**NGUYỄN MẠNH THÀNH ĐỒ ÁN TỐT NGHIỆP HÀ NỘI, NĂM 2025**

|  |  |
| --- | --- |
| **bỘ GIÁO DỤC VÀ ĐÀO TẠO** | **BỘ NÔNG NGHIỆP VÀ Môi trường** |

TRƯỜNG ĐẠI HỌC THỦY LỢI



NGUYỄN MẠNH THÀNH

Nghiên cứu và ứng dụng quy trình devops để tự động hoá triển khai website thương mại điện tử kiến trúc microservices

ĐỒ ÁN TỐT NGHIỆP

HÀ NỘI, NĂM 2025

|  |  |
| --- | --- |
| **bỘ GIÁO DỤC VÀ ĐÀO TẠO** | **BỘ NÔNG NGHIỆP VÀ môi trường** |

TRƯỜNG ĐẠI HỌC THỦY LỢI

NGUYỄN MẠNH THÀNH

nghiên cứu và ứng dụng quy trình devops để tự động hoá triển khai website thương mại điện tử kiến trúc Microservices

|  |  |  |
| --- | --- | --- |
| Ngành: Công nghệ thông tin | |  |
| Mã số: 7480201 |  | |

|  |  |
| --- | --- |
| NGƯỜI HƯỚNG DẪN: | **Ths. Đỗ Oanh Cường** |
|  |  |

HÀ NỘI, NĂM 2025

|  |  |
| --- | --- |
| A blue and white logo  Description automatically generated | CỘNG HOÀ XÃ HỘI CHỦ NGHĨA VIỆT NAM  **Độc lập - Tự do - Hạnh phúc**  ----------★----------  **NHIỆM VỤ ĐỒ ÁN TỐT NGHIỆP** |

Họ tên sinh viên: Nguyễn Mạnh Thành Hệ đào tạo: Chính quy

Lớp: 63CNTTNB Ngành: Công nghệ thông tin

Khoa: Công nghệ thông tin

1. TÊN ĐỀ TÀI:

**NGHIÊN CỨU VÀ ỨNG DỤNG QUY TRÌNH DEVOPS ĐỂ TỰ ĐỘNG HÓA TRIỂN KHAI WEBSITE THƯƠNG MẠI ĐIỆN TỬ KIẾN TRÚC MICROSERVICE**

CÁC TÀI LIỆU CƠ BẢN:

|  |  |
| --- | --- |
| [1] | M. Fowler, "Monolith First," Martin Fowler, 2015. [Online]. Available: https://martinfowler.com/bliki/MonolithFirst.html. |
| [2] | J. L. a. M. Fowler, "Microservices," Martin Fowler, 2014. [Online]. Available: https://martinfowler.com/articles/microservices.html. |
| [3] | R. Hat, "What is CI/CD?," Red Hat, 2022. [Online]. Available: https://www.redhat.com/en/topics/devops/what-is-ci-cd. |
| [4] | Docker, "Get Started with Docker," Docker, Docker. [Online]. Available: https://docs.docker.com/get-started/. |
| [5] | Kubernetes, "Kubernetes Documentation," CNCF, 2024. [Online]. Available: https://kubernetes.io/docs/home/. |
| [6] | R. Labs, "K3s - Lightweight Kubernetes," Rancher Labs, 2024. [Online]. Available: https://k3s.io/. |
| [7] | M. Project, "MetalLB," MetalLB, 2023. [Online]. Available: https://metallb.universe.tf/. |
| [8] | GitHub, "GitHub Actions Documentation," GitHub, 2024. [Online]. Available: https://docs.github.com/en/actions. |
| [9] | A. Security, "Trivy Documentation," Aqua Security, 2023. [Online]. Available: https://trivy.dev/latest/docs/. |
| [10] | P. Authors, "Prometheus Documentation," CNCF, 2023. [Online]. Available: https://prometheus.io/docs/introduction/overview/. |
| [11] | G. Labs, "Grafana Documentation," Grafana Labs, 2024. [Online]. Available: https://grafana.com/docs/grafana/latest/. |

2 - NỘI DUNG CÁC PHẦN THUYẾT MINH VÀ TÍNH TOÁN:

|  |  |
| --- | --- |
| **Nội dung cần thuyết minh** | **Tỷ lệ** |
| Chương 1: Giới thiệu | 5% |
| Chương 2: Cơ sở lý thuyết | 30% |
| Chương 3: Thiết kế hệ thống | 35% |
| Chương 4: Kết quả triển khai hệ thống | 25% |
| Chương 5: Kết luận | 5% |

3. GIÁO VIÊN HƯỚNG DẪN TỪNG PHẦN

|  |  |
| --- | --- |
| Phần | Họ và tên giáo viên hướng dẫn |
| Chương 1: Giới thiệu | ThS. Đỗ Oanh Cường |
| Chương 2: Cơ sở lý thuyết |
| Chương 3: Thiết kế hệ thống |
| Chương 4: Kết quả triển khai hệ thống |
| Chương 5: Kết luận |

**4. NGÀY GIAO NHIỆM VỤ ĐỒ ÁN TỐT NGHIỆP**

Ngày...... tháng...... năm 2025

|  |  |
| --- | --- |
| **Trưởng Bộ môn**  *(Ký và ghi rõ Họ tên)* | **Giáo viên hướng dẫn chính**  *(Ký và ghi rõ Họ tên)*  *ThS. Đỗ Oanh Cường* |

Nhiệm vụ Đồ án tốt nghiệp đã được Hội đồng thi tốt nghiệp của Khoa thông qua.

|  |  |
| --- | --- |
| Ngày...... tháng...... năm 2025 | **Chủ tịch Hội đồng** |
|  | *(Ký và ghi rõ Họ tên)* |

Sinh viên đã hoàn thành và nộp bản Đồ án tốt nghiệp cho Hội đồng thi ngày...... tháng...... năm 20......

**Sinh viên làm Đồ án tốt nghiệp**

*(Ký và ghi rõ Họ tên)*

*Nguyễn Mạnh Thành*

|  |  |
| --- | --- |
| logo | TRƯỜNG ĐẠI HỌC THUỶ LỢI  **KHOA CÔNG NGHỆ THÔNG TIN**  BẢN TÓM TẮT ĐỀ CƯƠNG ĐỒ ÁN TỐT NGHIỆP |

TÊN ĐỀ TÀI: Nghiên cứu và ứng dụng quy trình DevOps để tự động hóa triển khai website thương mại điện tử kiến trúc Microservices.

*Sinh viên thực hiện*: Nguyễn Mạnh Thành

*Lớp*: 63CNTT.NB

*Mã sinh viên:* 2151062867

*Số điện thoại:* 0359113298

*Email:* nmthanh3107.03@gmail.com

*Giáo viên hướng dẫn*: ThS. Đỗ Oanh Cường Email: cuongdo@tlu.edu.vn

**TÓM TẮT ĐỀ TÀI**

Trong phát triển phần mềm hiện đại, Microservices đang trở thành một mô hình phổ biến nhờ tính linh hoạt, dễ mở rộng và triển khai nhanh chóng. Tuy nhiên, việc triển khai Microservices cũng đặt ra nhiều thách thức, đặc biệt là đảm bảo tính ổn định và tránh sự cố khi cập nhật hệ thống. DevOps cung cấp các phương pháp tự động hóa giúp triển khai phần mềm liên tục, giảm thiểu downtime và đảm bảo dịch vụ hoạt động ổn định.

Đề tài này tập trung vào việc tích hợp DevOps vào quá trình triển khai và quản lý Microservices, nhằm đảm bảo hệ thống có thể vận hành liên tục ngay cả khi có sự thay đổi hoặc cập nhật. Nội dung chính bao gồm:

1. Phân tích kiến trúc Microservices và các rủi ro khi triển khai.
2. Sử dụng công cụ DevOps (Docker, Kubernetes, Jenkins) để tự động hóa quy trình phát triển, triển khai và giám sát áp dụng lên hệ thống.
3. Thiết kế pipeline CI/CD nhằm giảm thiểu thời gian ngừng hoạt động (downtime) và tăng độ tin cậy hệ thống.

**CÁC MỤC TIÊU CHÍNH**

1. Tự động hóa quy trình build, test, deploy từng service độc lập.
2. Xây dựng pipeline CI/CD đảm bảo zero-downtime deployment
3. Đề xuất mô hình DevOps áp dụng cho website TMĐT.

**KẾT QUẢ DỰ KIẾN**

1. Xây dựng thành công hệ thống Microservices cho thương mại điện tử được triển khai tự động hóa thông qua quy trình DevOps (CI/CD, Infrastructure as Code, Monitoring), đảm bảo tính ổn định, khả năng mở rộng và phục hồi sự cố.
2. Đề xuất mô hình DevOps tối ưu cho hệ thống sử dụng mô hình Microservice.

**KẾ HOẠCH THỰC HIỆN**

|  |  |  |  |
| --- | --- | --- | --- |
| **TT** | **Thời gian** | **Công việc chính** | **Mục tiêu** |
| 1 | 31/03 - 07/04 | - Nghiên cứu yêu cầu - Thiết kế kiến trúc | Tài liệu đặc tả + Sơ đồ hệ thống |
| 2 | 08/04 - 21/04 | - Xây dựng UI cơ bản cho hệ thống thương mại điẹn tử minh hoạ | Hoàn thành giao diện người dùng |
| 3 | 22/04 - 05/05 | - Phát triển Microservices | Product, Cart, User, Payment services chạy trong container |
| 4 | 06/05 - 19/05 | - Triển khai Kubernetes - CI/CD đơn giản | Hệ thống chạy ổn định trên K8s cluster |
| 5 | 20/05 - 02/06 | - Thiết lập CI/CD với GitHub Actions - Thử áp dụng Rolling Update | Tự động deploy khi push code + Cập nhật không downtime |
| 6 | 03/06 - 09/06 | - Kiểm thử hiệu năng và độ ổn định | Báo cáo kiểm thử |
| 7 | 10/06 - 16/06 | - Fix lỗi phát sinh - Tối ưu hệ thống | Hệ thống hoạt động ổn định |
| 8 | 17/06 - 06/07 | - Viết tài liệu hướng dẫn - Chuẩn bị báo cáo cuối cùng | Hoàn thành báo cáo + Slide thuyết trình |

**TÀI LIỆU THAM KHẢO**

1. <https://learn.microsoft.com/en-us/azure/architecture/microservices/>
2. <https://docs.docker.com/get-started/>
3. <https://kubernetes.io/docs/home/>
4. https://www.jenkins.io/doc/tutorials/build-a-node-js-and-react-app-with-npm/

LỜI CAM ĐOAN

Em xin cam đoan đồ án tốt nghiệp với đề tài:“Nghiên cứu và ứng dụng quy trình DevOps để tự động hoá triển khai website thương mại điện tử kiến trúc Microservices”  
là kết quả nghiên cứu và thực hiện của riêng em dưới sự hướng dẫn của ThS. Đỗ Oanh Cường, giảng viên Khoa Công nghệ thông tin, Trường Đại học Thủy Lợi.

Các số liệu, kết quả, hình ảnh, tài liệu trích dẫn trong đồ án đều được em ghi rõ nguồn gốc và hoàn toàn trung thực. Em xin chịu hoàn toàn trách nhiệm về tính chính xác và trung thực của nội dung đồ án này.

