụng để theo dõi log phản hồi từ các dịch vụ, thiết lập các ngưỡng cảnh báo như CPU, memory hoặc lỗi hệ thống. Thông qua log và biểu đồ, người vận hành có thể phát hiện các service gặp sự cố hoặc phản hồi bất thường.

b. AWS SNS (Simple Notification Service):

SNS là dịch vụ gửi thông báo theo mô hình publish-subscribe [5]. Khi Service B phát hiện lỗi từ Service A, nó sẽ publish một thông điệp lỗi lên một topic SNS. Các thành phần đăng ký nhận thông báo từ SNS có thể là Lambda, email, HTTP endpoint hoặc các dịch vụ khác. SNS đảm bảo việc truyền tải sự kiện cảnh báo đến các thành phần phản ứng phù hợp.

c. AWS Lambda:

Lambda là dịch vụ serverless cho phép thực thi các đoạn mã nhỏ mà không cần quản lý máy chủ [5]. Trong hệ thống mô phỏng, Lambda đảm nhiệm vai trò phản ứng với lỗi bằng cách thực hiện các hành động như: ghi log chi tiết, gửi email cảnh báo, hoặc gọi API để khởi động lại container bị lỗi. Do Lambda có thể chạy gần như tức thì, nó giúp hệ thống phản ứng nhanh chóng với các sự cố phát sinh.

Việc phối hợp nhịp nhàng giữa CloudWatch → SNS → Lambda giúp xây dựng pipeline xử lý lỗi hoàn chỉnh, minh họa rõ năng lực phát hiện – phản ứng tự động mà AWS hỗ trợ [3].

Việc tích hợp các dịch vụ AWS không chỉ giúp hệ thống phản ứng linh hoạt trước sự cố mà còn minh họa rõ khả năng xây dựng pipeline vận hành tự động. Bằng cách thiết lập quy trình từ phát hiện đến hành động khắc phục, hệ thống có thể hoạt động ổn định ngay cả khi có lỗi xảy ra bất ngờ.

Ngoài ba dịch vụ chính là CloudWatch, SNS và Lambda, đồ án cũng có thể mở rộng để sử dụng thêm một số dịch vụ bổ trợ như:

d. Amazon S3 (Simple Storage Service):

Dùng để lưu trữ log, báo cáo, hoặc output từ Lambda phục vụ cho mục đích kiểm tra hoặc phân tích lỗi sau khi xảy ra sự cố [3].

e. Amazon EventBridge:

Thay thế hoặc bổ sung cho SNS trong việc truyền sự kiện theo thời gian thực giữa các service [5]. EventBridge có khả năng lọc event và định tuyến đến nhiều dịch vụ khác nhau.

f. Amazon ECS (Elastic Container Service):

Nếu hệ thống được triển khai thực tế trên AWS, ECS sẽ là nền tảng giúp quản lý container (thay cho Docker Compose) [3]. ECS tích hợp chặt chẽ với CloudWatch và Lambda giúp tự động hóa quản lý container theo trạng thái.

Trong phạm vi đồ án, việc sử dụng các dịch vụ này sẽ được mô phỏng bằng LocalStack. Hành vi của CloudWatch, SNS và Lambda được kiểm chứng qua log và phản hồi của hệ thống trong quá trình xử lý lỗi.

Việc lựa chọn bộ ba CloudWatch – SNS – Lambda không chỉ vì mức độ phổ biến trong thực tế, mà còn vì tính dễ hiểu, triển khai nhanh và phù hợp với mô hình nhỏ gọn phục vụ mục đích học tập. Chúng giúp thể hiện rõ khả năng tự động hóa trong quy trình giám sát và phản ứng lỗi, phù hợp với mục tiêu của đề tài là mô phỏng hệ thống tự phục hồi lỗi theo hướng hiện đại.

Trong chương tiếp theo, các dịch vụ này sẽ được tích hợp vào mô hình hệ thống cụ thể, thể hiện thông qua kiến trúc triển khai và kịch bản thử nghiệm lỗi trong môi trường giả lập.

1.5 So sánh tổng quan Local Stack và AWS thực tế

Trong môi trường máy cục bộ. Công cụ này đặc biệt hữu ích trong quá trình phát triển và kiểm thử ứng dụng khi người dùng không muốn hoặc không thể truy cập trực tiếp vào tài khoản AWS thật. Việc sử dụng LocalStack giúp tiết kiệm chi phí, tăng khả năng kiểm soát môi trường thử nghiệm và dễ dàng tái hiện các tình huống lỗi trong hệ thống phân tán [6].

Tuy nhiên, giữa LocalStack và AWS thật vẫn tồn tại nhiều điểm khác biệt cần được hiểu rõ để tránh những sai lệch trong đánh giá hoặc triển khai thực tế. Dưới đây là so sánh giữa hai nền tảng:

- Chi phí: LocalStack hoàn toàn miễn phí (bản cộng đồng), trong khi AWS tính phí theo tài nguyên sử dụng [6].
- Mục tiêu sử dụng: LocalStack phù hợp cho phát triển và kiểm thử, còn AWS dùng cho triển khai thật và vận hành ở quy mô lớn [6].
- Tính năng: AWS hỗ trợ đầy đủ mọi dịch vụ, còn LocalStack chỉ mô phỏng một phần (khoảng 70–80%) các dịch vụ phổ biến như Lambda, S3, SNS, CloudWatch [6].
- Tốc độ và hiệu suất: LocalStack chạy trong Docker, phụ thuộc cấu hình máy cục bộ; AWS cung cấp hạ tầng mạnh mẽ và có khả năng tự động mở rộng. - Độ tin cậy và bảo mật: LocalStack không cung cấp khả năng bảo mật ở cấp độ sản phẩm như IAM, VPC, Shield... trong khi AWS được tối ưu để phục vụ các ứng dụng thực tế với các chứng chỉ bảo mật cao cấp [3][4].
- Tích hợp DevOps: AWS tích hợp sâu với các công cụ CI/CD như CodePipeline, CodeBuild, CodeDeploy... LocalStack cần cấu hình thủ công nếu muốn tích hợp tương tự [5][6].

Trong phạm vi đề án này, việc sử dụng LocalStack là hoàn toàn phù hợp vì mục tiêu là thử nghiệm mô hình, kiểm tra pipeline phản ứng lỗi và đào tạo thực hành môi trường gần giống AWS thật. Dù không thể thay thế hoàn toàn AWS thật trong môi trường production, LocalStack vẫn là lựa chọn tuyệt vời trong các mô hình mô phỏng và giáo dục [6].

Dù LocalStack không thể thay thế hoàn toàn AWS thật trong môi trường sản xuất, nhưng nó đóng vai trò rất quan trọng trong quá trình phát triển phần mềm. Khả năng giả lập các dịch vụ phổ biến của AWS giúp các nhóm phát triển có thể tạo ra pipeline CI/CD cục bộ, kiểm thử tích hợp sớm và phát hiện lỗi trước khi triển khai thật[6].

Một lợi ích quan trọng khác của LocalStack là khả năng hoạt động offline.

Người dùng không cần kết nối internet vẫn có thể mô phỏng hệ sinh thái AWS trên máy cá nhân. Điều này phù hợp với môi trường học tập, nghiên cứu hoặc khi cần thử nghiệm trên môi trường không có mạng.

Tuy nhiên, khi hệ thống cần triển khai ở quy mô lớn, yêu cầu bảo mật cao, tích hợp sâu với các dịch vụ AWS như VPC, IAM, hoặc phân vùng đa khu vực (multi-region), thì LocalStack sẽ không đáp ứng được. Bên cạnh đó, một số dịch vụ cao cấp như AWS Shield, CloudFront, EFS, hoặc dịch vụ machine learning không được hỗ trợ đầy đủ hoặc chỉ có trong phiên bản LocalStack Pro (trả phí) [6].

Một số giới hạn khác của LocalStack:

- Không đảm bảo hiệu năng ổn định như AWS thật (do chạy trong Docker local).
- Không có dashboard trực quan, cần sử dụng AWS CLI hoặc SDK để tương tác.
- Một số behavior hoặc response API có thể chưa hoàn toàn chính xác với AWS thật.

Tuy nhiên, đối với mục tiêu mô phỏng pipeline phát hiện – xử lý lỗi đơn giản, thì LocalStack là công cụ rất phù hợp. Nó giúp học viên thực hiện mô hình hóa toàn bộ quy trình: từ service bị lỗi, phát hiện bằng log, gửi SNS, kích hoạt Lambda... ngay trên môi trường cục bộ.

Việc tận dụng công cụ giả lập như LocalStack giúp chúng ta dễ tiếp cận hơn với kiến trúc hiện đại, đồng thời tiết kiệm chi phí và tăng khả năng lặp lại thử nghiệm. Đây là nền tảng quan trọng để bước sang phần triển khai cụ thể trong chương 2.

Chương tiếp theo sẽ trình bày chi tiết hệ thống được triển khai, bao gồm kiến trúc tổng thể, mô phỏng lỗi và phản ứng hệ thống thông qua các dịch vụ giả lập này.

1.6 Kết luận chương

Chương 1 đã trình bày tổng quan lý thuyết và cơ sở khoa học làm nền tảng cho việc xây dựng mô hình phát hiện và khắc phục lỗi trong hệ thống microservices sử dụng nền tảng AWS. Từ bối cảnh thực tiễn đến kiến trúc hệ thống, các loại lỗi thường gặp, giải pháp của AWS và môi trường giả lập LocalStack, toàn bộ chương đã cung

cấp đầy đủ thông tin giúp người đọc hiểu rõ lý do và phương pháp tiếp cận mà đồ án lựa chọn.

Đầu tiên, đồ án đã chỉ ra xu hướng phát triển tất yếu từ hệ thống monolithic sang microservices, cùng với những lợi ích và thách thức đi kèm. Đặc biệt là trong việc vận hành, giám sát và khắc phục lỗi, microservices yêu cầu một hệ thống quan sát thông minh và khả năng tự động phản ứng kịp thời để đảm bảo độ tin cậy và tính liên tục của dịch vụ.

Tiếp theo, các loại lỗi điển hình trong môi trường microservices đã được phân loại rõ ràng như lỗi container, lỗi mạng, quá tải tài nguyên, lỗi bảo mật và lỗi triển khai. AWS là một trong những nền tảng mạnh mẽ nhất hiện nay cung cấp đầy đủ công cụ để phát hiện và xử lý những lỗi này thông qua CloudWatch, SNS và Lambda. Việc sử dụng LocalStack giúp mô phỏng các dịch vụ này một cách hiệu quả, tạo điều kiện thuận lợi để sinh viên thực hành và triển khai giải pháp trong môi trường học thuật.

Chương này cũng đã đề cập đến các mục tiêu nghiên cứu, phạm vi đồ án, cũng như so sánh thực tế giữa LocalStack và AWS thật, nhằm đảm bảo sự nhất quán trong mô hình mô phỏng và khả năng áp dụng trong môi trường thực tế.

Tóm lại, chương 1 đã thiết lập cơ sở lý luận cần thiết cho đồ án. Những nội dung được trình bày sẽ là nền móng vững chắc để triển khai chương tiếp theo, nơi hệ thống được hiện thực hóa, mô phỏng các lỗi và xây dựng pipeline phát hiện – khắc phục sự cố một cách tự động và hiệu quả.

Chương 2 - XÂY DỰNG HỆ THỐNG PHÁT HIỆN VÀ KHẮC PHỤC LỖI TRÊN AWS

Mô tả kiến trúc hệ thống đề xuất, quy trình phát hiện và xử lý lỗi sử dụng SNS, SQS và script tự động. Trình bày quá trình mô phỏng lỗi, triển khai thử nghiệm và mô phỏng.

2.1 Phân tích, thiết kế hệ thống phát hiện và khắc phục lỗi trên AWS

Dựa trên cơ sở lý luận và mục tiêu đã trình bày ở chương trước, chương 2 sẽ tập trung vào việc thiết kế hệ thống mô phỏng có khả năng phát hiện lỗi trong hệ thống microservices và thực hiện phản ứng tự động để khắc phục sự cố bằng cách sử dụng các dịch vụ AWS được mô phỏng thông qua LocalStack.

Yêu cầu chức năng:

Hệ thống phát hiện và phục hồi lỗi trong môi trường microservice cần đảm bảo các chức năng chính:

- Giám sát trạng thái dịch vụ theo thời gian thực.
- Phát hiện ba loại lỗi phổ biến:
    - Dịch vụ dừng đột ngột.
    - HTTP lỗi (500, 503).
    - Phản hồi chậm (Timeout).
- Gửi cảnh báo đến người quản trị thông qua cơ chế publish-subscribe (SNS, SQS).
- Tự động thực hiện hành động khôi phục (như khởi động lại container).

Yêu cầu phi chức năng:

- Tính sẵn sàng cao: hệ thống giám sát không bị gián đoạn.
- Khả năng mở rộng: dễ dàng bổ sung thêm loại lỗi mới hoặc dịch vụ mới.
- Tính mô-đun: các thành phần hoạt động độc lập, giao tiếp qua API.
- Khả năng mô phỏng môi trường thực tế bằng công cụ như LocalStack.

Để mô phỏng quá trình phát hiện và khắc phục lỗi trong môi trường microservices sử dụng AWS, đồ án xây dựng một hệ thống đơn giản gồm hai dịch vụ chính cùng với các thành phần hỗ trợ để tạo thành một pipeline giám sát – cảnh báo – phản ứng lỗi.

Mô hình tổng quát gồm các thành phần sau:

- Service A: Là dịch vụ mục tiêu, đóng vai trò cung cấp API. Dịch vụ này sẽ được cấu hình để có thể tạo ra nhiều loại lỗi khác nhau như: phản hồi lỗi, chậm phản hồi, phản hồi sai định dạng, hoặc ngừng hoạt động hoàn toàn. Mục tiêu là tạo điều kiện kiểm tra khả năng phát hiện và phản ứng lỗi.
- Service B: Là dịch vụ giám sát chủ động. Nó có nhiệm vụ định kỳ gửi yêu cầu HTTP đến Service A, theo dõi phản hồi và phân tích để phát hiện lỗi. Khi phát hiện có sự cố, Service B sẽ gửi một thông điệp cảnh báo dưới dạng JSON đến hệ thống tiếp nhận sự kiện.
- SNS (Simple Notification Service – mô phỏng): Là thành phần nhận thông báo lỗi từ Service B. SNS đóng vai trò truyền tải sự kiện lỗi tới các thành phần xử lý tiếp theo. Trong đồ án, SNS được mô phỏng bằng công cụ LocalStack.
- Lambda function (mô phỏng): Là chức năng phản ứng lỗi. Khi nhận được sự kiện từ SNS, Lambda sẽ phân tích nội dung và thực hiện hành động phù hợp như ghi log hoặc khởi động lại dịch vụ. Lambda trong đồ án cũng được triển khai bằng LocalStack và mã Python.
- LocalStack: Công cụ giả lập môi trường AWS trên máy cá nhân, giúp mô phỏng các dịch vụ như SNS và Lambda mà không cần truy cập vào AWS thật.

Thông qua mô hình này, đồ án mô phỏng một pipeline xử lý lỗi giống như trong môi trường AWS thực tế:

(1) Giám sát → (2) Phát hiện lỗi → (3) Gửi cảnh báo → (4) Phản ứng → (5) Ghi nhận kết quả.

Từ phần tiếp theo, từng loại lỗi sẽ được phân tích cụ thể theo cấu trúc: mô tả lỗi – cách AWS xử lý – cách mô phỏng trong đồ án.