.

|  |  |
| --- | --- |
|  | **Tác giả ĐATN/KLTN**  *Chữ ký*  **Nguyễn Mạnh Thành** |

LỜI CÁM ƠN

Em xin bày tỏ lòng biết ơn sâu sắc và trân trọng nhất đến Thạc sĩ Đỗ Oanh Cường, giảng viên Khoa Công nghệ thông tin, Trường Đại học Thủy Lợi, người đã tận tình hướng dẫn, chỉ bảo và đồng hành cùng em trong suốt quá trình thực hiện đồ án tốt nghiệp với đề tài:“Nghiên cứu và ứng dụng quy trình DevOps để tự động hoá triển khai website thương mại điện tử kiến trúc Microservices”.Những ý kiến chỉ dẫn quý báu và sự hỗ trợ tận tâm của thầy đã giúp em hoàn thành tốt nhiệm vụ nghiên cứu và hoàn thiện đồ án này.Em xin gửi lời cảm ơn chân thành đến quý thầy cô Khoa Công nghệ thông tin, Trường Đại học Thủy Lợi, những người đã truyền đạt cho em những kiến thức, kỹ năng và kinh nghiệm quý giá trong suốt quá trình học tập và rèn luyện tại nhà trường. Đây chính là nền tảng quan trọng giúp em hoàn thành đồ án tốt nghiệp này.

Mặc dù đã nỗ lực hết mình, song do thời gian và năng lực còn hạn chế, đồ án chắc chắn không tránh khỏi những thiếu sót. Em rất mong nhận được sự cảm thông và những góp ý quý báu từ quý thầy cô và các anh chị, bạn bè để hoàn thiện hơn trong tương lai.

Em xin trân trọng cảm ơn!

.

MỤC LỤC

[DANH MỤC CÁC HÌNH ẢNH v](#_Toc202641037)

[DANH MỤC CÁC TỪ VIẾT TẮT VÀ GIẢI THÍCH CÁC THUẬT NGỮ vi](#_Toc202641038)

[CHƯƠNG 1 GIỚI THIỆU 1](#_Toc202641039)

[1.1 Giới thiệu đề tài 1](#_Toc202641040)

[1.1.1 Lí do chọn đề tài 1](#_Toc202641041)

[1.1.2 Mục tiêu nghiên cứu đề tài 2](#_Toc202641042)

[1.2 Đối tượng và phạm vi nghiên cứu 2](#_Toc202641043)

[1.2.1 Đối tượng nghiên cứu 2](#_Toc202641044)

[1.2.2 Phạm vi nghiên cứu 2](#_Toc202641045)

[1.3 Phương pháp nghiên cứu và công cụ sử dụng 3](#_Toc202641046)

[1.3.1 Phương pháp nghiên cứu 3](#_Toc202641047)

[1.3.2 Công nghệ sử dụng 3](#_Toc202641048)

[1.4 Cấu trúc nội dung báo cáo 4](#_Toc202641049)

[CHƯƠNG 2 CƠ SỞ LÝ THUYẾT 5](#_Toc202641050)

[2.1 Kiến trúc Microservices và sự khác nhau với kiến trúc Monolithic 5](#_Toc202641051)

[2.1.1 Kiến trúc Monolithic 5](#_Toc202641052)

[2.1.2 Kiến trúc Microservices 8](#_Toc202641053)

[2.2 Mô hình DevOps và CI/CD 10](#_Toc202641054)

[2.2.1 Mô hình DevOps 10](#_Toc202641055)

[2.2.2 Tích hợp liên tục và triển khai liên tục (CI/CD) 13](#_Toc202641056)

[2.3 Công nghệ sử dụng trong đề tài 14](#_Toc202641057)

[2.3.1 Docker và containerization 15](#_Toc202641058)

[2.3.2 Kubernetes 19](#_Toc202641059)

[2.3.3 MetalLB – LoadBalancer cho môi trường bare-metal 23](#_Toc202641060)

[2.3.4 NFS – Hệ thống chia sẻ tệp cho Kubernetes 24](#_Toc202641061)

[2.3.5 GitHub Actions 26](#_Toc202641062)

[2.3.6 Trivy: Quét lỗ hổng của Docker Image 27](#_Toc202641063)

[2.3.7 Prometheus: Hệ thống thu thập và lưu trữ số liệu 29](#_Toc202641064)

[2.3.8 Grafana: Nền tảng trực quan hóa và phân tích 30](#_Toc202641065)

[CHƯƠNG 3 THIẾT KẾ HỆ THỐNG 31](#_Toc202641066)

[3.1 Bối cảnh và vấn đề 31](#_Toc202641067)

[3.2 Yêu cầu giải pháp 31](#_Toc202641068)

[3.3 Phân tích và đưa ra kiến trúc tổng quan cho quy trình 32](#_Toc202641069)

[3.4 Phân tích và thiết kế kiến trúc ứng dụng Microservices 33](#_Toc202641070)

[3.5 Thiết kế kiến trúc hạ tầng. 35](#_Toc202641071)

[3.6 Thiết kế quy trình CI/CD với GitHub Actions 37](#_Toc202641072)

[CHƯƠNG 4 KẾT QUẢ TRIỂN KHAI HỆ THỐNG 40](#_Toc202641073)

[4.1 Kết quả xây dựng và cấu hình hạ tầng 40](#_Toc202641074)

[4.2 Kết quả đóng gói và phân phối ứng dụng 42](#_Toc202641075)

[4.3 Kết quả tự động hóa CI/CD 43](#_Toc202641076)

[4.4 Kết quả Tích hợp An ninh và Giám sát (DevSecOps) 44](#_Toc202641077)

[CHƯƠNG 5 KẾT LUẬN 46](#_Toc202641078)

[5.1 Tóm tắt kết quả đạt được 46](#_Toc202641079)

[5.2 Đánh giá chung 47](#_Toc202641080)

[5.2.1 Ưu điểm 47](#_Toc202641081)

[5.2.2 Hạn chế 47](#_Toc202641082)

[5.3 Hướng phát triển trong tương lai 48](#_Toc202641083)

[TÀI LIỆU THAM KHẢO 49](#_Toc202641084)

DANH MỤC CÁC HÌNH ẢNH

[Hình 2.1Kiến trúc Monolithic – toàn bộ ứng dụng được đóng gói triển khai dưới một khối duy nhất 5](#_Toc202640950)

[Hình 2.2Kiến trúc Microservices – hệ thống được chia thành các dịch vụ độc lập, giao tiếp thông qua API 8](#_Toc202640952)

[Hình 2.3Vòng đời DevOps – minh họa quy trình phát triển và vận hành phần mềm liên tục 12](#_Toc202640954)

[Hình 2.4Khác biệt giữa máy ảo và container 15](#_Toc202640956)

[Hình 2.5Biểu tượng (logo) của Docker 16](#_Toc202640958)

[Hình 2.6 Kiến trúc Docker 18](#_Toc202640960)

[Hình 2.7Biểu tượng (logo) của Kubernetes 19](#_Toc202640962)

[Hình 2.8 Biểu tượng của K3s 22](#_Toc202640964)

[Hình 2.9 Biểu tượng của MetalLB 23](#_Toc202640966)

[Hình 2.10 Biểu tượng (logo) của GitHub Actions 26](#_Toc202640968)

[Hình 2.11Biểu tượng của Trivy 28](#_Toc202640970)

[Hình 2.12 Biểu tượng của Prometheus 29](#_Toc202640972)

[Hình 2.13Biểu tượng của Grafana 30](#_Toc202640974)

[Hình 3.1Sơ đồ kiến trúc ứng dụng, minh họa các Microservices và luồng giao tiếp thông qua API Gateway 34](#_Toc202640976)

[Hình 3.2Sơ đồ hạ tầng trên một cụm K3s 36](#_Toc202640978)

[Hình 3.3Sơ đồ kiến trúc CI/CD của GitHub Actions với cụm K3s 38](#_Toc202640980)

[Hình 4.1Kết quả cụm K3s 40](#_Toc202640982)

[Hình 4.2Kết quả khi sử dụng MetalLB 41](#_Toc202640984)

[Hình 4.3Các thư mục được dùng chung 42](#_Toc202640986)

[Hình 4.4 Các image của dự án được lưu trữ và phiên bản hóa trên Docker Hub 43](#_Toc202640988)

[Hình 4.5 Một pipeline CI/CD đã chạy thành công trên GitHub Actions 44](#_Toc202640990)

[Hình 4.6 Kết quả giám sát Cluster và các Pods 46](#_Toc202640992)

DANH MỤC CÁC TỪ VIẾT TẮT VÀ GIẢI THÍCH CÁC THUẬT NGỮ

**API** Application Programming Interface

**CD** Continuous Delivery / Deployment (Giao hàng / Triển khai liên tục)

**CI** Continuous Integration (Tích hợp liên tục)

**DNS** Domain Name System

**ĐATN** Đồ án tốt nghiệp

**IaC** Infrastructure as Code

**K8s** Kubernetes

**KLTN** Khóa luận tốt nghiệp

**LAN** Local Area Network

**LVTN** Luận văn tốt nghiệp

**NFS** Network File System

**PTNT** Phát triển Nông thôn

**TMĐT** Thương mại điện tử

**VM** Virtual Machine

# GIỚI THIỆU

## Giới thiệu đề tài

### Lí do chọn đề tài

Trong bối cảnh hiện nay,việc phát triển phần mềm đang dần chuyển đổi theo hướng hiện đại hóa, tự động hóa và phân tán để đáp ứng yêu cầu ngày càng cao về tốc độ, hiệu năng và độ tin cậy. Các mô hình phát triển truyền thống, nơi quy trình triển khai chủ yếu thực hiện thủ công và phụ thuộc nhiều vào con người, thường gặp phải các vấn đề như thời gian phát hành chậm, khó kiểm soát lỗi và không đảm bảo tính ổn định giữa các môi trường. Chính vì vậy, các mô hình mới như Microservices kết hợp với DevOps và CI/CD đã được ứng dụng rộng rãi để khác phục những hạn chế trên.

Sự xuất hiện của mô hình DevOps cùng với kỹ thuật CI/CD (Continuous Integration / Continuous Deployment) đã mang đến một hướng tiếp cận mới giúp tự động hóa toàn bộ quy trình xây dựng, kiểm thử và triển khai phần mềm. DevOps không chỉ giúp rút ngắn thời gian phát triển mà còn nâng cao tính ổn định của hệ thống, cải thiện khả năng phát hiện và khắc phục lỗi, đồng thời tăng cường sự phối hợp giữa các bộ phận phát triển và vận hành.

Đặc biệt, trong lĩnh vực thương mại điện tử (TMĐT), nơi tốc độ triển khai và khả năng cập nhật nhanh tính năng là yếu tố sống còn, việc áp dụng DevOps mang lại nhiều lợi thế. Người dùng ngày nay kỳ vọng trải nghiệm dịch vụ liền mạch, liên tục, không gián đoạn và được cập nhật thường xuyên. Mô hình truyền thống khó có thể đáp ứng được những kỳ vọng này do hạn chế về tốc độ triển khai và sự phức tạp trong quản lý thay đổi.