Đồ án sẽ mô phỏng 3 lỗi sau:

- Lỗi hạ tầng: Sự cố gián đoạn hạ tầng

- Lỗi logic ứng dụng: Lỗi phản hồi phía ứng dụng (Application-level HTTP error)
- Lỗi hiệu năng: Lỗi vượt ngưỡng thời gian phản hồi

2.1.1 Sự cố gián đoạn hạ tầng

a, Mô tả lỗi

Lỗi " Sự cố gián đoạn hạ tầng " đề cập đến tình huống một thành phần trong hệ thống microservices ngừng hoạt động hoàn toàn, khiến các dịch vụ khác không thể kết nối hoặc tương tác với nó. Đây là một lỗi nghiêm trọng vì nó ảnh hưởng trực tiếp đến khả năng cung cấp chức năng của hệ thống.

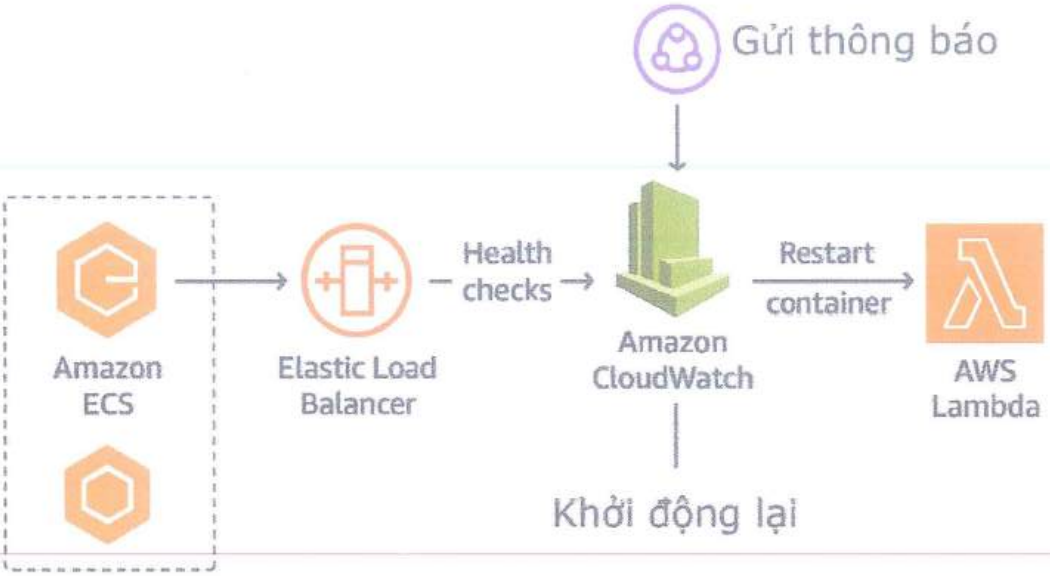
Một số nguyên nhân phổ biến có thể bao gồm:

- Sập tiến trình (crash) do lỗi logic, tràn bộ nhớ, lỗi runtime
- Dừng ngoài ý muốn do lỗi hệ điều hành, lỗi hạ tầng hoặc mất kết nối mạng
- Dừng thủ công trong quá trình vận hành, bảo trì mà không có cơ chế thay thế kịp thời
- Cạn kiệt tài nguyên (CPU, RAM, disk) khiến dịch vụ không thể duy trì trạng thái chạy

Khi một dịch vụ bị dừng, bất kỳ dịch vụ nào phụ thuộc vào nó có thể dẫn đến lỗi dây chuyền, gây mất ổn định toàn hệ thống nếu không có cơ chế phát hiện và phản ứng tự động.

b, Mô hình AWS phát hiện và khắc phục lỗi

Hình ảnh dưới đây mô tả các thành phần của AWS trong quá trình phát hiện và khắc phục lỗi



Hình 2.1: Mô hình AWS phát hiện và khắc phục lỗi sự cố gián đoạn hạ tầng

Amazon ECS

- Dịch vụ (container) chạy trên Amazon ECS.
- ECS là nơi khởi tạo và vận hành các microservices dưới dạng task hoặc service.

Elastic Load Balancer (ELB)

- ELB đóng vai trò là cổng vào phân phối request đến các container chạy trong ECS.
- ELB định kỳ thực hiện health check tới các container.
- Nếu container không phản hồi đúng hoặc bị dừng, ELB sẽ đánh dấu là "unhealthy" và loại ra khỏi load balancing.

Amazon CloudWatch

- CloudWatch theo dõi log và sự kiện từ ECS và ELB.

- Khi container bị dừng hoặc bị đánh dấu lỗi, CloudWatch ghi nhận trạng thái và có thể kích hoạt Alarm.
- CloudWatch cũng có thể trực tiếp thực hiện khôi phục container (nếu có Auto Recovery cấu hình) hoặc chuyển lỗi đi.

Send alerts

- Khi Alarm phát hiện sự cố, CloudWatch có thể cấu hình để gửi cảnh báo đến các hệ thống khác như SNS, email, Slack hoặc dashboard.

AWS Lambda

- CloudWatch có thể kích hoạt AWS Lambda function để thực hiện phản ứng tùy biến.
- Lambda có thể:
    - Khởi động lại container
    - Ghi log xử lý
    - Gửi báo cáo
    - Trigger thêm các workflow khác (ví dụ gửi đến Jira, mở ticket...)

Luồng hoạt động:

1. Một dịch vụ trong Amazon ECS bị dừng đột ngột do sự cố như crash, lỗi hệ thống, tràn bộ nhớ hoặc bị dừng ngoài ý muốn trong quá trình vận hành.
2. Elastic Load Balancer (ELB) thực hiện health check định kỳ đến container đó và không nhận được phản hồi.
3. ELB xác định container ở trạng thái "unhealthy" và chuyển thông tin phản hồi đến Amazon CloudWatch dưới dạng metric về tình trạng container.
4. Amazon CloudWatch Alarm được cấu hình để giám sát số lượng container bị unhealthy. Khi số lượng vượt ngưỡng trong một khoảng thời gian nhất định, alarm được kích hoạt.
5. CloudWatch gửi cảnh báo đến AWS SNS hoặc trực tiếp kích hoạt AWS Lambda để xử lý lỗi.
6. AWS Lambda thực hiện hành động khắc phục như:
    - Tự động khởi động lại container bị dừng

- Gửi cảnh báo đến quản trị viên hệ thống
- Ghi log chi tiết về sự cố để phục vụ việc giám sát và phân tích

7. Sau khi container được khôi phục thành công, hệ thống quay lại trạng thái ổn định và tiếp tục phục vụ các request như bình thường.

c, Mô phỏng trong đề án

Trong mô hình mô phỏng của đề án, việc xử lý lỗi dịch vụ bị dừng đột ngột được thực hiện thông qua một chuỗi thành phần được xây dựng để tương đương với luồng giám sát – phát hiện – phản ứng lỗi trên AWS.

AWS thực tế	Thành phần mô phỏng trong đồ án
Amazon ECS	Docker container chạy Service A – dịch vụ được lập trình để trả về lỗi 500/503
Elastic Load Balancer	Service B – đóng vai trò gửi request như client/ELB để kiểm tra phản hồi
Amazon CloudWatch	Service B theo dõi phản hồi và đánh giá lỗi. Có thể ghi log vào file/console
AWS SNS	SNS giả lập bằng LocalStack, nhận thông điệp lỗi từ Service B
AWS Lambda	Script Python nội bộ (thay Lambda) – xử lý cảnh báo và thực hiện hành động khắc phục

Các bước mô phỏng:

Bước 1: Người dùng dừng Service A bằng lệnh docker stop service-a

Bước 2: Service B tiếp tục gửi request HTTP đến Service A nhưng không nhận được phản hồi

Bước 3: Sau khi phát hiện lỗi, Service B tạo một thông điệp cảnh báo với đầy đủ thông tin (dịch vụ lỗi, loại lỗi, thời điểm xảy ra) và gửi thông điệp này đến hệ thống SNS giả lập được triển khai bằng công cụ LocalStack.

Bước 4: Do Lambda không thể chạy trên môi trường Windows, một đoạn mã Python nội bộ được sử dụng để thay thế. Thành phần này hoạt động như một hàm phản ứng: tiếp nhận thông điệp lỗi và thực hiện hành động khôi phục tương ứng, chẳng hạn như khởi động lại Service A.

2.1.2 Lỗi phản hồi phía ứng dụng (Application-level HTTP error)

a, Mô tả lỗi

Lỗi phản hồi phía ứng dụng (server-side response error) là loại lỗi xảy ra khi một dịch vụ microservice không thể xử lý yêu cầu từ client và trả về phản hồi lỗi ở phía backend. Những lỗi này thường biểu hiện qua các mã HTTP 5xx như 500 – Internal Server Error, 503 – Service Unavailable, nhưng không giới hạn ở các mã đó.

Nguyên nhân có thể đến từ nhiều tầng trong quá trình xử lý yêu cầu, bao gồm:

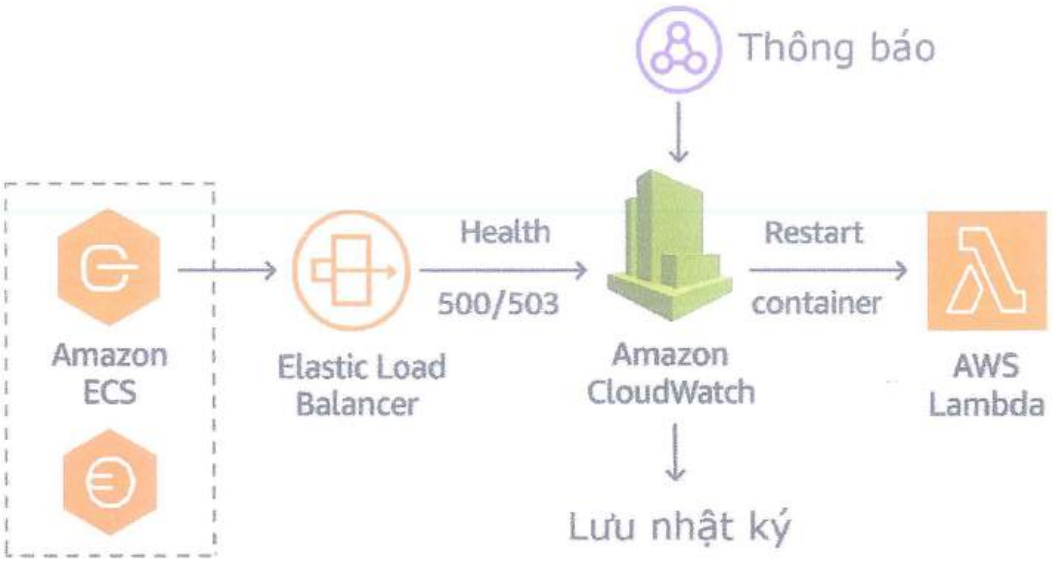
- Lỗi logic trong mã nguồn (ví dụ: chia cho 0, gọi biến null, lỗi xử lý điều kiện)
- Lỗi tương tác với cơ sở dữ liệu (query sai, timeout DB, khoá bị deadlock)
- Dịch vụ quá tải do lưu lượng cao hoặc tài nguyên hệ thống cạn kiệt
- Dịch vụ tạm thời bị mất kết nối với dịch vụ phụ thuộc khác
- Quá trình khởi tạo tài nguyên hoặc cấu hình sai trong lúc xử lý request

Lỗi này thường xảy ra ở lớp nghiệp vụ (business layer) của hệ thống microservices, và nếu không được phát hiện, cảnh báo và xử lý kịp thời, có thể gây ảnh hưởng dây chuyền đến các dịch vụ khác hoặc trải nghiệm người dùng cuối.

Đặc điểm của lỗi dạng này là dịch vụ vẫn đang chạy (khác với bị dừng hoàn toàn), nhưng không thể phản hồi đúng hoặc đầy đủ yêu cầu đầu vào.

b, Mô hình AWS phát hiện và khắc phục lỗi

Hình ảnh dưới đây mô tả các thành phần của AWS trong quá trình phát hiện và khắc phục lỗi



Hình 2.2: Mô hình AWS phát hiện và khắc phục lỗi phản hồi phía ứng dụng (Application-level HTTP error)

Amazon ECS

- Dịch vụ (container) chạy trên Amazon ECS.
- ECS là nơi khởi tạo và vận hành các microservices dưới dạng task hoặc service.

Elastic Load Balancer (ELB)

- ELB đóng vai trò là cổng vào phân phối request đến các container chạy trong ECS.
- ELB định kỳ thực hiện health check tới các container.
- Nếu container không phản hồi đúng hoặc bị dừng, ELB sẽ đánh dấu là "unhealthy" và loại ra khỏi load balancing.

Amazon CloudWatch

- CloudWatch theo dõi log và sự kiện từ ECS và ELB.

- Khi container bị dừng hoặc bị đánh dấu lỗi, CloudWatch ghi nhận trạng thái và có thể kích hoạt Alarm.
- CloudWatch cũng có thể trực tiếp thực hiện khôi phục container (nếu có Auto Recovery cấu hình) hoặc chuyển lỗi đi.

Send alerts

- Khi Alarm phát hiện sự cố, CloudWatch có thể cấu hình để gửi cảnh báo đến các hệ thống khác như SNS, email, Slack hoặc dashboard.

AWS Lambda

- CloudWatch có thể kích hoạt AWS Lambda function để thực hiện phản ứng tùy biến.
- Lambda có thể:
  - Khởi động lại container
  - Ghi log xử lý
  - Gửi báo cáo
  - Trigger thêm các workflow khác (ví dụ gửi đến Jira, mở ticket...)

Luồng hoạt động

1. Dịch vụ trong ECS cluster gặp sự cố, trả về phản hồi HTTP lỗi (500 hoặc 503).
2. Elastic Load Balancer (ELB) gửi request đến container của dịch vụ đó và nhận về mã lỗi 500/503.
3. ELB chuyển thông tin phản hồi lỗi đến Amazon CloudWatch dưới dạng metric.
4. CloudWatch Alarm được thiết lập để giám sát số lượng lỗi HTTP 500/503 trong một khoảng thời gian nhất định. Khi vượt ngưỡng, alarm được kích hoạt.
5. CloudWatch gửi alert đến SNS hoặc trực tiếp kích hoạt AWS Lambda.
6. AWS Lambda thực hiện hành động khắc phục:

- Tự động khởi động lại container lỗi

- Gửi cảnh báo đến hệ thống quản lý
- Ghi log xử lý

7. Hệ thống trở lại trạng thái ổn định.

c, Mô phỏng trong đồ án

Trong mô hình mô phỏng của đồ án, việc xử lý phản hồi mã lỗi được thực hiện thông qua một chuỗi thành phần được xây dựng để tương đương với luồng giám sát – phát hiện – phản ứng lỗi trên AWS.

AWS thực tế	Thành phần mô phỏng trong đồ án
Amazon ECS	Docker container chạy Service A – dịch vụ được lập trình để trả về lỗi 500/503
Elastic Load Balancer	Service B – đóng vai trò gửi request như client/ELB để kiểm tra phản hồi
Amazon CloudWatch	Service B theo dõi phản hồi và đánh giá lỗi. Có thể ghi log vào file/console
AWS SNS	SNS giả lập bằng LocalStack, nhận thông điệp lỗi từ Service B
AWS Lambda	Script Python nội bộ (thay Lambda) – xử lý cảnh báo và thực hiện hành động khắc phục

Các bước mô phỏng

Bước 1: Kịch bản lỗi:

- Service A được lập trình để trả về mã lỗi HTTP 500 hoặc 503 tại một thời điểm cụ thể (ngẫu nhiên hoặc theo chu kỳ).
- Ví dụ: truy cập endpoint /fail luôn trả về HTTP 500.

Bước 2: Phát hiện lỗi:

- Service B gửi request định kỳ đến Service A (giống như ELB hoặc client load balancer).

- Khi nhận được mã phản hồi HTTP 500 hoặc 503, Service B xác định có lỗi ứng dụng.

Bước 3: Gửi cảnh báo:

- Service B tạo thông điệp cảnh báo dưới dạng JSON (chứa mã lỗi, thời gian, dịch vụ bị lỗi…).
- Thông điệp này được gửi đến SNS topic giả lập trong LocalStack.