Đồng thời, sự phát triển của các công nghệ mã nguồn mở như Docker, Kubernetes, GitHub Actions... cho phép triển khai các mô hình DevOps một cách linh hoạt ngay trên hạ tầng tự quản lý, mà không cần phụ thuộc vào các dịch vụ cloud thương mại. Điều này mở ra cơ hội để các hệ thống thương mại điện tử cỡ vừa và nhỏ có thể tiếp cận mô hình Microservices và DevOps một cách tiết kiệm chi phí nhưng vẫn hiệu quả và dễ mở rộng.

Xuất phát từ thực tiễn trên, em lựa chọn đề tài: **“Nghiên cứu và ứng dụng quy trình DevOps để tự động hóa triển khai website thương mại điện tử kiến trúc Microservices”**.

### Mục tiêu nghiên cứu đề tài

* Tự động hoá quy trình build, test và deploy từng service trong hệ thống.
* Xây dựng pipeline CI/CD nhằm giảm thiểu thời gian triển khai và hạn chế downtime.
* Đề xuất mô hình DevOps phù hợp để áp dụng cho các hệ thống thương mại điện tử theo kiến trúc Microservices.

## Đối tượng và phạm vi nghiên cứu

### Đối tượng nghiên cứu

Đối tượng nghiên cứu của đề tài là quy trình DevOps trong triển khai hệ thống website TMĐT phát triển theo kiến trúc Microservices.

Cụ thể:

* Quy trình DevOps: bao gồm tự động hoá quá trình CI/CD và triển khai dịch vụ backend theo mô hình container hóa.
* Hệ thống website TMĐT mẫu: được chia thành các Microservices như User Service, Product Service, Order Service và Payment Service.
* Hạ tầng triển khai: sử dụng công nghệ Container (Docker), hệ thống điều phối (Kubernetes).

### Phạm vi nghiên cứu

Về quy trình: Đề tài tập trung nghiên cứu và ứng dụng quy trình DevOps nhằm tự động hóa việc triển khai hệ thống thương mại điện tử phát triển theo mô hình Microservices.

Về công nghệ sử dụng:

* Frontend: Xây dựng giao diện người dùng bằng ReactJS và Tailwind CSS.
* Backend: Phát triển các dịch vụ backend với Node.js.
* Dữ liệu: Lưu trữ đơn giản dưới dạng file CSV ứng với từng service.
* DevOps: Sử dụng Docker để container hoá ứng dụng, thiết lập CI/CD với GitHub Actions, triển khai hệ thống qua Kubernetes.

Về chức năng hệ thống TMĐT: Xây dựng website mẫu với các chức năng cơ bản như: đăng nhập/đăng ký, hiển thị danh sách sản phẩm, tạo đơn hàng và xử lý thanh toán.

Giới hạn:

* Không tập trung vào phát triển nghiệp vụ TMĐT phức tạp.
* Không tập trung nghiên cứu các giải pháp bảo mật chuyên sâu.

## Phương pháp nghiên cứu và công cụ sử dụng

### Phương pháp nghiên cứu

Tìm hiểu tài liệu: Tìm kiếm và tham khảo các tài liệu, video hướng dẫn liên quan đến DevOps, kiến trúc Microservices, Docker, Kubernetes, Github Actions, cũng như các công nghệ React, Nodejs, Tailwind CSS.

Xây dựng hệ thống mẫu: Tiến hành xây dựng một website TMĐT mẫu, chia thành các service nhỏ. Sau đó thực hành đóng gói từng service bằng Docker, viết các file cấu hình CI/CD, triển khai các container này bằng Kubernetes.

Thực hành triển khai: Triển khai thử hệ thống lên môi trường thử nghiệm, kiểm tra khả năng tự động hoá triển khai, phát hiện và khắc phục lỗi trong quá trình thực hiện.

Đánh giá kết quả: Sau khi triển khai thành công, em đánh giá lại quá trình, và rút ra kinh nghiệm cho các dự án khác.

### Công nghệ sử dụng

Trong quá trình thực hiện đề tài, em sử dụng nhiều công cụ và công nghệ khác nhau nhắm đáp ứng đúng kế hoạch định ra. Ở phần giao diện người dùng (frontend), em đã sử dụng ReactJS kết hợp với Tailwind CSS để xây dựng thành phần UI một cách cơ bản để demo một web thương mại điện tử đơn giản với một số chức năng mẫu. Phía backend, em sử dụng Node.js để xây dựng các service ứng với các chức năng riêng biệt trong cấu trúc Microservices như User Service, Product Service, Order Service và Payment Service.

Dữ liệu của từng dịch vụ được lưu trữ dưới dạng file CSV đơn giản, phù hợp cho mục đích mô phỏng hệ thống TMĐT ở mức nhỏ. Để đảm bảo tính đóng gói và khả năng linh hoạt, các dịch vụ này được container hóa bằng Docker, sử dụng thêm Docker Compose để quản lý các container trong môi trường phát triển cục bộ.

Hệ thống sau đó được triển khai lên môi trường phân tán sử dụng K3s – một phiên bản nhẹ của Kubernetes – kết hợp với MetalLB để hỗ trợ cân bằng tải trong mạng nội bộ. Bên cạnh đó, để thực hiện công việc quan trọng nhất của DevOps là quy trình tự động hóa CI/CD. Em đã sử dụng đến GitHub Actions giúp tự động hóa hoàn toàn quá trình build, test và deploy mỗi khi có một bản cập nhật mới lên nhánh main trong GitHub. Việc triển khai thử nghiệm được thực hiện trên cụm máy ảo nội bộ gồm ba node: một master và hai worker node, mô phỏng mô hình triển khai thực tế trong môi trường product.

## Cấu trúc nội dung báo cáo

Báo cáo được tổ chức thành các chương như sau:

* Chương 1: Giới thiệu đề tài: Trình bày lý do chọn đề tài, mục tiêu, phạm vi và đối tượng nghiên cứu, phương pháp nghiên cứu và công nghệ sử dụng, cấu trúc nội dung báo cáo.
* Chương 2: Cơ sở lý thuyết : Trình bày kiến thức nền tảng về Microservices, DevOps, CI/CD và các công nghệ sử dụng trong quá trình nghiên cứu và triển khai.
* Chương 3: Thiết kế hệ thống: Phân tích quy trình xây dựng hệ thống thực tế cùng sơ đồ triển khai.
* Chương 4: Triển khai hệ thống: Trình bày quy trình xây dựng hệ thống thực tế, đóng gói bằng Docker, thiết lập CI/CD và triển khai trên Kubernetes.
* Chương 5: Kết luận: Đưa ra kết luận cùng những điểm đã làm, chưa làm được và hướng phát triển.

# CƠ SỞ LÝ THUYẾT

## Kiến trúc Microservices và sự khác nhau với kiến trúc Monolithic

### Kiến trúc Monolithic

#### Khái niệm về Monolithic

Kiến trúc Monolithic (kiến trúc nguyên khối) là kiểu kiến trúc truyền thống trong phát triển phần mềm, trong đó một ứng dụng được xây dựng như một đơn vị duy nhất và không thể tách rời. Trong mô hình này, tất cả thành phần chức năng từ giao diện người dùng, xử lý logic nghiệp vụ, và các lớp cơ sở dữ liệu đều sẽ được gộp chung và triển khai chung trong một tiến trình duy nhất.

A screenshot of a computer

AI-generated content may be incorrect.

Hình .Kiến trúc Monolithic – toàn bộ ứng dụng được đóng gói triển khai dưới một khối duy nhất

Hình .Kiến trúc Monolithic – toàn bộ ứng dụng được đóng gói triển khia dưới một khối duy nhất

#### Các đặc điểm của kiến trúc Monolithic

Kiến trúc Monolithic được định hình và nhận diện thông qua một số đặc điểm cấu trúc sau:

* Mã cơ sở (Codebase) hợp nhất: Toàn bộ mã nguồn của ứng dụng từ tầng giao diện người dùng đến tầng logic nghiệp vụ, sẽ được quản lý và lưu trữ tại một kho lưu trữ duy nhất. Điều này giúp việc thiết lập môi trường và quản lý dự án tốt hơn, tuy nhiên, khi quy mô dự án và đội ngũ phát triển tăng lên, nó có thể dẫn đến sự phức tạp trong việc phối hợp và nguy cơ xảy ra xung đột khi hợp nhất mã.
* Liên kết chặt chẽ: Các Module hoặc các thành phần chức năng trong ứng dụng có mức độ phụ thuộc vào nhau rất cao. Một sự thay đổi trong module có thể kéo theo việc điều chỉnh ở nhiều module khác đang sử dụng nó. Mặc dù vậy, tính liên kết chặt chẽ này giúp cho khả năng giao tiếp nội bộ tăng lên.
* Triển khai dưới dạng một khối duy nhất: Toàn bộ ứng dụng được biên dịch và đóng gói thành một tệp thực thi duy nhất, không thể phân chia. Quá trình triển khai cơ bản là thay thế toàn bộ khối ứng dụng cũ bằng khối ứng dụng mới. Điều này đồng nghĩa với việc, dù chỉ thay đổi nho nhỏ cho một chức năng, toàn bộ ứng dụng vẫn phải được xây dựng lại và triển khai lại từ đầu, thường đòi hỏi phải khởi động lại toàn bộ tiến trình máy chủ.
* Cơ sở dữ liệu tập trung: Thông thường, một ứng dụng Monolithic sẽ kết nối và sử dụng chung một cơ sở dữ liệu lớn, duy nhất, nơi lưu trữ toàn bộ dữ liệu của hệ thống. Mô hình này giúp đơn giản hóa việc quản lý giao dịch và đảm bảo tính nhất quán của dữ liệu. Tuy nhiên, nó cũng tạo ra một điểm gây lỗi duy nhất là khi cơ sở dữ liệu gặp sự cố, toàn bộ ứng dụng sẽ bị ảnh hưởng.

#### Ưu điểm của kiến trúc Monolithic

Kiến trúc Monolithic có rất nhiều ưu điểm:

* Đơn giản để phát triển (với một ứng dụng nhỏ và vừa): Với một đội nhóm nhỏ và một ứng dụng có phạm vi chức năng rõ ràng, việc quản lý một codebase duy nhất giúp đơn giản hóa đáng kể quy trình làm việc.
* Dễ dàng kiểm thử: Việc thực hiện kiểm thử trong một ứng dụng Monolithic thường đơn giản hơn đáng kể. Vì toàn bộ hệ thống chạy trong một tiến trình duy nhất, các kịch bản kiểm thử có thể được thực thi mà không cần phải thiết lập điều phối một loạt dịch vụ phụ thuộc.
* Dễ dàng triển khai(ở quy mô nhỏ: Quá trình triển khai một ứng dụng Monolithic thường rất đơn giản. Toàn bộ ứng dụng được đóng gói thành một tệp hoặc thư mục duy nhất. Việc triển khai chỉ đơn giản là sao chép gói này lên một hoặc nhiều máy chủ và khởi chạy tiến trình. Mô hình này không đòi hỏi các hệ thống điều phối container phức tạp hoặc các cơ chế quản lý cấu hình phân tán.
* Hiệu năng giao tiếp nội bộ: Đây là một lợi thế kỹ thuật lớn nhất của kiến trúc Monolithic. Do tất cả các thành phần chức năng cùng tồn tại trong một không gian bộ nhớ và một tiến trình.