Bước 4: Phản ứng khắc phục lỗi:

- Do Lambda không khả dụng trên Windows, một script Python nội bộ đóng vai trò tiếp nhận cảnh báo và xử lý.
- Script này thực hiện:

  Ghi log lỗi

  Khởi động lại Service A bằng Docker

  In ra kết quả xử lý

Bước 5: Kiểm tra kết quả:

- Service A được khởi động lại.
- Service B tiếp tục kiểm tra, nếu nhận được HTTP 200 ở lần tiếp theo → lỗi được coi là khắc phục thành công.
- Các log của Service B và script sẽ là bằng chứng xử lý.

2.1.3 Lỗi vượt ngưỡng thời gian phản hồi

a, Mô tả lỗi

Lỗi vượt ngưỡng thời gian phản hồi là tình huống xảy ra khi một dịch vụ trong hệ thống microservices mất quá nhiều thời gian để phản hồi một yêu cầu, vượt quá ngưỡng thời gian cho phép được thiết lập ở phía client hoặc gateway. Khi điều này xảy ra, kết nối thường bị đóng lại và người gọi nhận được lỗi vượt ngưỡng thời gian phản hồi, dù dịch vụ vẫn đang hoạt động.

Nguyên nhân của lỗi này có thể đến từ:

- Tắc nghẽn xử lý nội bộ, như vòng lặp tính toán phức tạp, chờ I/O hoặc gọi API phụ trợ bị chậm

- Tải hệ thống cao, dẫn đến việc các request bị hàng đợi hoặc xử lý chậm
- Lỗi truy vấn cơ sở dữ liệu, như truy vấn không tối ưu hoặc khoá bị chờ
- Thư viện bên ngoài hoặc dịch vụ phụ trợ phản hồi quá lâu
- Không kiểm soát thời gian xử lý các tác vụ không đồng bộ

Lỗi vượt ngưỡng thời gian phản hồi thường không được trả về dưới dạng mã lỗi HTTP cụ thể từ phía server, mà thường được ghi nhận từ phía client hoặc các thành phần trung gian (ELB, API Gateway) khi không nhận được phản hồi sau thời gian chờ định nghĩa trước (ví dụ 3s, 5s...).

Đặc trưng của lỗi này là dịch vụ không dừng hẳn, nhưng hoạt động không hiệu quả, làm suy giảm hiệu năng hệ thống, gây khó chịu cho người dùng và có thể kéo theo lỗi dây chuyền ở các microservice khác nếu không được giám sát và xử lý kịp thời.

b, Mô hình AWS phát hiện và khắc phục lỗi

Hình ảnh dưới đây mô tả các thành phần của AWS trong quá trình phát hiện và khắc phục lỗi vượt ngưỡng thời gian phản hồi

Hình 2.3: Mô hình AWS phát hiện và khắc phục lỗi vượt ngưỡng thời gian phản hồi

Amazon ECS

- Dịch vụ (container) chạy trên Amazon ECS.
- ECS là nơi khởi tạo và vận hành các microservices dưới dạng task hoặc service.

Elastic Load Balancer (ELB)

- ELB đóng vai trò là cổng vào phân phối request đến các container chạy trong ECS.
- ELB định kỳ thực hiện health check tới các container.
- Nếu container không phản hồi đúng hoặc bị dừng, ELB sẽ đánh dấu là "unhealthy" và loại ra khỏi load balancing.

Amazon CloudWatch

- CloudWatch theo dõi log và sự kiện từ ECS và ELB.
- Khi container bị dừng hoặc bị đánh dấu lỗi, CloudWatch ghi nhận trạng thái và có thể kích hoạt Alarm.
- CloudWatch cũng có thể trực tiếp thực hiện khôi phục container (nếu có Auto Recovery cấu hình) hoặc chuyển lỗi đi.

Send Alerts

- Khi Alarm phát hiện sự cố, CloudWatch có thể cấu hình để gửi cảnh báo đến các hệ thống khác như SNS, email, Slack hoặc dashboard.

AWS Lambda

- CloudWatch có thể kích hoạt AWS Lambda function để thực hiện phản ứng tùy biến.

- Lambda có thể:
    - Khởi động lại container
    - Ghi log xử lý
    - Gửi báo cáo
    - Trigger thêm các workflow khác (ví dụ gửi đến Jira, mở ticket...)

Luồng phát hiện và khắc phục lỗi vượt ngưỡng thời gian phản hồi

1. Một dịch vụ trong Amazon ECS mất quá nhiều thời gian để phản hồi request do quá tải, xử lý chậm hoặc tài nguyên bị nghẽn, gây ra lỗi vượt ngưỡng thời gian phản hồi.
2. Elastic Load Balancer (ELB) gửi request đến container nhưng không nhận được phản hồi trong thời gian quy định, từ đó đánh giá đây là một lỗi timeout.
3. ELB ghi nhận sự kiện vượt ngưỡng thời gian phản hồi và gửi metric này đến Amazon CloudWatch dưới dạng chỉ số phản hồi (response latency / failed health check).
4. CloudWatch Alarm được thiết lập để theo dõi số lượng lỗi vượt ngưỡng thời gian phản hồi hoặc độ trễ phản hồi trung bình. Khi vượt quá ngưỡng cho phép trong một khoảng thời gian, alarm được kích hoạt.
5. CloudWatch gửi cảnh báo đến AWS SNS hoặc trực tiếp kích hoạt AWS Lambda để phản ứng với sự cố.
6. AWS Lambda thực hiện hành động khắc phục như:
    - Tự động khởi động lại container chậm phản hồi
    - Gửi cảnh báo đến hệ thống quản trị hoặc các kênh thông tin liên quan

- Ghi log chi tiết về sự cố để phục vụ phân tích hiệu năng

7. Sau khi container được khởi động lại hoặc xử lý xong, dịch vụ khôi phục hoạt động bình thường và tiếp tục phục vụ các request đúng thời gian cho phép.

c, Mô phỏng trong đồ án

Trong mô hình mô phỏng của đồ án, việc xử lý lỗi vượt ngưỡng thời gian phản hồi được thực hiện thông qua một chuỗi thành phần được xây dựng để tương đương với luồng giám sát – phát hiện – phản ứng lỗi trên AWS.

So sánh thành phần AWS và mô phỏng tương ứng

AWS thực tế	**Thành phần mô phỏng trong đồ án**
Amazon ECS	Docker container chạy Service A – được cấu hình để phản hồi chậm nhằm mô phỏng lỗi timeout
Elastic Load Balancer	Service B – gửi request định kỳ đến Service A để đo thời gian phản hồi
Amazon CloudWatch	Service B theo dõi thời gian phản hồi, phát hiện khi vượt quá ngưỡng và xác định lỗi timeout
AWS SNS	SNS giả lập bằng LocalStack – nhận thông điệp lỗi timeout từ Service B
AWS Lambda	Script Python nội bộ (thay Lambda) – xử lý cảnh báo, có thể ghi log hoặc khởi động lại Service A

Các bước thực hiện mô phỏng

Bước 1: Kịch bản lỗi

- Service A được cấu hình để xử lý một số endpoint (ví dụ /slow) với độ trễ cao, vượt ngưỡng timeout cho phép (ví dụ 5 giây).

Bước 2: Phát hiện lỗi

- Service B gửi request định kỳ đến Service A và đo thời gian phản hồi.
- Nếu thời gian phản hồi vượt quá ngưỡng quy định (ví dụ 3 giây), Service B xác định có lỗi timeout.

Bước 3: Gửi cảnh báo

- Service B tạo một thông điệp cảnh báo dạng JSON chứa thông tin về lỗi timeout, thời điểm xảy ra, endpoint bị ảnh hưởng.
- Thông điệp được gửi đến SNS topic (mô phỏng bằng LocalStack).

Bước 4: Phản ứng khắc phục lỗi:

- Script Python nội bộ (đóng vai trò Lambda) được kích hoạt, tiếp nhận cảnh báo và xử lý:
  - Ghi log sự kiện
  - Gửi thông báo ra console hoặc file
  - (Tùy cấu hình) khởi động lại Service A

Bước 5: Kiểm tra kết quả:

- Service B tiếp tục giám sát. Nếu thời gian phản hồi trở lại bình thường, lỗi được xem là đã khắc phục thành công.
- Log kiểm tra là cơ sở xác nhận toàn bộ quy trình hoạt động chính xác.

2.2 Môi trường triển khai và công cụ sử dụng

Hệ thống mô phỏng được triển khai hoàn toàn trên máy tính cá nhân với kiến trúc container hóa và sử dụng các công cụ mã nguồn mở. Môi trường này đảm bảo tính linh hoạt, dễ tái tạo và không phụ thuộc vào tài khoản AWS thật, phù hợp mô phỏng hệ thống hiện đại.

2.2.1 Môi trường triển khai

Hệ điều hành: Windows 10 / 11 (64-bit)

CPU: Intel Core i5 hoặc tương đương

RAM: Tối thiểu 8 GB

Ổ cứng: SSD, còn trống ít nhất 10 GB

Mạng: Internet ổn định để tải công cụ và tài nguyên

2.2.2 Công cụ và công nghệ sử dụng

Ta sử dụng công cụ và công nghệ sau:

Công cụ / Công nghệ	Mục đích sử dụng
Java 17	Ngôn ngữ lập trình chính dùng để phát triển Service A và Service B
Spring Boot	Framework để xây dựng RESTful API, xử lý logic dịch vụ
Maven	Quản lý thư viện và build ứng dụng Java
Docker Desktop	Triển khai các service Java dưới dạng container
Docker Compose	Quản lý cấu hình đa container (Service A, Service B, LocalStack…)
LocalStack	Giả lập các dịch vụ AWS như SNS, Lambda
Postman	Gửi request kiểm tra API, mô phỏng lỗi từ client
IntelliJ IDEA	Môi trường phát triển và gỡ lỗi ứng dụng Java
Command Prompt	Quản lý container, giám sát log và chạy lệnh kiểm thử
AWS CLI	Giao tiếp với LocalStack thông qua dòng lệnh để tạo SNS, kiểm thử event, mô phỏng luồng AWS
Python	Nhận SNS, quyết định hành động khắc phục lỗi

2.3 Cấu hình hệ thống

2.3.1 Cấu hình chung

a, Docker và Docker Compose

Tạo Dockerfile cho Service A và Service B (Java Spring Boot), có vai trò đóng gói ứng dụng Spring Boot thành image Docker, để có thể triển khai và chạy chúng như các container độc lập trong hệ thống microservice. Đặt file này trong thư mục service-a/ hoặc service-b/.

```
FROM eclipse-temurin:17-jdk-alpine
VOLUME /tmp
ARG JAR_FILE=target/*.jar
COPY ${JAR_FILE} app.jar
ENTRYPOINT ["java","-jar","/app.jar"]
```

Viết file docker-compose.yml, đây là file cấu hình trung tâm cho phép định nghĩa nhiều container ứng với từng service như service-a, service-b, localstack... và thiết lập cách các service này liên kết, tương tác với nhau. Với docker-compose.yml, người dùng có thể dễ dàng chạy toàn bộ hệ thống chỉ bằng một lệnh duy nhất (docker compose up), thay vì phải chạy từng container thủ công. Ngoài ra, file này còn giúp cấu hình chi tiết cổng kết nối, biến môi trường, volume lưu trữ, và các phụ thuộc giữa các dịch vụ, đảm bảo hệ thống khởi động đúng thứ tự và hoạt động ổn định. Tạo một file docker-compose.yml ở thư mục gốc (nơi chứa service-a/, service-b/)

```
version: '3.8'
services:
  service-a:
    build:
      context: ./servicea
```

```
    container_name: service-a
    ports:
      - "8080:8080"

  service-b:
    build:
      context: ./serviceb
    container_name: service-b
    ports:
      - "8081:8081"
    depends_on:
      - service-a

  localstack:
    image: localstack/localstack:latest
    container_name: localstack
    ports:
      - "4566:4566"
    environment:
      - SERVICES=sns,sqs
      - DEFAULT_REGION=us-east-1
    volumes:
      - "./localstack:/var/lib/localstack"
      - "/var/run/docker.sock:/var/run/docker.sock"
```

SNS thường được dùng để gửi cảnh báo khi có lỗi xảy ra, ví dụ như: service bị dừng, phản hồi lỗi HTTP, hoặc xảy ra timeout. Khi một service phát hiện lỗi, nó sẽ gửi thông báo lên SNS topic, dưới đây là câu lệnh tạo SNS

```
aws --endpoint-url=http://localhost:4566 sns create-topic --name
service-alerts
aws --endpoint-url=http://localhost:4566 sns list-topics
```

SQS đóng vai trò là nơi trung chuyển thông điệp cảnh báo lỗi, được gửi từ Service B thông qua SNS. Sau đó, một script Python nội bộ sẽ đọc các thông điệp từ hàng đợi này để khởi động lại Service A hoặc thực hiện các hành động khắc phục tương ứng. Dưới đây là câu lệnh tạo, đọc tin nhắn và phân quyền

```
aws --endpoint-url=http://localhost:4566 sqs create-queue --queue-name
alert-queue
aws --endpoint-url=http://localhost:4566 sns subscribe --topic-arn
arn:aws:sns:ap-southeast-1:000000000000:service-alerts --protocol sqs --
notification-endpoint arn:aws:sqs:ap-southeast-1:000000000000:alert-
queue
aws --endpoint-url=http://localhost:4566 sqs set-queue-attributes --queue-
url http://localhost:4566/000000000000/alert-queue --attributes
"{\"Policy\":\"{\\\"Version\\\":\\\"2012-10-
17\\\",\\\"Statement\\\":[{\\\"Effect\\\":\\\"Allow\\\",\\\"Principal\\\":\\\"*\\\",
\\\"Action\\\":\\\"sqs:SendMessage\\\",\\\"Resource\\\":\\\"arn:aws:sqs:ap-
southeast-1:000000000000:alert-
queue\\\",\\\"Condition\\\":{\\\"ArnEquals\\\":{\\\"aws:SourceArn\\\":\\\"arn
:aws:sns:ap-southeast-1:000000000000:service-alerts\\\"}}}]}\"}"
aws --endpoint-url=http://localhost:4566 sqs receive-message --queue-url
http://localhost:4566/000000000000/alert-queue
```

b, Java & Spring Boot

Vai trò trong hệ thống

Thành phần	Mục đích sử dụng
Service A	Mô phỏng các lỗi thường gặp như HTTP 500, timeout, hoặc dừng dịch vụ
Service B	Giám sát trạng thái Service A, phát hiện lỗi và gửi sự kiện đến hệ thống cảnh báo

Tiếp theo, tùy từng vào các lỗi sẽ có các dòng lệnh xử lý riêng

2.3.2 Cấu hình mô phỏng lỗi sự cố gián đoạn hạ tầng

Để mô phỏng lỗi gián đoạn hạ tầng trong môi trường microservice, hai ứng dụng Spring Boot là Service A và Service B đã được triển khai, với các chức năng cụ thể như sau:

1. Service A – Dịch vụ mục tiêu

Service A là một REST API đơn giản. Một controller với endpoint /hello được xây dựng nhằm trả về phản hồi bình thường, phục vụ mục đích kiểm thử hoạt động ổn định của dịch vụ:

```
@RestController
public class HelloController {
    @GetMapping("/hello")
    public String hello() {
        return "Service A is running.";
    }
}
```

Service A được đóng gói bằng Docker thông qua Dockerfile, sau đó chạy trong container tên là service-a.

2. Service B – Dịch vụ giám sát

Service B có nhiệm vụ định kỳ hoặc theo yêu cầu gửi request đến Service A. Một controller với endpoint /check/down được xây dựng nhằm thực hiện chức năng kiểm tra:

```
        @GetMapping("/check/down")
public String checkServiceA() {
   try {
      String response = restTemplate.getForObject("http://service-a:8080/hello",
String.class);
      return "Service A response: " + response;
   } catch (Exception e) {
      String alertMessage = """
         [ALERT]
         Service: Service A
         Type: Connection Refused
         Time: %s""".formatted(LocalDateTime.now());
      String topicArn = "arn:aws:sns:ap-southeast-1:000000000000:service-
alerts";
      PublishRequest request = PublishRequest.builder()
            .topicArn(topicArn)
            .message(alertMessage)
            .subject("Microservice Alert - Service A Down")
            .build();

      snsClient.publish(request);

      return "Service A is DOWN! SNS Alert Sent.";
   }
}
```

Học viên cấu hình RestTemplate với tên host service-a để gọi nội bộ giữa các container thông qua mạng Docker Compose.

3. Cấu hình SNS client trong Service B

AWS SDK v2 được sử dụng cùng với cấu hình SNS client để gửi thông báo đến môi trường giả lập LocalStack:

```
@Bean
public SnsClient snsClient() {
  return SnsClient.builder()
    .endpointOverride(URI.create("http://localstack:4566"))
    .region(Region.AP_SOUTHEAST_1)
    .credentialsProvider(StaticCredentialsProvider.create(
      AwsBasicCredentials.create("test", "test")
    ))
    .build();
}
```

4. Tạo Python script tự động khôi phục service

Một script Python chạy nền được xây dựng để lắng nghe tin nhắn từ hàng đợi SQS. Khi phát hiện thông báo lỗi "Service A is DOWN", script này sẽ thực hiện lệnh Docker nhằm khởi động lại container tương ứng:

```
def restart_service_a():
  subprocess.run(["docker", "start", "service-a"])


def poll_messages():
  while True:
    messages = sqs.receive_message(...)
    for msg in messages:
```

```
if "Service A is DOWN" in msg['Body']:
    restart_service_a()
sqs.delete_message(...)
```

2.3.3 Cấu hình mô phỏng lỗi phản hồi phía ứng dụng (Application-level HTTP error)

Để mô phỏng lỗi phản hồi phía ứng dụng trong hệ thống microservice, hai service Spring Boot là Service A và Service B tiếp tục được sử dụng, chạy trong container và giao tiếp nội bộ thông qua Docker Compose. Trong kịch bản này, Service A vẫn hoạt động, nhưng cố tình trả về mã lỗi HTTP để mô phỏng tình huống lỗi logic trong xử lý yêu cầu. Service B đóng vai trò giám sát và gửi cảnh báo nếu nhận được phản hồi lỗi từ Service A.

1. Service A – Trả về lỗi có chủ đích

Trong Service A, các endpoint giả lập lỗi được bổ sung để phục vụ quá trình mô phỏng:

```
@GetMapping("/error-500")
@ResponseStatus(HttpStatus.INTERNAL_SERVER_ERROR)
public String simulateError500() {
    return "Lỗi 500 - Internal Server Error";
}

@GetMapping("/error-503")
@ResponseStatus(HttpStatus.SERVICE_UNAVAILABLE)
public String simulateError503() {
    return "Lỗi 503 - Service Unavailable";
}
```

Các endpoint này được thiết kế nhằm chủ động tạo ra lỗi HTTP khi Service B thực hiện gọi API, phục vụ mục đích mô phỏng và kiểm thử khả năng phát hiện lỗi trong hệ thống.

2. Service B – Phát hiện lỗi HTTP

Tại Service B, một API /check/error500 được triển khai với nhiệm vụ gửi yêu cầu đến Service A thông qua endpoint /error-500. Nếu phát hiện mã lỗi HTTP từ phía Service A (ví dụ 500 hoặc 503), Service B sẽ gửi cảnh báo đến SNS topic đã cấu hình:

```
@GetMapping("/check/error500")
public String checkError500FromServiceA() {
  try {
    String response = restTemplate.getForObject("http://service-
a:8080/error-500", String.class);
    return "Service A response: " + response;
  } catch (HttpServerErrorException ex) {
    String alertMessage = String.format("""
        [ALERT - HTTP ERROR]
        Service: Service A
        Type: HTTP %s (%s)
        Time: %s
        """, ex.getStatusCode(), ex.getStatusText(),
LocalDateTime.now());

    snsClient.publish(PublishRequest.builder()
        .topicArn(topicArn)
        .message(alertMessage)
        .subject("Service A - HTTP ERROR")
        .build());
```

```
            return "Service A returned error: " + ex.getStatusCode();
        }
    }
```

RestTemplate được sử dụng để gửi yêu cầu và xử lý ngoại lệ HttpServerErrorException, nhằm nhận diện các lỗi HTTP phát sinh từ phía Service A.

3. Cảnh báo SNS và tiếp nhận qua SQS

Khi Service B phát hiện lỗi HTTP và gửi cảnh báo đến SNS, hệ thống đã cấu hình SNS forwarding đến một hàng đợi SQS. Mỗi cảnh báo lỗi đều được chuyển vào hàng đợi với nội dung dạng [ALERT - HTTP ERROR].

4. Python script xử lý lỗi thông minh

Script Python được tái sử dụng từ mô phỏng lỗi trước, với logic được điều chỉnh để phân biệt loại lỗi. Nếu thông điệp nhận từ hàng đợi SQS là lỗi HTTP, script chỉ thực hiện ghi log mà không khởi động lại dịch vụ.:

```
def should_restart_service(message: str) -> bool:
    critical_keywords = ["Service A is DOWN", "Connection Refused"]
    return any(keyword in message for keyword in critical_keywords)

# Trong hàm xử lý message
if should_restart_service(alert_message):
    restart_service_a()
else:
    print(" Lỗi HTTP - không khởi động lại service-a.")
```

2.3.4 Cấu hình mô phỏng lỗi vượt ngưỡng thời gian phản hồi

Trong hệ thống microservice, lỗi timeout xảy ra khi một dịch vụ gửi yêu cầu đến dịch vụ khác nhưng không nhận được phản hồi trong thời gian cho phép. Để mô phỏng lỗi này, hai service Spring Boot là Service A và Service B tiếp tục được sử

dụng trong quá trình mô phỏng. Trong đó, Service A cố tình trì hoãn phản hồi, còn Service B cấu hình thời gian chờ thấp hơn, nhằm tạo điều kiện phát sinh lỗi timeout.

1. Service A – Trì hoãn phản hồi cố ý

Một endpoint /timeout được bổ sung trong Service A, cho phép dịch vụ này tạm dừng (delay) trong một khoảng thời gian nhất định (ví dụ: 10 giây) trước khi gửi phản hồi:

```
@GetMapping("/timeout")
public String simulateTimeout() {
  try {
    Thread.sleep(10000); // 10 giây
  } catch (InterruptedException e) {
    e.printStackTrace();
  }
  return "Đã trả về sau khi delay.";
}
```

Mục đích của đoạn mã này là khiến Service A phản hồi quá lâu, trong khi Service B chỉ cho phép chờ trong 3 giây.

2. Service B – Giám sát và cấu hình timeout

Tại Service B, RestTemplate được cấu hình với thời gian chờ ngắn nhằm chủ động tạo ra lỗi phản hồi chậm (timeout).:

```
@Bean
public RestTemplate restTemplate() {
  SimpleClientHttpRequestFactory factory = new
SimpleClientHttpRequestFactory();
  factory.setConnectTimeout(3000); // 3 giây
  factory.setReadTimeout(3000);
```

```
        return new RestTemplate(factory);
    }
```

Tiếp theo, một controller với endpoint /check/timeout được triển khai trong Service B để gửi yêu cầu đến endpoint /timeout của Service A:

```
    @GetMapping("/check/timeout")
    public String checkTimeoutFromServiceA() {
        try {
            String response = restTemplate.getForObject("http://service-
a:8080/timeout", String.class);
            return "Service A response: " + response;
        } catch (Exception e) {
            String alertMessage = String.format("""
                    [ALERT - TIMEOUT]
                    Service: Service A
                    Type: TimeoutException
                    Time: %s
                    """, LocalDateTime.now());

            snsClient.publish(PublishRequest.builder()
                    .topicArn(topicArn)
                    .message(alertMessage)
                    .subject("Service A - Timeout")
                    .build());

            return "Timeout khi gọi Service A - đã gửi cảnh báo SNS.";
        }
    }
```

3. Gửi cảnh báo và xử lý hậu kỳ

Tương tự như hai lỗi trước, Service B gửi thông điệp cảnh báo timeout đến SNS (trên LocalStack). SNS chuyển tiếp cảnh báo sang SQS queue. Script Python chạy nền sẽ nhận message và ghi log. Tuy nhiên, timeout không được xem là lỗi nghiêm trọng nên không thực hiện khởi động lại Service A.

Trong script Python, loại lỗi được xác định dựa trên nội dung của thông điệp nhận từ hàng đợi:

```
if "TimeoutException" in alert_message:
    print("⏱ Phát hiện lỗi timeout - ghi log, không khởi động lại.")
```

2.4 Tổng kết chương

Trong chương này, đề án tập trung phân tích, thiết kế và triển khai hệ thống phát hiện và phục hồi lỗi trong môi trường microservice. Hệ thống được xây dựng dựa trên kiến trúc Docker, trong đó các dịch vụ được đóng gói riêng biệt và điều phối bằng Docker Compose nhằm đảm bảo tính linh hoạt và khả năng mở rộng.

Các thành phần chính bao gồm:

- Dịch vụ nghiệp vụ chính (Service A, Service B),
- Thành phần giám sát và cảnh báo được phát triển bằng Spring Boot,
- Môi trường mô phỏng hạ tầng AWS sử dụng LocalStack, với các dịch vụ như SNS (Simple Notification Service) và SQS (Simple Queue Service),
- Script Python đóng vai trò xử lý cảnh báo từ hàng đợi SQS và thực hiện hành động phù hợp (ví dụ: khởi động lại container hoặc ghi log).

Phần phân tích đã làm rõ các yêu cầu chức năng và phi chức năng, từ đó thiết kế kiến trúc hệ thống với các luồng tương tác giữa các dịch vụ. Hệ thống được cấu hình để có thể nhận diện và phản ứng linh hoạt trước nhiều loại lỗi khác nhau, đảm bảo tính sẵn sàng và phục hồi tự động khi xảy ra sự cố.

Việc sử dụng công nghệ mã nguồn mở giúp hệ thống dễ triển khai, tiết kiệm tài nguyên và phù hợp với môi trường nghiên cứu. Chương tiếp theo sẽ đánh giá hiệu quả hoạt động của hệ thống, rút ra điểm mạnh, hạn chế và tiềm năng mở rộng trong tương lai.

Chương 3 - MÔ PHỎNG, ĐÁNH GIÁ VÀ KẾT LUẬN

Chương 3 Tiến hành mô phỏng và tổng kết toàn bộ kết quả đạt được, phân tích ưu điểm – hạn chế của hệ thống, so sánh với các phương pháp khác, ứng dụng thực tiễn, định hướng, khả năng tích hợp và đưa ra các định hướng phát triển tiếp theo.

3.1 Triển khai mô phỏng

3.1.1 Mô phỏng lỗi sự cố gián đoạn hạ tầng

Bước 1: Dừng container của Service A

Việc dừng hoạt động của container service-a được thực hiện một cách chủ động nhằm mô phỏng tình huống lỗi hệ thống hoặc dịch vụ bị crash đột ngột, thông qua lệnh dòng lệnh:

```
docker stop service-a
```

```
E:\ProjectMaster\ProjectMicroservice\microservice-ab>docker stop service-a
service-a

E:\ProjectMaster\ProjectMicroservice\microservice-ab>docker ps
CONTAINER ID   IMAGE                          COMMAND                  CREATED             STATUS
PORTS                                                 NAMES
067ea6c6ae95   microservice-ab-service-b       "java -jar /app.jar"     About an hour ago   Up About an hour
0.0.0.0:8081->8081/tcp                                service-b
81ce2723f0d8   localstack/localstack:latest   "docker-entrypoint.sh"   3 hours ago         Up 3 hours (healthy)
4510-4559/tcp, 5678/tcp, 0.0.0.0:4566->4566/tcp       localstack
```

Hình 3.1: Ảnh mô phỏng dừng service A

Sau khi thực hiện lệnh này, Service A ngừng hoạt động và không còn khả năng phản hồi các yêu cầu HTTP từ các dịch vụ khác.

Bước 2: Service B gửi request và phát hiện lỗi

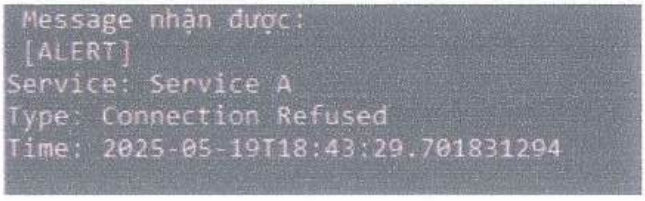
Trong Service B, một endpoint giám sát có địa chỉ /check/down được thiết lập để gửi HTTP request đến Service A thông qua địa chỉ nội bộ http://service-a:8080/hello.

Hình 3.2: Ảnh mô phỏng gọi từ Service B đến Service A

Do Service A đã bị dừng nên Service B không nhận được phản hồi, và ngoại lệ kết nối (Connection Refused) được ném ra.

Tại thời điểm đó, Service B sẽ ghi nhận đây là một lỗi nghiêm trọng và khởi tạo thông điệp cảnh báo dưới định dạng:

```
[ALERT]
Service: Service A
Type: Connection Refused
Time: <timestamp>
```



Hình 3.3: Ảnh tin nhắn SNS nhận được khi service A bị dừng

Bước 3: Gửi cảnh báo qua SNS và lưu vào hàng đợi SQS

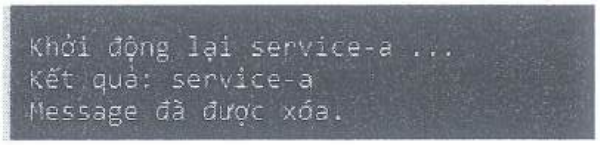
Dịch vụ SNS được cấu hình để tự động chuyển tiếp các thông điệp cảnh báo đến một hàng đợi SQS có tên alert-queue. SQS đảm nhận vai trò lưu trữ tạm thời tất cả các thông báo lỗi, đảm bảo tính không mất mát và khả năng xử lý bất đồng bộ. Mỗi message trong SQS đều chứa nội dung chi tiết về sự cố, được ghi lại để phục vụ cho bước khắc phục lỗi tiếp theo.

Bước 4: Tự động khởi động lại dịch vụ thông qua script Python

Cuối cùng, một script Python chạy nền được triển khai với nhiệm vụ liên tục lắng nghe các thông điệp mới trong hàng đợi alert-queue. Nếu thông điệp chứa từ khóa như "Service A is DOWN" hoặc "Connection Refused", script sẽ tự động thực hiện lệnh:

```
docker start service-a
```

giúp khôi phục lại dịch vụ bị sự cố. Sau khi xử lý xong, message tương ứng sẽ được xóa khỏi hàng đợi để tránh trùng lặp.



Hình 3.4: Ảnh service A đang được khởi động lại và xóa tin nhắn trong hàng đợi

Kết quả mô phỏng cho thấy hệ thống có thể phát hiện chính xác tình huống dịch vụ bị dừng, gửi cảnh báo kịp thời và thực hiện khôi phục tự động trong thời gian ngắn. Đây là minh chứng rõ ràng cho khả năng tự phục hồi (self-healing) của hệ thống microservice khi được giám sát và điều phối hợp lý.