#### Nhược điểm và thách thức

Tuy nhiều ưu điểm của kiến trúc Monolithic nhưng khi ứng dụng phát triển về quy mô và độ phức tạp, kiến trúc này đang dần lộ ra những nhược điểm:

* Khó bảo trì và mở rộng: Theo thời gian, codebase càng ngày càng đồ sộ và các thành phần sẽ chồng chéo lên nhau. Điều này khiến việc bảo trì, nâng cấp trở nên khó khăn và rủi ro, đồng thời kéo dài thời gian để một nhà phát triển mới có thể nắm bắt và đóng góp hiệu quả cho dự án.
* Cản trở việc áp dụng công nghệ mới: Ứng dụng bị ràng buộc chặt chẽ vào một ngăn xếp công nghệ duy nhất, khiến việc nâng cấp hoặc áp dụng công nghệ mới cho từng phần trở nên bất khả thi nếu không thực hiện một cuộc tái cấu trúc toàn diện, vốn rất tốn kém và rủi ro.
* Triển khai chậm và rủi ro cao: Bất kỳ thay đổi nhỏ nào cũng đòi hỏi phải kiểm thử và triển khai lại toàn bộ hệ thống, làm chậm chu kỳ phát hành và tăng "bán kính ảnh hưởng" của lỗi. Điều này biến mỗi lần triển khai thành một sự kiện rủi ro cao, thường yêu cầu thời gian chết (downtime) theo kế hoạch.
* Tính chịu lỗi kém: Một sự cố trong một module không trọng yếu, ví dụ như rò rỉ bộ nhớ, cũng có thể làm toàn bộ hệ thống ngừng hoạt động. Hệ quả là tất cả các chức năng, bao gồm cả những chức năng cốt lõi, đều bị gián đoạn, gây ảnh hưởng nghiêm trọng đến trải nghiệm người dùng và hoạt động kinh doanh.

### Kiến trúc Microservices

#### Khái niệm Microservices

Trái ngược với Monolithic, Kiến trúc Microservices là một phương pháp tiếp cận kiến trúc trong đó một ứng dụng lớn được cấu thành từ một tập hợp các dịch vụ nhỏ, độc lập và có khả năng tự triển khai. Mỗi dịch vụ được thiết kế để thực hiện một chức năng nghiệp vụ cụ thể.

Khác với kiến trúc Monolithic nơi mọi thứ được đóng gói làm một, các microservice giao tiếp với nhua thông qua các cơ chế gọn nhẹ và được định nghĩa rõ ràng, phổ biến nhất là thông qua các APIs.

A screenshot of a computer

AI-generated content may be incorrect.

Hình .Kiến trúc Microservices – hệ thống được chia thành các dịch vụ độc lập, giao tiếp thông qua API

A screenshot of a computer

AI-generated content may be incorrect.

Hình .Kiến trúc Microservices – hệ thống được chia thành các dịch vụ độc lập, giao tiếp thông qua API

#### Các đặc điểm của kiến trúc Microservices

Một hệ thống được xem là tuân thủ kiến trúc Microservices khi nó thể hiện các đặc điểm sau:

* Tính đơn nhiệm và tập trung vào nghiệp vụ: Mỗi microservice chỉ chịu trách nhiệm cho một chức năng nghiệp vụ duy nhất.
* Tính độc lập và tự chủ: Mỗi service có thể được phát triển, triển khai, vận hành và mở rộng một cách độc lập mà không gây ảnh hưởng đến các service khác. Một đội nhóm có thể sửa đổi cập nhật ở một service này mà không gây ảnh hưởng đến service khác.
* Liên kết lỏng lẻo: Sự phụ thuộc giữa các service được giữ ở mức độ tối thiểu. Chúng giao tiếp qua các API ổn định. Nếu một service thay đổi logic nội bộ mà vẫn giữ nguyên API, các service khác sẽ không bị ảnh hưởng.
* Quản lý dữ liệu phi tập trung: Đây là một đặc điểm quan trọng nhất. Để đảm bảo tính độc lập hoàn toàn, mỗi Microservice sở hữu và quản lý cơ sở dữ liệu riêng của mình.
* Đa dạng về công nghệ: Vì các service là độc lập nên việc lựa chọn công nghệ đều phụ thuộc và quyết định của đội nhóm để giải quyết bài toán của họ.
* Khả năng chịu lỗi: Hệ thống được thiết kế để có thể đối phó với sự cố. Nếu một service không hoạt động, toàn bộ hệ thống vẫn có thể tiếp tục hoạt động.

#### Ưu điểm của kiến trúc Microservices

Kiến trúc Microservices mang lại nhiều lợi ích chiến lược, giúp các tổ chức phát triển phần mềm có khả năng linh hoạt thích ứng và cạnh tranh tốt hơn:

* Sự linh hoạt và tốc độ: Việc các service độc lập với nhau nên việc chia nhóm thực hiện các service cùng một lúc giúp cho giảm đáng kể thời gian và chi phí giữa các nhóm.
* Khả năng mở rộng linh hoạt: Đây là một trong những ưu điểm vượt trội nhất về mặt tối ưu hóa tài nguyên. Thay vì phải mở rộng tất cả toàn bộ ứng dụng thì với kiến trúc Microservice thì khi quá tải ở service nào sẽ được mở rộng cho service đấy là tiết kiệm được tài nguyên khi không cần mở rộng cho những service ít biến động. Điều này giúp tối ưu chi phí hạ tầng một cách hiệu quả.
* Dễ dàng bảo trì: Mỗi service có một codebase nhỏ, tập trung và có ranh giới rõ ràng. Điều này giúp cho các nhà phát triển chỉ cần hiểu rõ dịch vụ cụ thể thay vì phải hiểu toàn bộ hệ thống khổng lồ của Monolithic. Giúp việc bảo trì một service là rất dễ dàng.
* Tăng cường khả năng chịu lỗi: Kiến trúc Microservices cải thiện đáng kể độ tin cậy của hệ thống bằng cách làm giảm đáng kể phạm vi lỗi. Vì sự cố sẽ chỉ xảy ra ở một service độc lập và ít có khả năng làm sập hệ thống.
* Tự do lựa chọn công nghệ: Kiến trúc cho phép các đội nhóm sử dụng công nghệ phù hợp nhất cho từng bải toán cụ thể, thúc đẩy sự đổi mới. Dù trong một dự án có thể sử dụng các công nghệ khác nhau cho mỗi service. Sự đa dạng này không chỉ tối ưu hóa về mặt kỹ thuật mà còn giúp doanh nghiệp dễ dàng thu hút và giữ chân nhân tài có chuyên môn đa dạng.

#### Nhược điểm và thách thức

Mặc dù có rất nhiều ưu điểm, việc áp dụng kiến trúc Microservices cũng đồng nghĩa với việc chấp nhận đối mặt với độ phức tạp vốn có của hệ thống.

* Sự phức tạp trong vận hành: Việc triển khai,giám sát, và quản lý hệ thống với hàng chục và hàng trăm dịch vụ (Các hệ thống lớn) là một công việc phức tạp hơn rất nhiều so với một ứng dụng Monolithic.
* Độ phức tạp trong kiểm thử: Trong khi việc kiểm thử từng service thì khá đơn giản nhưng khi thử từ đầu đến cưới lại trở lên vô cùng phức tạp.
* Giao tiếp giữa các dịch vụ: Việc các dịch vụ giao tiếp với nhau ảnh hưởng trực tiếp đến hiệu năng và độ phức tạp của hệ thống. Việc quản lý các API cũng trở thành một vấn đề quan trọng để đảm bảo các thay đổi không làm hỏng các dịch vụ phụ thuộc.

## Mô hình DevOps và CI/CD

### Mô hình DevOps

#### Định nghĩa DevOps

DevOps là một thuật ngữ kết hợp từ Development (Phát triển) và Operations (Vận hành). Tuy nhiên, sẽ thật thiếu sót nếu chỉ định nghĩa DevOps như một chức danh công việc hay một bộ công cụ. Về bản chất, đây là một văn hóa, một tập hợp cac quy trình thực tiễn và công cụ nhằm xóa bỏ rào cản, tăng cường sự hợp tác và tích hợp giữa đội ngũ phát triển phần mềm và đội ngũ vận hành hệ thống.

Mục tiêu cuối cùng của DevOps là tạo ra một quy trình liền mạch và tự động, cho phép xây dựng, kiểm thử và phát hành phần mềm một cách nhanh chóng, thường xuyên và đáng tin cậy. Điều này không chỉ giúp cải thiện hiệu quả kỹ thuật mà còn trực tiếp phục vụ các mục tiêu chiến lược kinh doanh, chẳng hạn như tăng tốc độ đưa sản phẩm ra thị trường và nâng cao khả năng phản ứng trước những thay đổi của người dùng.

#### Vấn đề giải quyết

Để hiểu rõ tại sao DevOps ra đời, chúng ta cần nhìn vào mô hình làm việc truyển thống, nơi tồn tại một sự chia sẻ sâu sắc giữa hai bộ phận cốt lõi. Trong mô hình này, hai đội nhóm thường hoạt động trong các nơi riêng biệt với những mục tiêu và thước đo hiệu suất dường như đối lập nhau:

* Đội phát triển (Dev): Mục tiêu của họ là tạo ra sự thay đổi. Họ được khuyển khích liên tục phát triển các tính năng mới, cải thiện sản phẩm và đưa chúng đến tay người dùng nhanh nhất có thể. Tốc độ là yếu tố then chốt.
* Đội vận hành (Ops): Mục tiêu của họ là duy nhất sự ổn định. Họ chịu trách nhiệm đảm bảo hệ thống luôn hoạt động mượt mà, đáng tin cậy với thời gian hoạt động cao nhất. Do đó, họ thường có xu hướng e ngại và kiểm soát chặt chẽ sự thay đổi, vì mỗi thay đổi đều tiềm ẩn rủi ri gây ra sự cố.

Sự xung đột về mục tiêu này tạo ra rào cản vô hình của sự hiểu lầm, thiếu tin tưởng và giao tiếp kém hiệu quả, gây ra sự chậm trễ, văn hóa đổ lỗi cho nhau và làm giảm hiệu suất tổng thể. DevOps ra đời chính là để phá vỡ bức tường này, thống nhất hai đội nhóm hướng tới một mục tiêu chung: cung cấp giá trị cho người dùng cuối một cách hiệu quả và bền vững nhất.

#### Ba thành phần chính của DevOps

DevOps được xây dựng trên ba thành phần chính không thể tách rời, thường được biết đến với:

* Văn hóa: Đây là yếu tố nền tảng và quan trọng nhất. DevOps thúc đẩy một văn hóa làm việc cao sự hợp tác, giao tiếp minh bạch và trách nhiệm chung. Thay vì đổi lỗi khi có sự cố, các thành viên từ cả hai phía cùng nhau phân tích nguyên nhân gốc rễ và tìm cách cải thiện quy trình. Các đội nhóm được tổ chức theo mô hình đa chức năng, nơi họ cùng nhau sở hữu sản phẩm từ lúc lên ý tưởng cho đến khi vận hành.
* Quy trình và thực tiễn: Văn hóa DevOps được hiện thực hóa thông qua việc áp dụng các quy trình có tính tự động hóa cao và lặp đi lặp lại. Các thực tiễn cốt lõi bao gồm: Tích hợp liên tục (CI), triển khai liên tục (CD), hạ tầng dưới dạng mã (Infrastructure as Code – IaC) – nơi hạ tầng được định nghĩa trong các tệp mã nguồn để đảm bảo tính nhất quán trong khả năng tái tạo.
* Công cụ: Công cụ là phương tiện vật chất để tự động hóa các quy trình và hỗ trợ văn hóa hợp tác. Mỗi chuỗi công cụ DevOps hoàn chỉnh không phải là một sản phẩm duy nhất, mà là một tập hợp các công cụ được tích hợp với nhau để hỗ trợ toàn bộ vòng đời phát triển phần mềm, từ lập kế hoạch, viết mã, xây dựng, cho đến triển khai và giám sát.

#### Vòng đời của DevOps

Quy trình DevOps thường được minh họa bằng một vòng lặp vô tận, tượng trưng cho tính liên tục, không có điểm kết thúc của việc phát triển và vận hành phần mềm. Sơ đồ này không chỉ là một hình ảnh biểu trưng mà còn mô tả một chu trình công việc có cấu trúc rõ ràng.

A diagram of a diagram

AI-generated content may be incorrect.

Hình .Vòng đời DevOps – minh họa quy trình phát triển và vận hành phần mềm liên tục

A diagram of a diagram

AI-generated content may be incorrect.

Hình .Vòng đời DevOps – minh họa quy trình phát triển và vận hành phần mềm liên tục

Vòng lặp này bao gồm các giai đoạn kết nối chặt chẽ với nhau, mỗi giai đoạn có mục tiêu và công cụ hỗ trợ riêng:

* Plan (Lên kế hoạch): Giai đoạn khởi đầu, nơi các yêu cầu kinh doạn và phản hồi từ người dùng được phân tích để xác định các tính năng cần phát triển.
* Code (Viết mã): Các nhà phát triển thực hiện việc lập trình, việc sử dụng các công nghệ và hệ thống quản lý phiên bản như Git để theo dõi thay đổi.
* Build (Xây dựng): Mã nguồn từ nhiều nhà phát triển được tích hợp lại và biên dịch thành các gói thực thi.
* Test (Kiểm thử): Các bài kiểm thử tự động được thực thi để đảm bảo chất lượng, phát hiện lỗi sớm và ngăn chặn chúng lọt ra môi trường sản xuất.
* Release (Phát hành): Các gói thực thi đã vượt qua kiểm thử sẽ được gắn phiên bản và lưu trữ trong một kho lưu trữ để sẵn sàng triển khai.
* Deploy (Triển khai): Ứng dụng được tự động triển khai trên các môi trường, từ môi trường thử nghiệm đến môi trường sản xuất.
* Monitor (Giám sát): Liên tục thu thập dữ liệu về hiệu năng ứng dụng, trạng thái hà tầng và hành vi người dùng.

### Tích hợp liên tục và triển khai liên tục (CI/CD)

CI/CD là một thành phần cốt lõi trong DevOps. Nó thực hiện các bước trong DevOps thông qua một pipeline tự động.

#### Tích hợp liên tục (Continuous Integration -CI)

* Định nghĩa: CI là một kỹ thuật phát triển phần mềm, trong đó các lập trình viên thường xuyên hợp nhất (merge) các thay đổi về mã nguồn của họ vào một kho lưu trữ trung tâm. Sau mỗi lần hợp nhất, một quy trình xây dựng và kiểm thử tự động sẽ được kích hoạt.
* Vấn đề CI giải quyết: Trong mô hình cũ, các lập trình viên thường làm việc trên các nhành riêng biệt trong một thời gian dài. Đến cuối giai đoạn, khi họ cố gắng hợp nhất tất cả các thay đổi lại với nhau, họ sẽ phải đối mặt với một vấn đề là xung đột giữa các phiên bản. Vấn đề này rất khó giải quyết.
* Mục tiêu của CI: Bằng cách tích hợp thường xuyên, các lỗi tích hợp sẽ được phát hiện ngay lập tức, khi chúng còn nhỏ và dễ sửa chữa. Kết quả của một quy trình CI thành công là một gói thực thi đã được xác minh là có thể hoạt động được.

#### Giao hàng/Triển khai liên tục (Continuous Delivery/Deployment – CD)

CD là bước tiếp theo sau CI, tự động việc phát hành phần mềm. Có hai khái niệm liên quan nhưng khác nhau cần phân biệt rõ.

Giao hàng liên tục (Continuous Delivery):

* Định nghĩa: Mở rộng từ CI, Continuous Delivery tự động hóa việc phát hành mọi thay đổi đã qua kiểm thử đến một môi trường tương tự môi trường sản xuất.
* Điểm mấu chốt: Việc triển khai lên môi trường sản xuất thật vẫn cần một sự phê duyệt thủ công – thường là một cú nhấp chuột từ người quản lý hay trưởng nhóm.
* Mục đích: Đảm bảo rằng tại bất cứ thời điểm nào, bạn cũng có một phiên bản đã được kiểm thử kỹ lưỡng, sẵn sàng để phát hành cho người dùng cuối chỉ bằng một nút bấm. Điều này giúp giảm thiểu rủi ro và cho phép việc ra quyết định phát hành dựa trên yếu tố kinh doanh thay vì kỹ thuật.

Triển khai liên tục (Continous Deployment):

* Định nghĩa: Đây là mức độ tự động hóa cao nhất. Mỗi thay đổi vượt qua tất cả các giai đoạn kiểm thử tự động sẽ được tự động triển khai trực tiếp trên môi trường sản xuất mà không cần bất cứ sự can thiệp nào của con người.
* Mục đích: Tối ưu hóa tốc độ phản hồi và rút ngắn vòng lặp phản hồi đến mức tối thiểu. Chỉ những công ty có một hệ thống kiểm thử tự động cực kỳ đáng tin cậy và một văn hóa DevOps trưởng thành mới có thể áp dụng mô hình này.

## Công nghệ sử dụng trong đề tài

Trong đề tài này, các công nghệ cốt lõi được lựa chọn và triển khai không chỉ vì tính phổ biến mà còn vì khả năng tương thích cao với mô hình kiến trúc Microservices và quy trình DevOps hiện đại. Dưới đây là những công nghệ cốt lõi được dùng trong mô hình DevOps.

### Docker và containerization

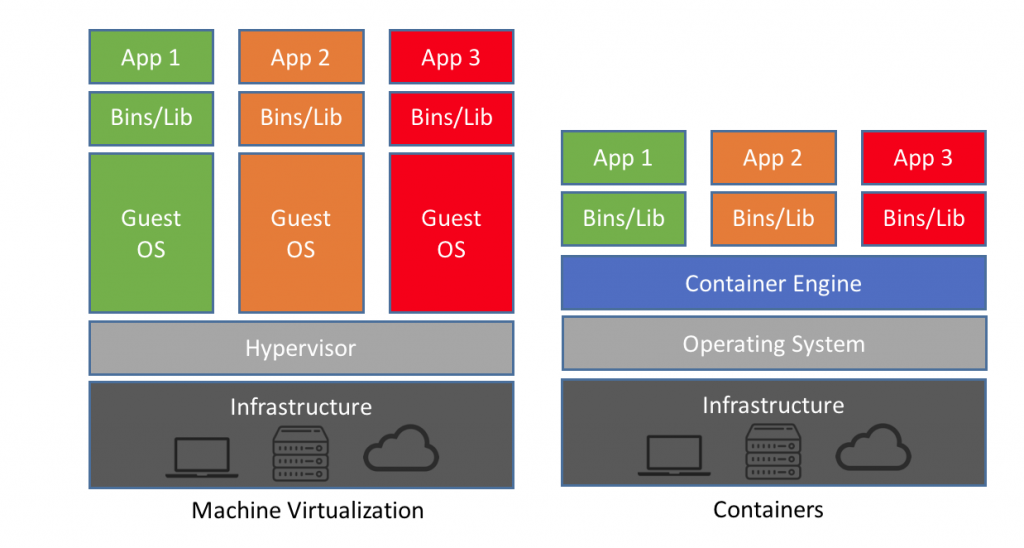
#### Containerization (Container hóa) là gì ?

Containerization, hay container hóa, là một hình thức ảo hóa ở cấp độ hệ điều hành. Đây là một phương pháp cho phép đóng gói một ứng dụng cùng với toàn bộ các thành phần phụ thuộc của nó – bao gồm mã nguồn, thư viện, file cấu hình và các biến môi trường – vào một đơn vị thực thi tiêu chuẩn, gọn nhẹ và độc lập gọi là container.

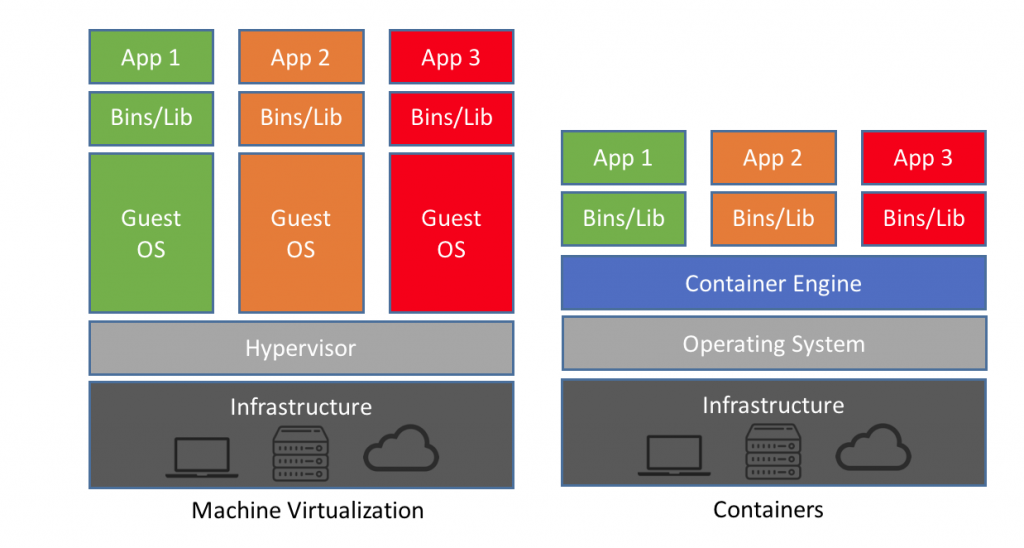
Về cơ bản, container tạo ra một môi trường nhất quán, đảm bảo rằng ứng dụng sẽ hoạt động giống hệt nhau ở mọi nơi, giải quyết triệt để vấn đề kinh điển kiểu “trên máy tôi nó vẫn chạy”. Một container đang chạy trên máy tính của lập trình viên có thể dễ dàng được chuyển và vận hành trên máy chủ thử nghiệm hoặc môi trường sản xuất mà không cần bất cứ thay đổi nào.

So sánh giữa Containerization và máy ảo (Virtual Machine – VM):

Mặc dù cả hai đều là công nghệ ảo hóa nhằm mục đích cô lập các ứng dụng, chúng hoạt động ở các cấp độ khác nhau và có những khác biệt cơ bản về kiến trúc và hiệu suất.