Hình 3.5: Ảnh hiển thị trạng thái service A đã được khởi động lại thành công

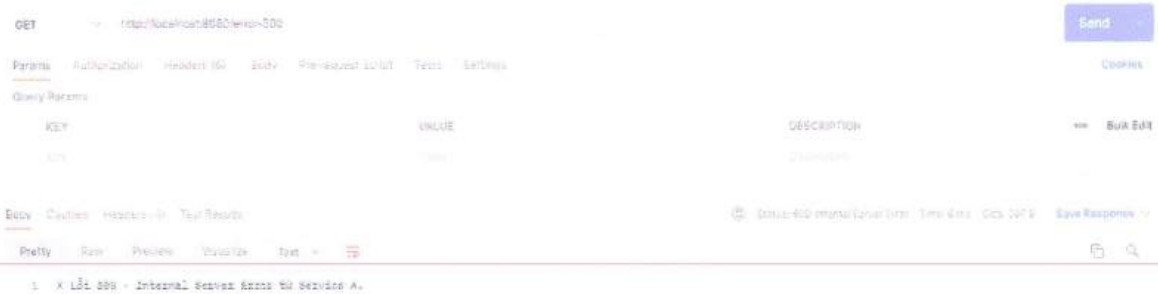
3.1.2 Mô phỏng lỗi phản hồi phía ứng dụng (Application-level HTTP error)

Lỗi phản hồi phía ứng dụng (Application-level HTTP error) xảy ra khi một dịch vụ vẫn đang hoạt động nhưng trả về mã lỗi phía server (5xx), thường do lỗi nội bộ hoặc không sẵn sàng xử lý yêu cầu. Đây là loại lỗi phổ biến trong các hệ thống vi mô, đặc biệt khi các thành phần phụ thuộc lẫn nhau trong chuỗi xử lý. Để mô phỏng lỗi phản hồi phía ứng dụng (Application-level HTTP error), hai endpoint mới được

bổ sung trong Service A với mục đích trả về mã lỗi có chủ đích, nhằm phục vụ quá trình mô phỏng và kiểm thử lỗi trong hệ thống

Bước 1: Service B gọi API và phát hiện lỗi

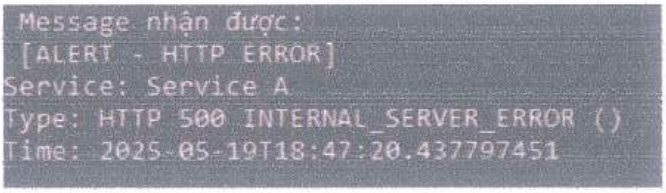
Một *endpoint /check/error500 được xây dựng trong Service B nhằm gửi* request đến endpoint /error-500 của Service A để mô phỏng lỗi HTTP 500. Nếu phản hồi trả về là mã lỗi 5xx, ứng dụng sẽ bắt lỗi thông qua HttpServerErrorException.



Hình 3.6: Ảnh Service B gọi sang service A lấy lỗi phản hồi

Lúc này, Service B sẽ lập tức tạo một thông điệp cảnh báo với nội dung bao gồm mã lỗi, loại lỗi, thời gian xảy ra và gửi lên hệ thống SNS:

```
[ALERT - HTTP ERROR]
Service: Service A
Type: HTTP 500 (Internal Server Error)
Time: <timestamp>
```



Hình 3.7: Tin nhắn SNS nhận được khi service B nhận được mã lỗi từ service A

Bước 3: Gửi thông báo đến SNS và forward sang SQS

Giống như lỗi trước đó, SNS topic nhận cảnh báo được cấu hình chuyển tiếp sang hàng đợi alert-queue của SQS. Điều này cho phép hệ thống lưu trữ lỗi tạm thời và xử lý một cách bất đồng bộ, tránh làm gián đoạn luồng chính của Service B.

Bước 4: Script xử lý thông minh và phân loại lỗi

Trong script Python xử lý phía backend, cơ chế nhận diện loại lỗi được bổ sung dựa trên nội dung của thông điệp cảnh báo. Nếu là lỗi HTTP (chứa từ "HTTP ERROR" hoặc mã "500", "503"), script chỉ log lỗi mà không thực hiện hành động khôi phục (do dịch vụ vẫn đang hoạt động):

```
if "HTTP ERROR" in alert_message:
    print("Phát hiện lỗi HTTP từ Service A - ghi log, không khởi động lại.")
```

```
Không phải lỗi nghiêm trọng → chỉ log.
Message đã được xóa.
```

Hình 3.8: Thông báo hiển thị sau khi xử lý lỗi phản hồi

3.1.3 Mô phỏng lỗi vượt ngưỡng thời gian phản hồi

Lỗi timeout là một trong những lỗi phổ biến trong hệ thống microservice, xảy ra khi một dịch vụ gửi yêu cầu đến dịch vụ khác nhưng không nhận được phản hồi trong thời gian cho phép. Đây là lỗi tiềm ẩn gây ảnh hưởng đến hiệu năng và trải nghiệm người dùng, đặc biệt trong môi trường phân tán với nhiều thành phần tương tác chéo. Để mô phỏng tình huống này, Service A được triển khai với hành vi trì hoãn phản hồi, trong khi Service B được cấu hình với thời gian chờ ngắn hơn, tạo điều kiện để phát sinh lỗi phản hồi chậm (timeout).

Bước 1: Gửi yêu cầu kiểm tra và phát hiện lỗi

Hình 3.9: Service B gọi sang Service A để kiểm tra lỗi time out

Một API có endpoint /check/timeout được xây dựng trong Service B nhằm gửi yêu cầu đến endpoint /timeout của Service A. Khi Service A chưa kịp phản hồi

trong 3 giây, RestTemplate ném ra lỗi kết nối hoặc timeout, và Service B xử lý lỗi này bằng cách gửi cảnh báo đến SNS với nội dung:

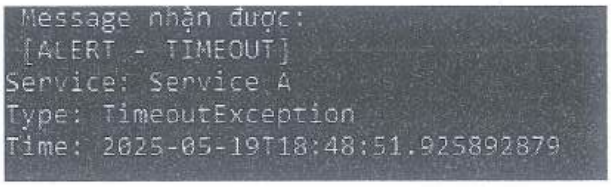
```
[ALERT - TIMEOUT]
Service: Service A
Type: TimeoutException
Time: <timestamp>
```

Bước 2: Chuyển thông báo qua SNS → SQS và xử lý ghi log

Thông báo lỗi timeout được gửi từ SNS đến hàng đợi SQS alert-queue. Script Python chạy nền tiếp tục nhận message từ hàng đợi.

Khi phát hiện nội dung chứa "TimeoutException", script sẽ **chỉ ghi log**, không thực hiện restart Service A, vì đây không phải lỗi ngừng dịch vụ mà chỉ là lỗi chậm phản hồi.

Hình 3.11: SNS nhận được khi service B nhận được lỗi time out



Hình 3.10: Thông báo sau khi nhận được lỗi time out

Thông qua mô phỏng này, hệ thống đã thể hiện khả năng phát hiện lỗi timeout một cách chính xác và xử lý phù hợp mà không can thiệp khôi phục không cần thiết. Điều này giúp nâng cao độ ổn định và khả năng thích ứng linh hoạt của hệ thống microservice trong môi trường có nhiều biến động về hiệu năng và độ trễ.

3.2 Phân tích kết quả sau khi mô phỏng

Khi thực hiện lệnh docker stop service-a, container của Service A ngừng hoạt động hoàn toàn. Trong lần kiểm tra tiếp theo qua API /check/down, Service B không thể kết nối đến Service A và ném ra lỗi Connection Refused. Hệ thống ngay lập tức

gửi cảnh báo đến SNS và lưu vào hàng đợi SQS. Script Python lắng nghe tin nhắn đã nhận thông báo này, xác định đây là lỗi nghiêm trọng, và thực hiện lệnh docker start service-a để khởi động lại container. Quá trình khôi phục diễn ra thành công trong thời gian ngắn, minh chứng cho khả năng tự chữa lành (self-healing) của hệ thống.

3.2.1 Lỗi sự cố gián đoạn hạ tầng

Khi thực hiện lệnh docker stop service-a, container của Service A ngừng hoạt động hoàn toàn. Trong lần kiểm tra tiếp theo qua API /check/down, Service B không thể kết nối đến Service A và ném ra lỗi Connection Refused. Hệ thống ngay lập tức gửi cảnh báo đến SNS và lưu vào hàng đợi SQS. Script Python lắng nghe tin nhắn đã nhận thông báo này, xác định đây là lỗi nghiêm trọng, và thực hiện lệnh docker start service-a để khởi động lại container. Quá trình khôi phục diễn ra thành công trong thời gian ngắn, minh chứng cho khả năng tự chữa lành (self-healing) của hệ thống.

3.2.2 Lỗi phản hồi phía ứng dụng (Application-level HTTP error)

Trong trường hợp gọi API /check/error500, Service B nhận được mã lỗi HTTP 500 từ phía Service A. Mặc dù Service A vẫn đang chạy, nhưng lỗi logic nội bộ khiến hệ thống không phản hồi đúng như kỳ vọng. Service B đã phát hiện mã lỗi HTTP thông qua HttpServerErrorException và gửi cảnh báo phù hợp lên SNS. Tin nhắn tiếp tục được lưu trong SQS, và script Python ghi nhận nội dung nhưng không thực hiện khởi động lại dịch vụ, đúng với yêu cầu xử lý lỗi không nghiêm trọng. Điều này cho thấy hệ thống phân biệt được lỗi logic với lỗi hệ thống.

3.2.3 Lỗi vượt ngưỡng thời gian phản hồi

Khi gọi đến endpoint /check/timeout, Service A trì hoãn phản hồi trong 10 giây trong khi Service B chỉ cho phép chờ tối đa 3 giây. Kết quả là một SocketTimeoutException được ném ra, và Service B xử lý lỗi này tương tự như lỗi trước đó bằng cách gửi thông báo lỗi "TimeoutException" lên SNS → SQS. Python script ghi nhận lỗi timeout nhưng không khởi động lại service, vì dịch vụ vẫn hoạt động. Việc này cho thấy hệ thống có khả năng giám sát hiệu năng và phản hồi chậm, một yếu tố quan trọng trong thực tế vận hành.

3.3 Đánh giá hiệu quả

Sau khi tiến hành mô phỏng ba loại lỗi phổ biến trong hệ thống microservice, các kết quả thực nghiệm đã được ghi nhận và phân tích nhằm đánh giá mức độ hiệu quả của giải pháp phát hiện và khắc phục lỗi được đề xuất.

3.3.1 Độ chính xác trong phát hiện lỗi

Hệ thống đã cho thấy khả năng phát hiện lỗi chính xác theo từng loại:

- Lỗi sự cố gián đoạn hạ tầng được phát hiện thông qua lỗi kết nối (Connection Refused) và được xử lý khôi phục tự động bằng cách khởi động lại container.
- Lỗi phản hồi phía ứng dụng (Application-level HTTP error) được bắt thông qua ngoại lệ HTTP và phân biệt rõ ràng với lỗi hệ thống. Hệ thống chỉ ghi nhận và gửi cảnh báo mà không can thiệp khôi phục.
- Lỗi vượt ngưỡng thời gian phản hồi được xử lý thông qua cấu hình RestTemplate với thời gian chờ ngắn, phát hiện đúng tình huống phản hồi chậm từ dịch vụ đích.

Tất cả các tình huống đều được ghi nhận đúng bản chất, không có trường hợp nào phát hiện nhầm loại lỗi hay phản ứng sai lệch.

3.3.2 Khả năng cảnh báo kịp thời và đầy đủ

Dịch vụ giám sát (Service B) đã gửi thông điệp cảnh báo đầy đủ thông tin gồm: tên dịch vụ lỗi, loại lỗi, và thời điểm xảy ra. Các cảnh báo đều được gửi thành công đến **SNS** và lưu trữ tại hàng đợi **SQS**, đảm bảo độ tin cậy và không mất mát thông tin. Thời gian từ lúc lỗi xảy ra đến khi thông điệp xuất hiện trong SQS chỉ trong khoảng 1–2 giây, cho thấy hiệu năng hệ thống là đạt yêu cầu.

3.3.3 Tính chủ động trong khôi phục

Script Python xử lý hậu kỳ đã lắng nghe hàng đợi SQS liên tục và tự động thực hiện hành động phù hợp. Đối với lỗi dừng dịch vụ, container được khởi động lại thành công trong thời gian rất ngắn. Với lỗi HTTP và timeout, hệ thống chủ động ghi log và không khởi động lại dịch vụ không cần thiết, giúp tránh các thao tác thừa gây ảnh

hưởng đến vận hành. Tính năng này góp phần tạo nên cơ chế tự chữa lành (self-healing) cho hệ thống.

3.3.4 Tính linh hoạt và mở rộng

Kiến trúc hệ thống được xây dựng dựa trên Docker, Spring Boot, và AWS giả lập (LocalStack) cho phép mở rộng dễ dàng. Các lỗi khác như lỗi CPU cao, lỗi database, hoặc lỗi vòng lặp vô hạn hoàn toàn có thể tích hợp thêm vào quy trình giám sát mà không cần thay đổi nhiều cấu trúc hiện tại.

Từ các kết quả thực nghiệm, có thể khẳng định rằng giải pháp phát hiện và phản ứng lỗi trong hệ thống microservice được đề xuất là khả thi, hiệu quả và phù hợp với các yêu cầu thực tế của một hệ thống phân tán hiện đại.

3.4 So sánh với các giải pháp hiện có

Việc phát hiện và phục hồi lỗi trong hệ thống microservice có thể được thực hiện theo nhiều cách, từ các phương pháp thủ công, tích hợp riêng lẻ từng thành phần, đến các giải pháp đồng bộ sử dụng nền tảng đám mây như AWS. Trong khuôn khổ đồ án, giải pháp giám sát lỗi dịch vụ đã được triển khai dựa trên hệ sinh thái AWS, sử dụng các dịch vụ như SNS và SQS, kết hợp với script xử lý tùy biến nhằm thực hiện cơ chế phục hồi. Sau đây là phân tích so sánh hiệu quả của cách làm này với các phương pháp khác.

3.4.1 Tính kịp thời trong phát hiện lỗi

Trong giải pháp sử dụng AWS, việc phát hiện lỗi được tích hợp trực tiếp trong logic dịch vụ (ví dụ Service B gọi Service A và kiểm tra kết quả), sau đó cảnh báo được đẩy ngay lập tức đến SNS – một hệ thống publish/subscribe hoạt động theo thời gian thực. Điều này giúp cảnh báo được phát hiện gần như tức thời (sub-second) ngay tại thời điểm lỗi xảy ra.

Ngược lại, các hệ thống giám sát truyền thống thường dựa trên các công cụ kiểm tra định kỳ như cron, shell script, hoặc đọc log từ hệ thống và phân tích sau. Khoảng thời gian phát hiện lỗi trong các hệ thống này có thể dao động từ vài giây đến vài phút, dẫn đến độ trễ trong phản ứng và dễ bỏ sót lỗi ngắn hạn [2].

Ngoài ra, trên AWS, các công cụ như CloudWatch Metrics và CloudWatch Alarms có thể được cấu hình để phát hiện lỗi vượt ngưỡng (ví dụ số lượng lỗi 5xx, số request timeout, CPU tăng cao...) và đẩy thông tin vào SNS gần như tức thì. Trong khi đó, nếu dùng ELK Stack hoặc APM nội bộ thì cần thu thập, index và query dữ liệu – vốn tiêu tốn thời gian và tài nguyên xử lý [4].

3.4.2 Tính tự động trong phản ứng và phục hồi lỗi

Giải pháp sử dụng SNS → SQS → Lambda hoặc script Python đã chứng minh khả năng phản ứng linh hoạt theo từng loại lỗi. Cụ thể:

- Lỗi sự cố gián đoạn hạ tầng → tự động docker start
- Lỗi vượt ngưỡng thời gian phản hồi→ ghi log và đánh dấu độ trễ
- Lỗi phản hồi phía ứng dụng (Application-level HTTP error)→ gửi thông báo nhưng không restart

Hành động xử lý có thể được thực thi ngay qua Lambda (AWS) hoặc script Python tự động, không cần sự can thiệp của con người. Điều này tạo nên cơ chế tự phục hồi (self-healing) – vốn là một tiêu chí quan trọng trong thiết kế hệ thống phân tán hiện đại.

Trong khi đó, các hệ thống không sử dụng cloud hoặc chưa tích hợp Lambda thường chỉ dừng lại ở mức cảnh báo. Việc phục hồi dịch vụ phụ thuộc vào thao tác thủ công hoặc người quản trị xử lý khi nhận được thông báo, dẫn đến thời gian downtime dài hơn.

3.4.3 Độ tin cậy và tính bất đồng bộ trong xử lý

Khi sử dụng SNS kết hợp SQS, thông báo lỗi được xử lý theo mô hình push → queue → process. Điều này có ba lợi ích:

- Không mất cảnh báo khi hệ thống xử lý đang tạm dừng
- Có thể lưu trữ, truy vết toàn bộ lịch sử lỗi
- Cho phép nhiều hệ thống cùng subscribe và phản ứng theo logic riêng

Ngược lại, với mô hình đồng bộ hoặc các công cụ log đơn giản, nếu thời điểm lỗi xảy ra mà hệ thống chưa ghi nhận kịp (ví dụ log collector bị treo), thì rất dễ xảy ra tình trạng mất tín hiệu cảnh báo, đặc biệt trong môi trường đa dịch vụ, tốc độ cao.

3.4.4 Tính linh hoạt, mở rộng và tích hợp

Triển khai trên nền tảng AWS giúp hệ thống có thể mở rộng theo chiều ngang dễ dàng. Với mỗi service mới, chỉ cần bổ sung logic gọi API + gửi SNS mà không cần thay đổi cấu trúc tổng thể. Bên cạnh đó, SNS cho phép mở rộng thêm kênh cảnh báo khác (email, SMS, HTTP endpoint, AWS Lambda, Chatbot...).

Trong môi trường truyền thống, để làm được điều tương tự cần cấu hình thêm server log, dashboard, cảnh báo rule riêng – rất phức tạp và tốn công bảo trì.

3.4.5 Tối ưu chi phí và hiệu quả triển khai

Một điểm thú vị là: dù sử dụng công nghệ cloud, giải pháp của tôi vẫn được mô phỏng hiệu quả trên máy cục bộ nhờ LocalStack. Điều này cho phép kiểm thử logic giống thật nhưng không phát sinh chi phí thực tế. Khi triển khai trên AWS thật, chi phí của SNS và SQS cũng rất thấp nếu lượng cảnh báo không quá lớn. So với việc tự xây dựng hệ thống cảnh báo nội bộ hoặc dùng APM thương mại, đây là một giải pháp chi phí thấp – hiệu quả cao [7], [9].

3.4.6 Tổng hợp đánh giá

Tiêu chí	AWS (SNS + SQS)	Truyền thống / tự xây
Phát hiện lỗi tức thời	Có, real-time qua SNS	Thường trễ (log, cron)
Tự động phục hồi	Dễ tích hợp Lambda, script	Phụ thuộc thủ công
Không mất cảnh báo	Queue buffer đảm bảo dữ liệu	Có thể mất log, miss error
Mở rộng theo chiều ngang	Tốt, dễ cấu hình thêm dịch vụ	Khó scale nếu dùng log

Tích hợp hệ sinh thái AWS	CloudWatch, Lambda, SNS, S3...	Không có hoặc tách rời
Chi phí triển khai ban đầu	Thấp với LocalStack, dễ test	Thường mất công và thời gian

Từ kết quả mô phỏng và phân tích so sánh, có thể khẳng định rằng giải pháp phát hiện và phục hồi lỗi dựa trên nền tảng AWS là phương án hiệu quả, đáng tin cậy và có khả năng áp dụng tốt trong thực tế. So với các phương pháp tự triển khai hoặc giám sát truyền thống, hệ thống sử dụng SNS, SQS và Lambda tỏ ra vượt trội về tốc độ phát hiện, độ tin cậy và khả năng phản ứng tự động. Đây là hướng đi phù hợp cho các hệ thống microservice hiện đại, đặc biệt khi được triển khai trên nền tảng đám mây AWS.

3.5 Kết luận và đóng góp của nghiên cứu

3.5.1 Ứng dụng thực tiễn trong môi trường doanh nghiệp

Giải pháp phát hiện và phục hồi lỗi hệ thống microservice được triển khai trong đồ án không chỉ có ý nghĩa về mặt học thuật, mà còn mang lại giá trị ứng dụng rõ rệt trong thực tế vận hành tại các doanh nghiệp, đặc biệt là các tổ chức đang chuyển đổi sang mô hình hệ thống phân tán, sử dụng dịch vụ đám mây như AWS.

1. Giám sát chủ động và tự phục hồi hệ thống

Trong môi trường doanh nghiệp, việc giám sát hàng chục đến hàng trăm dịch vụ microservice là một thách thức lớn. Việc áp dụng mô hình như đồ án đề xuất – sử dụng SNS để gửi cảnh báo lỗi và SQS để lưu trữ an toàn các thông điệp sự cố – cho phép doanh nghiệp chủ động phát hiện sự cố trong thời gian thực. Kết hợp với khả năng tự động khôi phục (thông qua Lambda, script hoặc các giải pháp container orchestration), doanh nghiệp có thể giảm thời gian downtime, từ đó nâng cao tính sẵn sàng (availability) của hệ thống.

2. Dễ tích hợp vào hệ thống hiện có

Giải pháp được thiết kế độc lập với công nghệ dịch vụ lõi, chỉ yêu cầu tích hợp tại các điểm kiểm tra lỗi (request giữa các service), nên rất dễ áp dụng cho hệ thống đã vận hành sẵn. Ngoài ra, sử dụng các dịch vụ có sẵn trên AWS như SNS, SQS, Lambda giúp doanh nghiệp tận dụng tối đa hạ tầng hiện có, không cần triển khai thêm các công cụ phức tạp hay duy trì server cảnh báo riêng.

3. Tiết kiệm chi phí giám sát và vận hành

So với các hệ thống giám sát chuyên dụng hoặc dịch vụ APM thương mại (Datadog, New Relic...), giải pháp sử dụng SNS/SQS giúp doanh nghiệp giảm chi phí đầu tư ban đầu, đặc biệt phù hợp với các startup hoặc tổ chức quy mô vừa. Mô hình hoạt động pay-as-you-go của AWS cũng giúp kiểm soát chi phí hiệu quả [10].

4. Phù hợp với DevOps và tự động hóa CI/CD

Trong các doanh nghiệp áp dụng DevOps, việc tích hợp hệ thống cảnh báo vào pipeline CI/CD là rất cần thiết. Với SNS, doanh nghiệp có thể dễ dàng kích hoạt cảnh báo khi một bản phát hành mới gây lỗi dịch vụ, từ đó tự động rollback hoặc gửi thông báo đến nhóm vận hành. Việc này góp phần tăng tốc độ triển khai, giảm rủi ro và đảm bảo chất lượng dịch vụ.

5. Nền tảng để mở rộng và nâng cấp

Hệ thống giám sát được xây dựng theo kiến trúc mở, có thể mở rộng để tích hợp thêm các tính năng như dashboard trực quan (Grafana, CloudWatch), giám sát hành vi bất thường (anomaly detection), và thậm chí học máy để phân loại lỗi thông minh hơn. Điều này giúp doanh nghiệp có một nền tảng linh hoạt, dễ nâng cấp và thích nghi với quy mô lớn [11], [12].

Tóm lại, giải pháp phát hiện và xử lý lỗi trong hệ thống microservice mà đồ án đề xuất hoàn toàn có thể áp dụng vào thực tế tại các doanh nghiệp hiện nay. Với ưu điểm về tốc độ phát hiện lỗi, phản ứng tự động, chi phí thấp, khả năng tích hợp cao và dễ mở rộng, hệ thống này là một lựa chọn phù hợp để nâng cao độ tin cậy, giảm thiểu sự cố và tăng hiệu quả vận hành cho hạ tầng microservice hiện đại trong doanh nghiệp.

3.5.2 Khả năng tích hợp với các công nghệ khác

Một trong những điểm mạnh nổi bật của giải pháp phát hiện và phục hồi lỗi dịch vụ được đề xuất trong đồ án là khả năng tích hợp linh hoạt với nhiều công nghệ, nền tảng và công cụ giám sát hiện đại. Việc thiết kế theo mô hình sự kiện (event-driven) kết hợp với các dịch vụ AWS giúp hệ thống dễ dàng mở rộng chức năng và phù hợp với nhiều nhu cầu vận hành thực tế.

1. Tích hợp với các nền tảng giám sát và trực quan hóa

Giải pháp có thể kết hợp dễ dàng với các công cụ giám sát phổ biến như:

- Grafana/Prometheus: Khi tích hợp với các exporter hoặc CloudWatch datasource, hệ thống có thể trực quan hóa trạng thái lỗi, tỷ lệ sự cố, thời gian khôi phục trung bình (MTTR).
- ELK Stack (Elasticsearch – Logstash – Kibana): Có thể thu thập log từ các container dịch vụ để phân tích nguyên nhân lỗi, xây dựng dashboard theo thời gian thực [6].

2. Tích hợp với hệ thống CI/CD và DevOps

Trong môi trường DevOps, việc tự động phát hiện lỗi sau khi triển khai ứng dụng là điều tối quan trọng. Giải pháp trong đồ án có thể tích hợp với:

- GitHub Actions / GitLab CI / Jenkins: Nếu xảy ra lỗi sau khi deploy, SNS có thể kích hoạt job rollback hoặc gửi cảnh báo trực tiếp đến người phát hành [13][14].
- Terraform / AWS CDK / Pulumi: Cho phép quản lý hạ tầng dịch vụ giám sát dưới dạng mã (IaC), đảm bảo tính nhất quán giữa môi trường phát triển và sản xuất [7].

3. Tích hợp với các công nghệ container và điều phối dịch vụ

- Kubernetes (EKS): Với các cụm dịch vụ chạy trên EKS, hệ thống có thể bổ sung sidecar hoặc init container để theo dõi liveness, kết hợp với SNS để phát hiện lỗi [11].
- Docker Swarm: Có thể dùng health check + log monitoring + SNS cảnh báo để phục hồi container trong môi trường nhẹ.

4. Tích hợp với các hệ thống cảnh báo và giao tiếp nội bộ

- Email, SMS: SNS có thể cấu hình gửi trực tiếp email hoặc tin nhắn SMS cho đội ngũ vận hành khi có sự cố nghiêm trọng.
- Slack / Microsoft Teams / Telegram: Thông qua webhook hoặc Lambda tích hợp, cảnh báo lỗi có thể được gửi về kênh làm việc của nhóm kỹ thuật theo thời gian thực [12].
- ServiceNow / Jira / Opsgenie: Có thể tạo ticket tự động khi có lỗi nhằm ghi nhận và phản hồi theo quy trình ITSM chuẩn [25].

5. Tích hợp với các nền tảng học máy và phân tích dữ liệu

Hệ thống hoàn toàn có thể tích hợp với:

- Amazon SageMaker hoặc nền tảng ML nội bộ để phân tích hành vi lỗi, phát hiện bất thường (anomaly detection).
- Athena, Redshift, S3 để lưu trữ và phân tích lịch sử lỗi theo thời gian dài.

Giải pháp giám sát và phục hồi lỗi dịch vụ được đề xuất trong đồ án không chỉ hoạt động độc lập hiệu quả, mà còn có khả năng mở rộng và tích hợp sâu với hầu hết các công nghệ phổ biến trong vận hành hệ thống hiện đại. Điều này cho phép hệ thống không bị giới hạn trong một môi trường cụ thể, mà có thể triển khai linh hoạt, mở rộng chức năng và tích hợp dễ dàng vào hệ sinh thái công nghệ mà doanh nghiệp đang sử dụng.

3.5.3 Giới hạn của mô hình

Mặc dù giải pháp phát hiện và phục hồi lỗi dịch vụ microservice được đề xuất trong đồ án đã đạt được những kết quả khả quan về hiệu quả và khả năng ứng dụng thực tiễn, tuy nhiên mô hình vẫn còn tồn tại một số giới hạn nhất định cần được thừa nhận và khắc phục trong các giai đoạn phát triển tiếp theo.

1. Giới hạn về phạm vi lỗi được mô phỏng

Mô hình hiện tại mới chỉ mô phỏng ba loại lỗi phổ biến: dịch vụ bị dừng đột ngột, phản hồi lỗi HTTP và lỗi timeout. Tuy đây là các tình huống cơ bản và thường gặp, nhưng trong thực tế sản xuất, hệ thống có thể gặp nhiều loại lỗi phức tạp hơn như:

- Lỗi do cạn kiệt tài nguyên (RAM, CPU)
- Lỗi giao tiếp bất đồng bộ (message queue quá tải, dead-letter queue)
- Lỗi liên quan đến database (kết nối timeout, lỗi khóa, lỗi transaction)
- Lỗi bảo mật (token hết hạn, quyền truy cập sai)

Việc mô phỏng và xử lý các lỗi này cần các cơ chế quan sát sâu hơn như tracing, profiling, và correlation ID – hiện chưa được tích hợp trong mô hình.

2. Khả năng phản ứng còn đơn giản

Trong mô hình hiện tại, việc phản ứng với lỗi (ví dụ khởi động lại service) chủ yếu được thực hiện thông qua một script Python đơn giản hoặc Lambda một chiều. Hệ thống chưa hỗ trợ các hành động nâng cao như:

- Phân tích nguyên nhân lỗi (root cause analysis)
- Thử nhiều phương án khôi phục khác nhau (retry, rollback)
- Gửi cảnh báo có điều kiện theo ngữ cảnh (ví dụ chỉ gửi khi lỗi vượt ngưỡng)
- Quản lý trạng thái dịch vụ toàn cục (service dependency graph)

Điều này có thể dẫn đến phản ứng thiếu tối ưu trong một số tình huống sản xuất phức tạp.

3. Thiếu giao diện trực quan để theo dõi trạng thái hệ thống

Toàn bộ mô hình hiện tại hoạt động dựa trên log dòng lệnh và console output. Hệ thống chưa tích hợp các công cụ trực quan như dashboard, biểu đồ lỗi, báo cáo thống kê theo thời gian thực. Điều này gây khó khăn trong việc đánh giá xu hướng lỗi, xác định dịch vụ bất ổn định hoặc tổng hợp báo cáo cho quản trị viên.

4. Phụ thuộc vào hạ tầng AWS

Mặc dù sử dụng AWS là một ưu điểm về mặt tích hợp, nhưng đồng thời cũng là một giới hạn nếu doanh nghiệp không sử dụng AWS hoặc chuyển sang nền tảng khác như GCP, Azure hoặc on-premise. Việc tái sử dụng mô hình cần điều chỉnh tương ứng với các dịch vụ thay thế như Google Pub/Sub, Azure Service Bus...

Thông qua việc xây dựng và thử nghiệm mô hình phát hiện và khắc phục lỗi trong hệ thống microservices, đồ án đã đạt được một số kết luận quan trọng và đóng góp có ý nghĩa về mặt kỹ thuật cũng như thực tiễn.

3.5.4 Khuyến nghị và định hướng phát triển

Dựa trên những kết quả đạt được, cũng như các giới hạn đã nêu, phần này đề xuất một số khuyến nghị và định hướng phát triển để hoàn thiện và mở rộng hệ thống phát hiện và phục hồi lỗi microservice trong tương lai. Các định hướng này không chỉ mang ý nghĩa kỹ thuật mà còn mang giá trị ứng dụng cao trong môi trường doanh nghiệp và nghiên cứu khoa học.

1. Phát triển dashboard giám sát trực quan

Để tăng tính tiện dụng cho người vận hành, hệ thống nên được tích hợp thêm một dashboard giao diện web, hiển thị các thông tin như:

- Trạng thái hoạt động của từng dịch vụ
- Lịch sử lỗi và thống kê lỗi theo thời gian
- Các cảnh báo đang chờ xử lý
- Hiệu quả phản ứng lỗi (tỷ lệ thành công, thời gian xử lý)

Việc này có thể thực hiện thông qua tích hợp với Grafana, CloudWatch Dashboard, hoặc xây dựng frontend riêng sử dụng React, Angular kết hợp API.