Hình .Khác biệt giữa máy ảo và container



Hình .Khác biệt giữa máy ảo và container

* Máy ảo: Ảo hóa tầng phần cứng. Máy ảo mô phỏng một máy tính vật lý đầy đủ từ CPU, RAM, ổ cứng cho đến card mạng. Để làm được điều này, VM chạy trên một lớp chung gian gọi là Hypervisor (giám sát máy ảo), và mỗi VM đều phải cài một hệ điều hành khách (Guest OS) hoàn chỉnh bên trong. Chính vì phải mang theo gánh nặng của cả một hệ điều hành riêng, VM rất nặng nề, chiếm dụng nhiều tài nguyên hệ thống và cần vài phút để khởi động.
* Container: Ảo hóa cấp hệ điều hành. Ngược lại, Container giống như việc không gian riêng biệt nhưng cùng chia sẻ một hạ tầng chung với tất cả container khác. Và hạ tâng chung đó là nhân (kernel) của hệ điều hành máy chủ. Container không cần một hệ điều hành khách riêng biệt. Thay vào đó, nó tận dụng các tính năng của kernel máy chủ để tạo ra không gian người dùng tách biệt. Vì không phải ảo hóa phần cứng nên container cực nhẹ, khởi động nhanh chóng và sử dụng tài nguyên hiệu quả hơn nhiều.

Nhờ những ưu điểm vượt trội về tốc độ, hiệu suất và tính gọn nhẹ, containerization đã trở thành công nghệ nền tảng và là lựa chọn ưu tiên trong các mô hình triển khai hiện đại như Microservices và DevOps.

#### Docker

Docker là một nền tảng mã nguồn mở dùng để phát triển, đóng gói và chạy ứng dụng. Docker cho phép người dùng tách biệt ứng dụng khỏi hạ tầng hệ thống, nhờ đó có thể triển khai phần mềm nhanh hơn và linh hoạt hơn.

Với Docker, người dùng có thể quản lý hạ tầng của mình tương tự như cách bạn quản lý mã nguồn ứng dụng. Bằng cách áp dụng các phương pháp của Docker trong việc đóng gói, kiểm thử và triển khai mã, người dùng có thể rút ngắn đáng kể thời gian từ lúc viết mã cho đến khi ứng dụng chạy thực tế trên môi trường production.

A blue letters on a black background

AI-generated content may be incorrect.

Hình .Biểu tượng (logo) của Docker

A blue letters on a black background

AI-generated content may be incorrect.

Hình .Biểu tượng (logo) của Docker

#### Kiến trúc của Docker

Docker được xây dựng dựa trên mô hình client – server (máy khách – máy chủ), trong đó các lệnh từ người dùng được gửi từ Docker client đến Docker daemon – tiến trình xử lý chính của Docker. Kiến trúc này giúp Docker đạt được sự linh hoạt trong vận hành và khả năng mở rộng cao khi triển khai trên các hệ thống phân tán.

Docker Daemon: Là thành phần trung tâm của kiến trúc Docker. Daemon chịu trách nhiệm quản lý toàn bộ vòng đời của container, từ việc xây dựng image, tạo container, kết nối mạng, gắn volume, cho đến việc khởi động và dừng container. Daemon luôn lắng nghe các yêu cầu từ client thông qua giao thức Docker REST API, và có thể giao tiếp với nhiều daemon khác để triển khai các dịch vụ phân tán.

Docker Client: Đây là công cụ dòng lệnh mà người dùng sử dụng sử dụng để tương tác với Docker. Khi người dùng gõ lệnh như docker build, docker run, … , client sẽ gửi yêu cầu tới daemon để thực hiện thao tác tương ứng.

Docker registry: Registry là nơi lưu trữ các Docker image. Khi người dùng sử lệnh docker pull hoặc docker run, Docker sẽ tìm và tải image từ registry được cấu hình sẵn. Mặc định, Docker sử dụng Docker Hub – một registry công khai do Docker Inc. quản lý. Ngoài ra, người dùng có thể thiết lậ registry riêng tư để lưu trữ các image nội bộ cho doanh nghiệp hoặc tổ chức.

Docker Object: Docker hoạt động dựa trên các đối tượng chính bao gồm:

* Images: Là khuôn mẫu chỉ đọc dùng để tạo ra container.
* Containers: Là một instance đang chạy của image, bao gồm mã nguồn, thư viện, file cấu hình cần thiết cho ứng dụng.
* Networks: Cho phép container giao tiếp với nhau và với thế giới bên ngoài.
* Volume: Dùng để lưu trữ dữ liệu bền vững, không bị mất khi container bị xóa.

A screenshot of a computer

AI-generated content may be incorrect.

Hình . Kiến trúc Docker

A screenshot of a computer

AI-generated content may be incorrect.

Hình . Kiến trúc Docker

Để hiểu rõ hơn về cách thức hoạt động của Docker, có thể quan sát mô hình tổng quan gồm ba thành phần chính: Client (máy khách), Docker Host (máy chủ Docker) và Registry (kho lưu trữ hình ảnh – image).

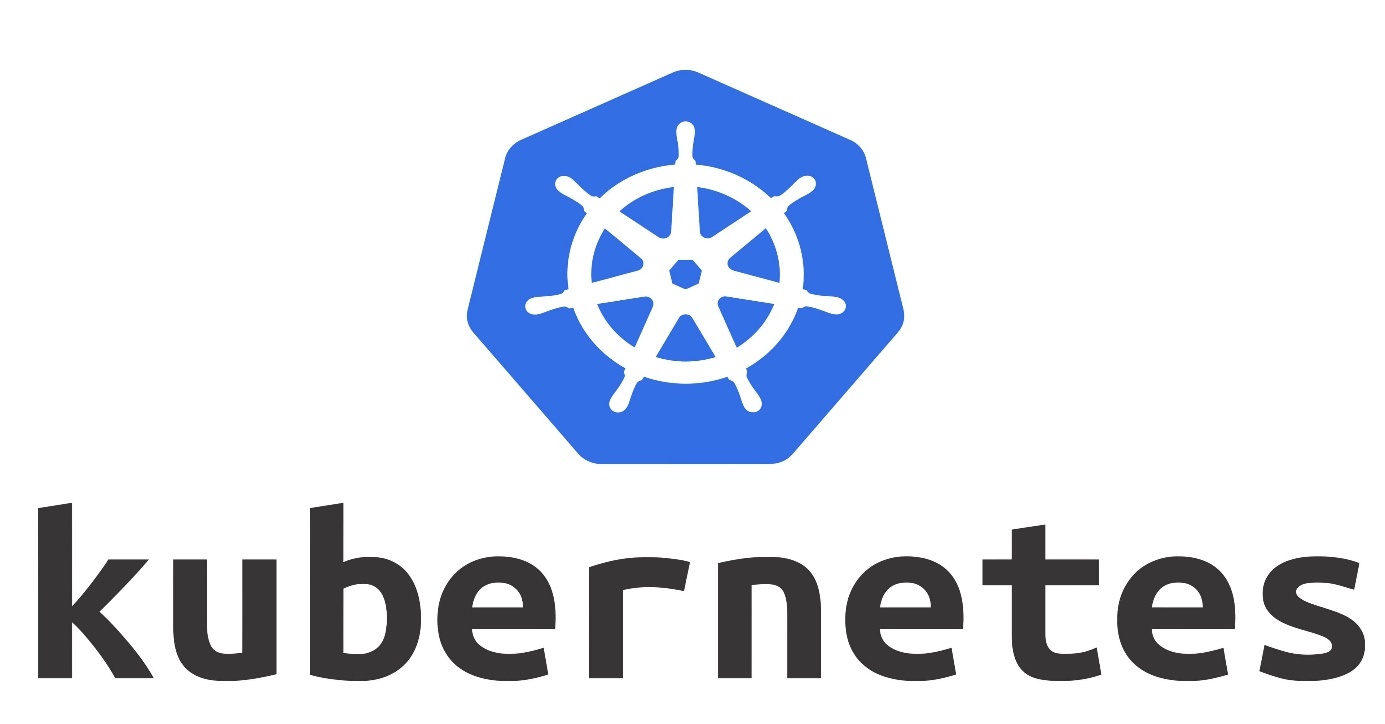
Docker Client là nơi người dùng trực tiếp tương tác với hệ thống thông qua các lệnh dòng lệnh (CLI) như docker build, docker run hoặc docker pull. Mỗi lệnh này đều gửi yêu cầu đến Docker daemon (dockerd) – thành phần trung tâm chịu trách nhiệm thực thi các tác vụ như xây dựng image, tạo container hoặc truy xuất image từ registry. Khi người dùng sử dụng lệnh docker build, daemon sẽ tiến hành xây dựng image từ Dockerfile và lưu image đó vào bộ nhớ cục bộ trên máy chủ. Lệnh docker run sẽ yêu cầu daemon tạo một container mới từ image có sẵn và khởi chạy nó. Trường hợp image chưa tồn tại trên máy cục bộ, Docker sẽ tự động kết nối đến Registry để tải image về thông qua lệnh docker pull.

Docker Host là môi trường lưu trữ và vận hành các thành phần cốt lõi bao gồm image và container. Image là các khuôn mẫu chỉ đọc, được sử dụng để tạo ra các container – các môi trường chạy ứng dụng thực tế, chứa toàn bộ mã nguồn, thư viện và cấu hình cần thiết. Mỗi container là một instance độc lập được sinh ra từ một image cụ thể.