2. Hỗ trợ thêm nhiều loại lỗi thực tế hơn

Mô hình cần được mở rộng để phát hiện các loại lỗi nâng cao trong môi trường thực tế như:

- Lỗi tắc nghẽn message queue (kết hợp với Amazon MQ hoặc Kafka)
- Lỗi tài nguyên (CPU, RAM cao bất thường – từ CloudWatch Metrics)
- Lỗi truy cập không hợp lệ (403, 401 – cần tích hợp APM hoặc logging middleware)
- Lỗi giao tiếp bất đồng bộ giữa các service

Từ đó xây dựng bộ quy tắc cảnh báo phong phú, tùy chỉnh theo đặc điểm hệ thống.

3. Nâng cấp cơ chế phản ứng lỗi thông minh

Thay vì phản ứng một chiều (ví dụ luôn restart container), hệ thống nên hỗ trợ các hành động đa dạng hơn:

- Tự động retry một số lần trước khi khởi động lại

- Gửi cảnh báo có điều kiện (ví dụ lỗi 3 lần trong 5 phút)
- Kết hợp với workflow phục hồi (AWS Step Functions hoặc custom FSM)
- Tạm thời redirect request đến phiên bản backup nếu có

Ngoài ra, có thể áp dụng mô hình policy-driven reaction – nơi hành động được định nghĩa theo cấu hình, thay vì mã cứng.

4. Áp dụng machine learning để phát hiện bất thường

Một hướng phát triển tiềm năng là tích hợp khả năng học máy để phát hiện lỗi theo hành vi. Ví dụ:

- Sử dụng mô hình anomaly detection để xác định thời điểm phản hồi bất thường
- Phân cụm lỗi theo loại, mức độ, tần suất xuất hiện
- Dự đoán dịch vụ có nguy cơ gặp lỗi cao dựa trên lịch sử

Các mô hình có thể được triển khai thông qua Amazon SageMaker, hoặc thư viện nội bộ như Scikit-learn, PyTorch...

5. Triển khai production-ready trên Kubernetes

Hệ thống hiện tại chủ yếu mô phỏng qua Docker Compose. Để phù hợp với môi trường thực tế doanh nghiệp, cần triển khai lên Kubernetes (EKS) với các thành phần như:

- Health check readiness/liveness
- Horizontal pod autoscaler
- Pod restart và circuit breaker
- Alert routing bằng Prometheus Operator + Alertmanager

Việc này đảm bảo giải pháp có thể hoạt động ổn định, mở rộng theo nhu cầu và tích hợp với các hệ thống CI/CD hiện đại.

6. Chuẩn hóa theo mô hình giám sát tập trung

Cuối cùng, hệ thống nên được chuẩn hóa theo kiến trúc giám sát hiện đại, với các lớp chức năng như:

- Thu thập dữ liệu (log/metric/tracing)
- Phát hiện lỗi (dựa trên rule hoặc ML)

- Gửi cảnh báo (SNS, Email, Slack...)
- Phản ứng tự động (Lambda, script, orchestration)
- Dashboard & audit log (giám sát và kiểm tra sau)

Điều này giúp dễ dàng bảo trì, mở rộng và đào tạo đội ngũ sử dụng.

Giải pháp được trình bày trong đồ án đã chứng minh được hiệu quả bước đầu trong việc phát hiện và xử lý lỗi trong hệ thống microservice. Tuy nhiên, để đưa vào ứng dụng ở quy mô doanh nghiệp lớn, cần tiếp tục cải tiến ở cả chiều sâu (nhiều loại lỗi hơn, phản ứng linh hoạt hơn) và chiều rộng (tích hợp dễ dàng, tự động hóa cao hơn). Những định hướng trên sẽ là nền tảng vững chắc để phát triển hệ thống giám sát – phục hồi thông minh, linh hoạt, và có khả năng học hỏi trong tương lai.

3.5.5 Tổng kết chương

Trong bối cảnh các hệ thống phần mềm hiện đại ngày càng phát triển theo hướng phân tán và phi tập trung, kiến trúc microservice đang trở thành xu hướng chủ đạo để xây dựng các hệ thống linh hoạt, dễ mở rộng và triển khai theo từng chức năng riêng biệt. Tuy nhiên, cùng với lợi ích mà mô hình microservice mang lại cũng xuất hiện một loạt các thách thức, đặc biệt là vấn đề giám sát, phát hiện và xử lý lỗi dịch vụ một cách kịp thời và chính xác. Lỗi ở một dịch vụ đơn lẻ có thể ảnh hưởng dây chuyền đến toàn bộ hệ thống, gây mất ổn định và ảnh hưởng đến chất lượng dịch vụ.

Xuất phát từ thực tế này, đồ án đã tập trung nghiên cứu và triển khai một giải pháp phát hiện và phục hồi lỗi trong hệ thống microservice dựa trên nền tảng AWS, trong đó sử dụng kết hợp các dịch vụ chủ chốt như SNS (Simple Notification Service) để gửi cảnh báo lỗi, SQS (Simple Queue Service) để lưu trữ và phân phối thông điệp, cùng với một thành phần phản ứng tự động sử dụng script Python hoặc AWS Lambda. Điểm đặc biệt của giải pháp là khả năng mô phỏng và kiểm thử hiệu quả ngay trên môi trường cục bộ thông qua LocalStack, từ đó dễ dàng chuyển đổi sang môi trường triển khai thực tế trên hạ tầng AWS mà không cần thay đổi kiến trúc hoặc logic xử lý.

Đồ án đã mô phỏng thành công ba loại lỗi phổ biến trong hệ thống microservice, bao gồm: (1) lỗi dịch vụ bị dừng đột ngột, (2) lỗi phản hồi HTTP bất

thường (500/503), và (3) lỗi timeout phản hồi do trì hoãn xử lý. Kết quả thực nghiệm cho thấy hệ thống có thể phát hiện lỗi với độ chính xác cao, cảnh báo kịp thời và đưa ra phản ứng linh hoạt, phù hợp với từng loại lỗi. Đặc biệt, hệ thống thể hiện năng lực tự phục hồi (self-healing) ban đầu khi có thể tự động khởi động lại dịch vụ bị dừng mà không cần sự can thiệp của con người. So sánh với các phương pháp giám sát truyền thống hoặc công cụ giám sát tập trung như Prometheus, ELK hay hệ thống log thủ công, mô hình trong đồ án cho thấy nhiều lợi thế về độ tin cậy, tính bất đồng bộ, khả năng mở rộng và chi phí triển khai thấp.

Bên cạnh những ưu điểm nổi bật, đồ án cũng thẳng thắn nhìn nhận những giới hạn hiện tại của hệ thống, bao gồm: phạm vi mô phỏng lỗi còn chưa toàn diện, cơ chế phản ứng lỗi còn đơn giản, thiếu giao diện người dùng trực quan, và chưa thử nghiệm trên hệ thống có quy mô lớn. Tuy nhiên, các định hướng phát triển tiếp theo đã được đề xuất cụ thể, như mở rộng mô hình phát hiện lỗi theo thời gian thực, tích hợp học máy để phát hiện bất thường, sử dụng dashboard trực quan hóa, và triển khai hệ thống production-ready trên Kubernetes hoặc EKS.

Từ góc độ học thuật, đồ án không chỉ đưa ra một giải pháp kỹ thuật cụ thể mà còn đề xuất một mô hình tư duy tổng thể về cách phát hiện – cảnh báo – phản ứng – khôi phục lỗi trong hệ thống microservice theo hướng event-driven – self-healing – cloud-native, phù hợp với các xu hướng vận hành hệ thống hiện đại như DevOps, GitOps, hoặc Serverless Monitoring. Giải pháp không bị giới hạn trong phạm vi lý thuyết mà có tính ứng dụng cao trong doanh nghiệp, đặc biệt là các tổ chức đang triển khai hệ thống trên nền tảng AWS.

Tổng kết lại, đồ án đã hoàn thành các mục tiêu đề ra, cụ thể:

- Phân tích đặc trưng của lỗi trong hệ thống microservice
- Xây dựng được mô hình phát hiện và phục hồi lỗi hiệu quả
- Mô phỏng, kiểm thử và đánh giá hệ thống trên cả môi trường giả lập và thực tế
- So sánh ưu điểm – hạn chế của giải pháp trên AWS với các phương pháp khác

- Đề xuất các hướng phát triển để mở rộng mô hình trong tương lai

Với các nội dung đã thực hiện, đồ án không chỉ mang lại một giải pháp kỹ thuật khả thi, mà còn đóng góp một cách tiếp cận mới, linh hoạt và dễ triển khai cho bài toán phát hiện và phục hồi lỗi trong môi trường microservice. Đây có thể xem là nền tảng ban đầu để tiếp tục phát triển các hệ thống giám sát tự động, thông minh và bền vững hơn trong thực tiễn công nghiệp cũng như trong nghiên cứu học thuật.

KẾT LUẬN

Đề án đã xây dựng thành công một mô hình phát hiện và phục hồi lỗi cho hệ thống microservice dựa trên nền tảng AWS, sử dụng kết hợp các dịch vụ SNS, SQS và thành phần phản ứng tự động để xử lý ba loại lỗi phổ biến: sự cố gián đoạn hạ tầng, lỗi phản hồi phía ứng dụng và lỗi vượt ngưỡng thời gian phản hồi. Hệ thống được triển khai thử nghiệm trên môi trường giả lập LocalStack và có khả năng triển khai thực tế trên AWS với chi phí thấp và cấu hình linh hoạt.

Thông qua quá trình mô phỏng, hệ thống cho thấy khả năng phát hiện lỗi chính xác, cảnh báo kịp thời và phục hồi hiệu quả trong các tình huống cụ thể. So với các phương pháp giám sát truyền thống, giải pháp này có ưu điểm về khả năng mở rộng, độ tin cậy và dễ tích hợp vào quy trình DevOps hiện đại. Bên cạnh đó, hệ thống cũng được thiết kế theo kiến trúc mở, cho phép phát triển thêm các tính năng như dashboard trực quan, phân tích lỗi nâng cao, hoặc tích hợp với các nền tảng giám sát chuyên nghiệp.

Kết quả đạt được không chỉ đáp ứng mục tiêu nghiên cứu đã đề ra, mà còn mở ra hướng phát triển thực tiễn cho các hệ thống giám sát thông minh, hỗ trợ vận hành hiệu quả các nền tảng microservice trong doanh nghiệp.

DANH MỤC TÀI LIỆU THAM KHẢO

[1] Amazon, Amazon CloudWatch Documentation. https://docs.aws.amazon.com/cloudwatch/. Truy cập tháng 4/2024.

[2] Amazon, Amazon ECS Developer Guide. https://docs.aws.amazon.com/ecs/latest/developerguide/. Truy cập tháng 4/2024.

[3] Amazon, Amazon S3 Documentation. https://docs.aws.amazon.com/s3/. Truy cập tháng 4/2024.

[4] Amazon, Amazon SageMaker Developer Guide. https://docs.aws.amazon.com/sagemaker/latest/dg/. Truy cập tháng 4/2024.

[5] Amazon, Amazon Simple Notification Service (SNS) Documentation. https://docs.aws.amazon.com/sns/. Truy cập tháng 4/2024.

[6] Amazon, Amazon SQS Developer Guide. https://docs.aws.amazon.com/sqs/latest/dg/. Truy cập tháng 4/2024.

[7] Amazon, AWS CDK Documentation. https://docs.aws.amazon.com/cdk/. Truy cập tháng 4/2024.

[8] Amazon, AWS Lambda Developer Guide. https://docs.aws.amazon.com/lambda/. Truy cập tháng 4/2024.

[9] Amazon, AWS Step Functions Developer Guide. https://docs.aws.amazon.com/step-functions/. Truy cập tháng 4/2024.

[10] Amazon, AWS Systems Manager Documentation. https://docs.aws.amazon.com/systems-manager/. Truy cập tháng 4/2024.

[11] Amazon, EKS Documentation. https://docs.aws.amazon.com/eks/. Truy cập tháng 4/2024.

[12] Elastic, ELK Stack Overview (Elasticsearch, Logstash, Kibana). https://www.elastic.co/what-is/elk-stack. Truy cập tháng 4/2024.

[13] GitHub, GitHub Actions Documentation. https://docs.github.com/en/actions. Truy cập tháng 4/2024.

[14] GitLab, GitLab CI/CD Documentation. https://docs.gitlab.com/ee/ci/. Truy cập tháng 4/2024.

[15] HashiCorp, Terraform Documentation. https://developer.hashicorp.com/terraform/docs. Truy cập tháng 4/2024.

[16] Jenkins, Jenkins User Documentation. https://www.jenkins.io/doc/. Truy cập tháng 4/2024.

[17] Martin Fowler, Microservices. https://martinfowler.com/articles/microservices.html. Truy cập tháng 4/2024.

[18] Microsoft, Azure Monitor Documentation. https://learn.microsoft.com/en-us/azure/azure-monitor/. Truy cập tháng 4/2024.

[19] Opsgenie, Opsgenie Documentation. https://docs.opsgenie.com/. Truy cập tháng 4/2024.

[20] Pulumi, Pulumi Documentation. https://www.pulumi.com/docs/. Truy cập tháng 4/2024.

[21] Red Hat, Introduction to Microservices. https://www.redhat.com/en/topics/microservices. Truy cập tháng 4/2024.

[22] Sam Newman, Building Microservices: Designing Fine-Grained Systems. O'Reilly Media, 2015.

[23] ServiceNow, ServiceNow Documentation. https://docs.servicenow.com/. Truy cập tháng 4/2024.

[24] Telegram, Telegram Bots Documentation. https://core.telegram.org/bots. Truy cập tháng 4/2024.

[25] Trello, Jira Service Management Documentation. https://support.atlassian.com/jira-service-management/. Truy cập tháng 4/2024.

BẢN CAM ĐOAN

Tôi cam đoan đã thực hiện việc kiểm tra mức độ tương đồng nội dung đề án qua phần mềm https://kiemtratailieu.vn một cách trung thực và đạt kết quả mức độ tương đồng 2% toàn bộ nội dung đề án. Đề án kiểm tra qua phần mềm là bản cứng đề án đã nộp để bảo vệ trước hội đồng. Nếu sai tôi xin chịu các hình thức kỷ luật theo quy định hiện hành của Học viện.