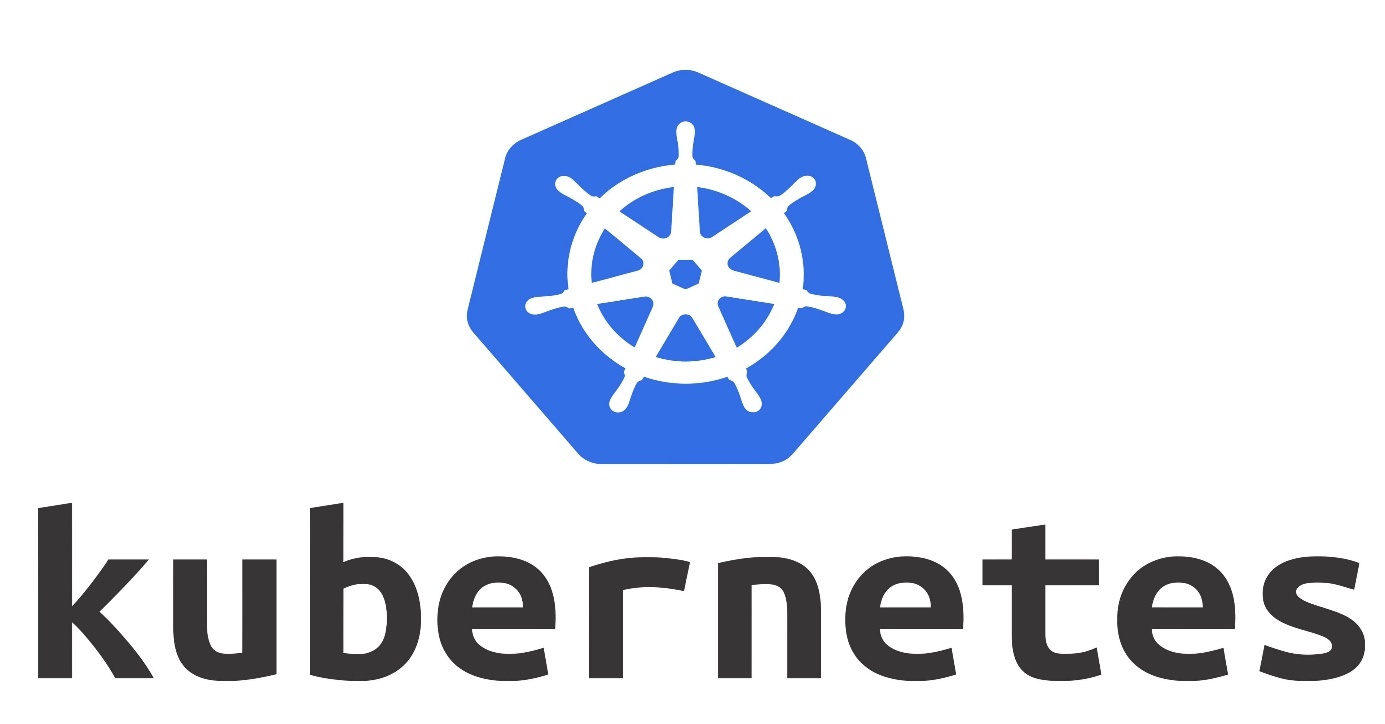
Cuối cùng, Registry là nơi lưu trữ và phân phối các Docker image. Docker sử dụng mặc định là Docker Hub, một kho lưu trữ công khai, nhưng người dùng hoàn toàn có thể thiết lập registry riêng tư. Khi cần một image, Docker daemon sẽ kiểm tra kho cục bộ trước, nếu không có sẽ tiến hành pull từ Registry về. Các image lưu trữ tại Registry có thể là ứng dụng mã nguồn mở phổ biến (như NGINX, Redis, PostgreSQL) hoặc image được người dùng tự xây dựng và đẩy lên (push) từ hệ thống cục bộ.

### Kubernetes

#### Giới thiệu chung về Kubernetes



Hình .Biểu tượng (logo) của Kubernetes



Hình .Biểu tượng (logo) của Kubernetes

Kubernetes (viết tắt là K8s) là một hệ thống điều phối container mã nguồn mở, được phát triển ban đầu bởi Google và hiện được duy trì bởi tổ chức CNCF (Cloud Native Computing Foundation). Kubernetes giúp tự động hóa việc triển khai, mở rộng, cân bằng tải, giám sát và quản lý vòng đời của các container trong môi trường sản suất.

Kubernetes có thể hiểu ngắn gọn qua hai vai trò chính:

* Thứ nhất, Kubernetes hoạt động như một cụm hạ tầng (cluster) cung cấp tài nguyên máy tính để triển khai ứng dụng container hóa.
* Thứ hai, Kubernetes đóng vai trò điều phối, quản lý và duy trì trạng thái hoạt động ổn định của các ứng dụng đã triển khai.

Để triển khai một ứng dụng trên Kubernetes theo mô hình Microservice, quy trình tiêu chuẩn thường bao gồm các bước sau:

* Thiết kế ứng dụng thành các thành phần độc lập theo mô hình Microservice.
* Đóng gói mỗi microservice vào một container riêng biết.
* Triển khai container đó vào trong một Pod trong Kubernetes.
* Sử dụng các đối tượng điều khiển để triển khai và quản lý các Pod đó.

Các tính năng nổi bật của Kubernetes bao gồm:

* Quản lý cụm máy chủ (cluster) với nhiều node.
* Tự động Scale, cân bằng tải và tự phục hổi (self-healing).
* Triển khai ứng dụng dạng rolling update không downtime.
* Dễ cấu hình, mở rộng và tích hợp CI/CD.

#### Kiến trúc hệ thống Kubernetes.

Một cụm Kubernetes (Kubernetes Cluster) được chia làm hai phần chính:

* Control Plane: Điều phối toàn bộ hệ thống
* Worker Nodes: Thực hiện chạy các container ứng dụng.

Control Plane là trung tâm chỉ huy của Kubernetes, chịu trách nhiệm giao tiếp với người dùng, theo dõi trạng thái hệ thống, ra quyết định và điều phối các Pod. Control Plane gồm các thành phần:

* API Server: Là điểm đầu mối cho tất cả các lệnh và yêu cầu giao tiếp với kubernetes. Tất cả các thao tác đều thông qua API Server. Nó cung cấp RESTful API và xác thực tất cả yêu cầu.
* etcd: Là hệ thống lưu trữ phân tán dạng Key-Value, lưu toàn bộ trạng thái cấu hình của cụm Kubernetes. Nó là một phần quan trọng trong việc duy trì tính nhất quán của Cluster.
* Controller Manager: Là tiến trình chạy nền thực hiện giám sát các đối tượng trong hệ thống. Controller Manager sẽ liên tục kiểm tra trạng thái thực tế so với trạng thái mong muốn đã được khai báo, từ đó thực hiện các hành động cần thiết để đồng bộ.
* Scheduler: Là thành phần chịu trách nhiệm phân phối các Pod mới đến các node phù hợp, dựa trên tài nguyên khả dụng, quy tắc chống lặp, ưu tiên hiệu suất và nhiều yếu tố khác. Scheduler không trực tiếp khởi chạy Pod mà chỉ quyết định node nào sẽ đảm nhận.

Worker Node là nơi thực thi thực tế các container ứng dụng, do đó Control Plane giao nhiệm vụ. Mối node bao gồm ba thành phần chính:

* Kubelet: Là một phần tử chạy trên mỗi worker node, có nhiệm vụ tiếp nhận từ API Server và quản lý vòng đời các Pod trên node đó. Kubelet còn báo cáo lại trạng thái node và container định kỳ.
* Container Runtime: Là công cụ thực thế dùng để chạy container. Nó chịu trách nhiệm kéo image, khởi động và dừng container.
* Kube-proxy: Là thành phần đảm bảo kết nối mạng, định tuyến và cân bằng tải các dịch vụ giữa các Pod, các node trong Cluster.

#### Các khái niệm cốt lõi

* Pod và Container: Pod là đơn vị nhỏ nhất mà Kubernetes quản lý. Nó là một lớp bao bọc xung quanh một hoặc nhiều container. Các container trong cùng một Pod chia sẻ chung tài nguyên mạng và lưu trữ, cho phép chúng giao tiếp dễ dàng với nhau.
* Service: Cung cấp một điểm truy cập mạng ổn định cho một nhóm các Pod. Vì Pod có thể bị xóa và tạo lại, địa chỉ của chúng không ổn định. Service giải quyết vấn đề này bằng cách cung cấp một địa chỉ IP và tên DNS duy nhất, không đổi. Kubernetes sẽ tự động cân bằng tải các yêu cầu đến Service tới các Pod khỏe mạnh tương ứng.
* Hệ thống DNS nội bộ: Kubernetes cung cấp một hệ thống DNS riêng bên trong cụm. Điều này cho phép các dịch vụ có thể gọi nhau thông qua tên thay vì phải biết địa chỉ IP, giúp hệ thống trở nên linh hoạt và dễ quản lý hơn.

#### K3s là gì ?

Tuy Kubernetes có rất nhiều lợi ích và tính năng, cài đặt và vận hành một cụm Kubernetes tiêu chuẩn đòi hỏi tài nguyên lớn và cấu hình phức tạp – không phù hợp cho môi trường thử nghiệm, học tập hoặc triển khai trong hệ thống hạ tầng bare-metal (máy thật hoặc máy ảo nội bộ).

K3s là phiên bản rút gọn (lightweight) của Kubernetes, được phát triển bởi Rancher Labs. Nó được thiết kế để trở thành một bản phân phối gọn nhẹ, dễ cài đặt và tối ưu cho các môi trường có tài nguyên hạn chế, hoạt động hiệu quả trên các hệ thống tài nguyên thấp như máy ảo cá nhân.



Hình . Biểu tượng của K3s



Hình . Biểu tượng của K3s

K3s tuân thủ đầy đủ tiêu chuẩn Kubernetes API, có thể dùng kubectl, Helm, hoặc các công cụ tương ứng với Kubernetes. Sự khác biệt chính nằm ở việc K3s đã loại bỏ các thành phần không cốt lõi và thay thế etcd bằng một cơ sở dữ liệu SQLite gọn nhẹ hơn.

Kiến trúc cụm K3s có thể gồm:

* 1 node master (control plane): Chịu trách nhiệm API Server, Scheduler, Controller, Cluster Store.
* 1 hoặc nhiều node worker: Nơi triển khai và chạy các ứng dụng (Pod).

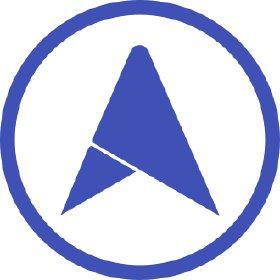
#### Tích hợp với MetalLB và NFS

Trong môi trường bare-metal, K3s không có sẵn LoadBalancer.

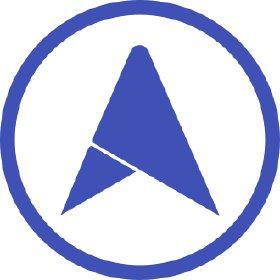
Do đó, trong đề tài, K3s được tích hợp với:

* MetalLB: cung cấp IP tĩnh dạng LoadBalancer cho service, giúp expose ra ngoài mạng LAN.
* NFS Server: để chia sẻ volume lưu trữ giữa các Pod và node.

### MetalLB – LoadBalancer cho môi trường bare-metal



Hình . Biểu tượng của MetalLB



Hình . Biểu tượng của MetalLB

Trong Kubernetes, để một ứng dụng có thể được truy cập từ bên ngoài cụm, người ta thường khai báo một Service loại LoadBalancer. Trên các nền tảng đám mây như AWS hay GCP, nhà cung cấp cloud sẽ tự động cấp phát một địa chỉ IP công cộng cho Service đó. Tuy nhiên, khi triển khai Kubernetes trên môi trường bare-metal (máy chủ vật lý hoặc máy ảo tự quản lý), không có một thực thể bên ngoài nào làm công việc này. Kết quả là, các Service loại LoadBalancer sẽ bị kẹt vĩnh viễn ở trạng thái "Pending" và không bao giờ nhận được địa chỉ IP.

MetalLB được phát triển để giải quyết chính xác vấn đề này. Nó hoạt động như một bộ cân bằng tải mạng được cài đặt bên trong cụm, giúp các Service loại LoadBalancer có thể hoạt động được như trên môi trường đám mây.