Hà Nội, ngày 31 tháng 7 năm 2025

HỌC VIÊN CAO HỌC

(Ký và ghi rõ họ tên)

Phong

Phạm Tuấn Phong

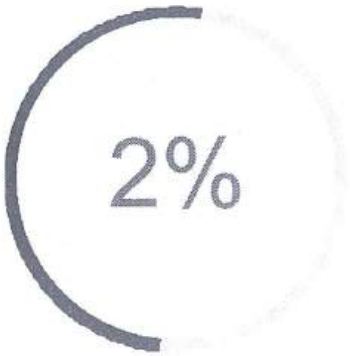
KiểmTraTàiLiệu

BÁO CÁO KIỂM TRA TRÙNG LẶP

Thông tin tài liệu

Tên tài liệu:	PhamTuanPhong_De an tot nghiep cao hoc
Tác giả:	Phạm Tuấn Phong
Điểm trùng lặp:	2
Thời gian tải lên:	04:15 31/07/2025
Thời gian sinh báo cáo:	04:18 31/07/2025
Các trang kiểm tra:	81/81 trang

Kết quả kiểm tra trùng lặp



Có 2% nội dung trùng lặp

Có 98% nội dung không trùng lặp

Có 0% nội dung người dùng loại trừ

Có 0% nội dung hệ thống bỏ qua

Nguồn trùng lặp tiêu biểu

123docz.net tailieu.vn 200lab.io

Học viên
(Ký và ghi rõ họ tên)

Phạm Tuấn Phong

Người hướng dẫn khoa học
(Ký và ghi rõ họ tên)

BỘ KHOA HỌC VÀ CÔNG NGHỆ
HỌC VIỆN CÔNG NGHỆ BƯU CHÍNH VIỄN THÔNG

CỘNG HOÀ XÃ HỘI CHỦ NGHĨA VIỆT NAM
Độc lập - Tự do - Hạnh phúc

BÁO CÁO GIẢI TRÌNH
SỬA CHỮA, HOÀN THIỆN ĐỀ ÁN TỐT NGHIỆP

Họ và tên học viên: Phạm Tuấn Phong

Chuyên ngành: HTTT

Khóa: 2023 đợt 2

Tên đề tài: PHÁT HIỆN VÀ KHẮC PHỤC LỖI TRONG HỆ THỐNG MICROSERVICES
Người hướng dẫn khoa học: TS. Nguyễn Duy Phương

Ngày bảo vệ: 19/07/2025

Các nội dung học viên đã sửa chữa, bổ sung trong đề án tốt nghiệp theo ý kiến đóng góp của Hội đồng chấm đề án tốt nghiệp:

TT	Ý kiến hội đồng	Sửa chữa của học viên
1	Chỉnh sửa lỗi soạn thảo, lỗi ngữ pháp, chính tả	Học viên đã rà soát, chỉnh sửa các lỗi soạn thảo, các lỗi ngữ pháp
2	Chỉnh sửa lại bố cục	Tiếp thu góp ý của Hội đồng, tác giả đã chỉnh sửa lại bố cục đề án, tách phần mô phỏng ở chương 2 sang chương 3
3	Bổ sung phân tích thiết kế và yêu cầu hệ thống	Tiếp thu góp ý của Hội đồng, tác giả đã sửa và bổ sung thêm phân tích thiết kế, yêu cầu ở chương 2
4	Hình vẽ đánh theo chương mục	Tiếp thu góp ý của Hội đồng, tác giả đã đánh số hình ảnh theo chương
5	Bổ sung tài liệu tham khảo và trích dẫn vào đề án	Tiếp thu góp ý của Hội đồng, tác giả đã bổ sung tài liệu tham khảo và trích dẫn trong đề án
6	Hạn chế sử dụng ngôi thứ nhất	Tiếp thu góp ý của Hội đồng, tác giả đã sửa hoàn toàn đoạn văn sử dụng ngôi thứ nhất trong đề án
7	Hình vẽ nên dịch sang tiếng Việt	Tiếp thu góp ý của Hội đồng, tác giả đã dịch một số hình ảnh sang tiếng Việt

Hà Nội, ngày 31 tháng 7 năm 2025

Ký xác nhận của

CHỦ TỊCH HỘI ĐỒNG CHẤM ĐỀ ÁN	THƯ KÝ HỘI ĐỒNG	NGƯỜI HƯỚNG DẪN KHOA HỌC	HỌC VIÊN
PGS.TS. Lê Hữu Lập	TS. Đinh Trường Duy	TS. Nguyễn Duy Phương	Phạm Tuấn Phong

BỘ KHOA HỌC VÀ CÔNG NGHỆ
HỌC VIỆN CÔNG NGHỆ BƯU CHÍNH VIỄN THÔNG

CỘNG HOÀ XÃ HỘI CHỦ NGHĨA VIỆT NAM
Độc lập - Tự do - Hạnh phúc

BIÊN BẢN
HỌP HỘI ĐỒNG CHẤM ĐỀ ÁN TỐT NGHIỆP THẠC SĨ

Căn cứ quyết định số Quyết định số 1098/QĐ-HV ngày 26 tháng 06 năm 2025 của Giám đốc Học viện Công nghệ Bưu chính Viễn thông về việc thành lập Hội đồng chấm đề án tốt nghiệp thạc sĩ. Hội đồng đã họp vào hồi....9....giờ.....55.....phút, ngày 19 tháng 07 năm 2025 tại Học viện Công nghệ Bưu chính Viễn thông để chấm đề án tốt nghiệp thạc sĩ cho:

Học viên: **Phạm Tuấn Phong**

Tên đề án tốt nghiệp: **Phát hiện và khắc phục lỗi trong hệ thống MICROSERVIES**

Chuyên ngành: **Hệ thống thông tin** Mã số: **8480104**

Các thành viên của Hội đồng chấm đề án tốt nghiệp có mặt: .....5...../ 05

TT	HỌ VÀ TÊN	TRÁCH NHIỆM TRONG HĐ	GHI CHÚ
1	**PGS.TS. Lê Hữu Lập**	Chủ tịch	
2	**TS. Đinh Trường Duy**	Thư ký	
3	**PGS.TS. Phạm Văn Cường**	Phản biện 1	
4	**TS. Nguyễn Chí Thành**	Phản biện 2	
5	**PGS.TS. Nguyễn Long Giang**	Uỷ viên	

Các nội dung thực hiện:

1. Chủ tịch Hội đồng điều khiển buổi họp. Công bố quyết định của Giám đốc Học viện Công nghệ Bưu chính Viễn thông về việc thành lập Hội đồng chấm đề án tốt nghiệp thạc sĩ.
2. Người hướng dẫn khoa học hoặc thư ký đọc lý lịch khoa học và các điều kiện bảo vệ đề án tốt nghiệp của học viên. (có bản lý lịch khoa học và kết quả các môn học cao học của học viên kèm theo).
3. Học viên trình bày tóm tắt đề án tốt nghiệp.
4. Phản biện 1 đọc nhận xét (có văn bản kèm theo)
5. Phản biện 2 đọc nhận xét (có văn bản kèm theo)
6. Các câu hỏi của thành viên Hội đồng:

Hệ thống phát hiện và khắc phục lỗi AWS được đánh giá như thế nào?
Giải thích những khác biệt và thách thức khi chuyển từ môi trường giả lập sang môi trường AWS thực tế

7. Trả lời của học viên:

Học viên thuyết trình và trả lời câu hỏi của Hội đồng
Học viên tiếp thu và sẽ chỉnh sửa theo ý kiến của
Hội đồng

8. Thư ký đọc nhận xét về quá trình thực hiện đề án tốt nghiệp của học viên *(có văn bản kèm theo).*

9. Hội đồng họp riêng:

- Bầu Ban kiểm phiếu:
  1. Trưởng Ban kiểm phiếu: PGS. TS. Phạm Văn Chung
  2. Uỷ viên Ban kiểm phiếu: TS. Đinh Trường Duy
  3. Uỷ viên Ban kiểm phiếu: PGS. TS. Nguyễn Long Giang
- Hội đồng chấm đề án tốt nghiệp bằng bỏ phiếu kín.
- Ban kiểm phiếu làm việc:
- Trưởng Ban kiểm phiếu báo cáo kết quả kiểm phiếu *(có Biên bản họp Ban kiểm phiếu kèm theo)*
- **Điểm trung bình của đề án tốt nghiệp:** 8,0

Kết luận:

1. Các nội dung cần chỉnh sửa, hoàn thiện sau bảo vệ đề án tốt nghiệp:

- Chỉnh sửa theo ý kiến của phản biện
- Chỉnh sửa lại tên mục: tính phần thử nghiệm Chương 2 sang chương 3
- Chương 2 bổ sung thêm phần tính toán thiết kế, yêu cầu của bể thông
- Hình vẽ đánh theo chương mục
- Bổ sung tài liệu tham khảo và trích dẫn vào đề án

2. Đề nghị Học viện công nhận (hoặc không) và cấp bằng (hoặc không) thạc sĩ cho học viên: Phạm Thắng Phong

3. Đề án tốt nghiệp có thể phát triển thành đề tài nghiên cứu cho NCS.....

Buổi làm việc kết thúc vào 10g40 cùng ngày.

Chủ tịch	**Thư ký**
PGS.TS. Lê Hữu Lập	TS. Đinh Trường Duy

CỘNG HÒA XÃ HỘI CHỦ NGHĨA VIỆT NAM

Độc lập – Tự do – Hạnh phúc

BẢN NHẬN XÉT ĐỀ ÁN TỐT NGHIỆP THẠC SĨ

(Dùng cho người phản biện)

Tên đề tài đề án tốt nghiệp: ***Phát hiện và khắc phục lỗi trong hệ thống Microservices***

Chuyên ngành: Hệ thống thông tin

Mã chuyên ngành: 8.48.01.04

Họ và tên học viên: ~~Nguyễn Ngọc Quý~~ Phạm Tuấn Phong

Họ và tên người nhận xét: Nguyễn Chí Thành

Học hàm, học vị: Tiến sĩ

Chuyên ngành: Công nghệ thông tin

Cơ quan công tác: Viện Công nghệ thông tin - Điện tử, Viện KH-CN quân sự

Số điện thoại: 0914625034 E-mail: thanhnc80@gmail.com

NỘI DUNG NHẬN XÉT

I/ Cơ sở khoa học và thực tiễn, tính cấp thiết của đề tài:

Trong bối cảnh chuyển đổi số, kiến trúc microservices đã trở thành tiêu chuẩn cho việc phát triển các hệ thống phần mềm quy mô lớn, linh hoạt. Tuy nhiên, bản chất phân tán của kiến trúc này lại làm cho bài toán giám sát, phát hiện và xử lý lỗi trở nên phức tạp và cấp thiết hơn bao giờ hết. Một sự cố nhỏ tại một dịch vụ có thể gây ra hiệu ứng dây chuyền, làm gián đoạn toàn bộ hệ thống, ảnh hưởng trực tiếp đến chất lượng dịch vụ và trải nghiệm người dùng. Các giải pháp giám sát truyền thống thường đòi hỏi cấu hình phức tạp và chi phí cao.

Đề án tập trung nghiên cứu và xây dựng một giải pháp phát hiện và tự động khắc phục lỗi bằng cách tận dụng các dịch vụ đám mây linh hoạt của Amazon Web Services (AWS). Đây là một đề tài có tính thời sự, giải quyết một bài toán kỹ thuật quan trọng và có giá trị thực tiễn đối với các doanh nghiệp.

II/ Nội dung của đề án tốt nghiệp, các kết quả đã đạt được:

Nội dung đề án tập trung vào việc thiết kế, triển khai và đánh giá một hệ thống có khả năng phát hiện và tự động phục hồi sau lỗi trong kiến trúc microservices. Đề án đã trình bày tổng quan chi tiết về kiến trúc microservices, các loại lỗi thường gặp, và vai trò của nền tảng AWS trong việc giám sát hệ thống. Phần thực nghiệm của đề án được xây dựng với một hệ thống mô phỏng gồm hai dịch vụ, sử dụng Docker và Spring Boot. Tác giả đã thiết kế và triển khai các quy trình (pipeline) xử lý cho ba loại lỗi cụ thể: sự cố gián đoạn hạ tầng (service bị dừng), lỗi phản hồi phía ứng dụng (HTTP 5xx), và lỗi vượt ngưỡng thời gian phản hồi (timeout). Kết quả thực nghiệm

cho thấy hệ thống hoạt động hiệu quả: phát hiện lỗi một cách chính xác, gửi cảnh báo kịp thời qua SNS và SQS, đồng thời thực hiện hành động khắc phục một cách thông minh và tự động (tự khởi động lại dịch vụ khi cần thiết).

III/ Những vấn đề cần giải thích thêm:

Trong Chương 1 và các chương khác, đề án trình bày nhiều kiến thức và thông tin nền tảng nhưng còn thiếu các trích dẫn khoa học tại các luận điểm. Tác giả cần bổ sung để tăng tính khoa học cho công trình.

Môi trường thử nghiệm sử dụng LocalStack là một phương pháp tiếp cận rất hiệu quả. Tuy nhiên, tác giả cần phân tích sâu hơn về những khác biệt và thách thức tiềm tàng khi chuyển đổi từ môi trường giả lập sang môi trường AWS thực tế.

Phạm vi lỗi được mô phỏng tập trung vào ba loại phổ biến. Tác giả cần giải thích thêm về khả năng mở rộng của mô hình để xử lý các loại lỗi phức tạp khác như cạn kiệt tài nguyên (CPU/RAM), lỗi kết nối cơ sở dữ liệu, hoặc lỗi trong hàng đợi tin nhắn (message queue).

Cơ chế phản ứng lỗi hiện tại còn đơn giản (chủ yếu là khởi động lại dịch vụ). Cần giải thích tại sao các mẫu thiết kế phục hồi nâng cao hơn như Circuit Breaker hay Exponential Backoff không được tích hợp và phân tích trong giải pháp.

Đây là đề án theo định hướng ứng dụng, tuy nhiên lại thiếu một giao diện giám sát trực quan (dashboard). Tác giả cần phân tích sâu hơn về tầm quan trọng của việc trực quan hóa dữ liệu lỗi đối với đội ngũ vận hành và đề xuất kiến trúc tích hợp với các công cụ như Grafana hoặc CloudWatch Dashboard.

Cần rà soát lại toàn bộ đề án về mặt trình bày, lỗi chính tả để đảm bảo tính chuyên nghiệp. Danh mục các ký hiệu, chữ viết tắt cần sắp xếp theo thứ tự ABC. Chương 3 và phần Kết luận cùng trình bày về kết luận là không hợp lý. Đề án cần hạn chế sử dụng ngôi thứ nhất. Các hình vẽ cần dịch sang tiếng Việt.

IV/ Kết luận:

Tuy còn một số nội dung cần bổ sung, chỉnh sửa để nâng cao chất lượng, nhưng đề án đã giải quyết được mục đích và nhiệm vụ đề ra. Tôi đồng ý để học viên được bảo vệ đề án tốt nghiệp.

Ngày 15 tháng 7 năm 2025

NGƯỜI NHẬN XÉT

Nguyễn Chí Thành

BỘ KHOA HỌC VÀ CÔNG NGHỆ **CỘNG HÒA XÃ HỘI CHỦ NGHĨA VIỆT NAM**

HỌC VIỆN CÔNG NGHỆ BCVT **Độc lập – Tự do – Hạnh phúc**

---------o0o--------

NHẬN XÉT CỦA NGƯỜI PHẢN BIỆN

Tên đề tài: Phát hiện và khắc phục lỗi trong hệ thống Microservices

Chuyên ngành: Hệ thống thông tin.....

Mã chuyên ngành:8.48.01.04 .....

Tên học viên: Phạm Tuấn Phong .....

Họ và tên người nhận xét: Phạm Văn Cường.....

Học hàm, học vị: PGS.TS.....

Chuyên ngành: Khoa học Máy tính

Nơi công tác: Học viện Công nghệ Bưu chính Viễn thông .....

NỘI DUNG NHẬN XÉT

I/ Cơ sở khoa học và thực tiễn, tính cấp thiết của đề tài:

Có ý nghĩa thực tế cho các hệ thống, nền tảng đa dịch vụ, hệ tống Web services.

II/ Về nội dung, chất lượng của luận văn, kết quả đạt được:

- Luận văn được chia thành 3 chương với 68 trang. Các chương được cấu trúc phù hợp với đề cương; Chương 1 trình bầy tổng quan về các lỗi trong hệ thống microservices, AWS, phương pháp phát hiện lỗi và cách khắc phục; Chương 2 trình bầy về xây dựng hệ thống phát hiện và khắc phục lỗi trên AWS; chương 3 trình bầy về thử nghiệm và đánh giá hệ thống.

III/ Những vấn đề cần giải thích thêm:

Chương 2 viết còn khá chung chung; nên viết theo hướng xây dựng một hệ thống, bao gồm phân tích các yêu cầu hệ thống, phân tích và thiết kế hệ thống. Cần giải thích rõ ý nghĩa và thứ thự ưu tiên khắc phục lỗi. Chương 3 cần bổ sung trình bầy về các kịch bản thử nghiệm, và giải thích chi tiết hơn về thử nghiệm, đánh giá hệ thống.

Câu hỏi: hệ thống phát hiện và khắc phục lỗi AWS được đánh giá như thế nào?

III/ Kết luận:

Đồng ý cho phép học viên được bảo vệ luận văn tốt nghiệp.

Hà Nội, ngày 16 tháng 7 năm 2025

Người nhận xét

Phạm Văn Cường

3 - 0,25
4