Sau khi cài đặt, người quản trị sẽ cấu hình cho MetalLB một dải địa chỉ IP (IP range) từ mạng LAN nội bộ. Khi một Service loại LoadBalancer được tạo, MetalLB sẽ tự động chọn một địa chỉ IP từ dải này và gán cho Service đó. MetalLB có hai chế độ hoạt động chính:

* Chế độ Layer 2 (Sử dụng ARP/NDP): Đây là chế độ đơn giản và phổ biến nhất. Trong chế độ này, một node trong cụm sẽ nhận trách nhiệm "trả lời" cho địa chỉ IP của Service. Nó sẽ quảng bá địa chỉ MAC của mình trên mạng LAN thông qua giao thức ARP (cho IPv4) hoặc NDP (cho IPv6). Các thiết bị khác trong cùng mạng sẽ biết rằng để gửi lưu lượng đến địa chỉ IP đó, chúng cần gửi đến node đang quảng bá. Nếu node đó gặp sự cố, một node khác sẽ tự động tiếp quản vai trò này, đảm bảo tính sẵn sàng cao.
* Chế độ BGP (Border Gateway Protocol): Đây là chế độ nâng cao dành cho các hệ thống mạng phức tạp hơn. Trong chế độ này, MetalLB sẽ thiết lập một phiên BGP với các router trong mạng của bạn để quảng bá các tuyến đường đến địa chỉ IP của Service. Điều này cho phép cân bằng tải thực sự trên nhiều node và cung cấp khả năng phục hồi tốt hơn.

Mặc dù MetalLB là một giải pháp hiệu quả và nhẹ cho môi trường Kubernetes bare-metal, nhưng nó vẫn tồn tại một số hạn chế đáng chú ý:

* Không hỗ trợ kiểm soát phân phối tải nâng cao.
* Không có tính năng tự động scale như cloud LoadBalancer.
* Phụ thuộc vào cấu hình IP tĩnh và mạng LAN nội bộ.
* Không có cơ chế bảo mật nâng cao tích hợp sẵn.

Đối với phạm vi của đề tài, việc sử dụng MetalLB ở chế độ Layer 2 là một giải pháp hiệu quả và nhẹ nhàng. Nó cho phép các dịch vụ trong cụm K3s nhận được một địa chỉ IP ổn định, có thể truy cập từ bên ngoài mạng LAN, mô phỏng chính xác cách một LoadBalancer trên đám mây hoạt động. Tuy nhiên, cần lưu ý rằng MetalLB chủ yếu giải quyết bài toán cấp phát IP và không cung cấp các tính năng phân phối tải nâng cao hay tự động mở rộng quy mô như các giải pháp thương mại.

### NFS – Hệ thống chia sẻ tệp cho Kubernetes

Một trong những đặc tính cơ bản của container là chúng có vòng đời tạm thời (ephemeral). Điều này có nghĩa là khi một Pod bị xóa, khởi động lại hoặc di chuyển sang một node khác, tất cả dữ liệu được tạo ra bên trong container đó cũng sẽ bị mất vĩnh viễn. Đây là một vấn đề nghiêm trọng đối với các ứng dụng cần lưu trữ trạng thái (stateful applications), chẳng hạn như cơ sở dữ liệu, hoặc các ứng dụng cần lưu trữ file do người dùng tải lên (hình ảnh, tài liệu).

NFS (Network File System) là một giải pháp cho phép giải quyết vấn đề này bằng cách cung cấp một hệ thống lưu trữ tập trung, độc lập với vòng đời của các Pod.

NFS là một giao thức cho phép một thư mục trên một máy chủ (NFS Server) có thể được truy cập và sử dụng như một ổ đĩa cục bộ bởi nhiều máy tính khác qua mạng. Kubernetes tích hợp với NFS thông qua hai đối tượng trừu tượng là PersistentVolume (PV) và PersistentVolumeClaim (PVC).

NFS có một số ưu điểm:

* Thiết lập: Đầu tiên, người quản trị sẽ thiết lập một máy chủ NFS riêng và tạo một thư mục chia sẻ trên đó.Dễ thiết lập, hoạt động tốt trong môi trường nội bộ.
* Khai báo PersistentVolume (PV): Người quản trị sau đó sẽ khai báo một đối tượng PV trong Kubernetes. PV này không phải là vùng lưu trữ thực tế, mà là một "con trỏ" trỏ đến thư mục chia sẻ trên máy chủ NFS. Nó mô tả các thông tin về vùng lưu trữ như dung lượng, chế độ truy cập (ví dụ: ReadWriteMany - cho phép nhiều Pod cùng đọc/ghi).
* Yêu cầu lưu trữ với PersistentVolumeClaim (PVC): Khi một ứng dụng (thông qua Pod) cần lưu trữ dữ liệu, nó sẽ không yêu cầu trực tiếp một PV. Thay vào đó, nó tạo ra một PersistentVolumeClaim (PVC), giống như một "yêu cầu lưu trữ". PVC này sẽ định nghĩa các yêu cầu như: "Tôi cần 5GB dung lượng với chế độ truy cập ReadWriteMany".
* Binding (Gắn kết): Kubernetes sẽ tự động tìm một PV phù hợp có thể đáp ứng được các yêu cầu trong PVC và "gắn kết" (bind) chúng lại với nhau.
* Mount vào Pod: Cuối cùng, Pod sẽ khai báo rằng nó muốn sử dụng PVC đó. Kubernetes sẽ tự động "mount" (gắn) thư mục NFS từ xa vào hệ thống tệp tin của container. Giờ đây, mọi dữ liệu mà ứng dụng ghi vào thư mục này sẽ được lưu trữ an toàn trên máy chủ NFS, ngay cả khi Pod bị xóa.

NFS là một giải pháp lưu trữ bền vững đơn giản và hiệu quả cho môi trường Kubernetes bare-metal.

* Lưu trữ bền vững: Đảm bảo dữ liệu không bị mất khi Pod khởi động lại.
* Chia sẻ dữ liệu: Cho phép nhiều Pod (kể cả khi chúng chạy trên các Worker Node khác nhau) có thể cùng đọc và ghi vào một vùng dữ liệu chung. Điều này rất hữu ích cho các ứng dụng cần chia sẻ file cấu hình, log, hoặc tài sản (assets).
* Dễ thiết lập: So với các giải pháp lưu trữ phân tán phức tạp khác, việc thiết lập một máy chủ NFS và tích hợp nó vào Kubernetes tương đối đơn giản.

### GitHub Actions

GitHub Actions là một nền tảng tích hợp sẵn trong GitHub, cho phép người dùng tự động hóa quy trình tích hợp liên tục (CI) và phân phối liên tục (CD). Với GitHub Actions, bạn có thể dễ dàng thiết lập các workflow để tự động build, kiểm thử và triển khai ứng dụng mỗi khi có thay đổi trong mã nguồn. Điều này giúp rút ngắn thời gian phát hành phần mềm và nâng cao độ tin cậy trong quy trình phát triển.



Hình . Biểu tượng (logo) của GitHub Actions



Hình . Biểu tượng (logo) của GitHub Actions

Mỗi workflow trong GitHub Actions được định nghĩa bằng tệp YAML, trong đó mô tả các sự kiện kích hoạt (trigger), các bước thực hiện, điều kiện rẽ nhánh, môi trường chạy và nhiều tuỳ chọn nâng cao khác. Ví dụ: khi có một pull request, hệ thống có thể tự động chạy quy trình kiểm thử; hoặc khi mã được merge vào nhánh chính (main/master), hệ thống có thể tự động triển khai lên môi trường staging hoặc production.

Một điểm nổi bật của GitHub Actions là không chỉ phục vụ cho mục tiêu DevOps, mà còn hỗ trợ nhiều tác vụ quản trị kho mã khác. Chẳng hạn, bạn có thể thiết lập workflow để tự động gán nhãn (label) cho các issue, gửi thông báo, cập nhật tài liệu, hoặc chạy mã tùy chỉnh bất cứ khi nào xảy ra một sự kiện trong kho lưu trữ.

GitHub cung cấp sẵn các runner (máy ảo để chạy workflow) với nhiều nền tảng khác nhau như Linux, Windows và macOS. Ngoài ra, người dùng có thể tự lưu trữ runner riêng để kiểm soát chặt chẽ hơn môi trường chạy – rất hữu ích cho các tổ chức lớn hoặc yêu cầu bảo mật cao.

Trong đề tài này, GitHub Actions đóng vai trò trung tâm trong quy trình DevOps, giúp tự động hóa toàn bộ pipeline CI/CD: từ khi lập trình viên cập nhật mã nguồn, hệ thống sẽ tự động kiểm thử, build image Docker, đẩy lên registry, và triển khai lên cụm Kubernetes K3s. Nhờ vậy, quy trình triển khai trở nên nhất quán, tiết kiệm thời gian và giảm thiểu sai sót.

### Trivy: Quét lỗ hổng của Docker Image

Công dụng và Tầm quan trọng: Một Docker image không chỉ chứa mã nguồn của ứng dụng mà còn kế thừa từ một image cơ sở (base image) và bao gồm hàng loạt các thư viện, gói phần mềm của bên thứ ba. Bất kỳ thành phần nào trong số này cũng có thể chứa các lỗ hổng bảo mật đã được công bố. Trivy là một công cụ giúp tự động hóa việc phát hiện các rủi ro này, trả lời cho câu hỏi then chốt: "Docker image mà chúng ta sắp triển khai có chứa lỗ hổng nào có thể bị khai thác hay không?".



Hình .Biểu tượng của Trivy



Hình .Biểu tượng của Trivy

Cách hoạt động chi tiết: Trivy là một công cụ quét lỗ hổng mã nguồn mở, được thiết kế để hoạt động nhanh và đơn giản, rất phù hợp để tích hợp vào các pipeline CI/CD. Khi thực thi, nó sẽ:

* Phân tích các lớp (layers) của image: Trivy đọc và phân tích hệ thống tệp tin của Docker image.
* Xác định các gói phần mềm: Nó xác định danh sách tất cả các gói phần mềm đã được cài đặt, bao gồm cả các gói của hệ điều hành (ví dụ: glibc, openssl trên Alpine Linux) và các thư viện phụ thuộc của ứng dụng (ví dụ: các gói npm của Node.js, pip của Python).
* Đối chiếu với cơ sở dữ liệu lỗ hổng: Trivy sau đó sẽ đối chiếu danh sách các gói này với cơ sở dữ liệu về lỗ hổng của nó, vốn được tổng hợp từ nhiều nguồn công khai uy tín (như National Vulnerability Database - NVD, các cơ sở dữ liệu của nhà cung cấp hệ điều hành...).
* Tạo báo cáo: Cuối cùng, nó tạo ra một báo cáo chi tiết, liệt kê các lỗ hổng tìm thấy, mã định danh của chúng (ví dụ: CVE-2021-44228), mức độ nghiêm trọng (Critical, High, Medium, Low), và phiên bản phần mềm bị ảnh hưởng.

Vai trò cơ bản trong đề tài: Trong đề tài này, Trivy được sử dụng như một "cổng chất lượng" (quality gate) tự động và bắt buộc trong pipeline CI/CD. Nó được cấu hình để quét image ngay sau khi được build. Nếu phát hiện bất kỳ lỗ hổng nào ở mức độ "Nghiêm trọng" (Critical), pipeline sẽ được thiết lập để tự động dừng lại và báo lỗi. Điều này hiện thực hóa triết lý "Shift-Left", buộc nhà phát triển phải nhận thức và khắc phục các vấn đề bảo mật ngay lập tức, thay vì để chúng tồn tại và có nguy cơ bị triển khai ra môi trường sản xuất.

### Prometheus: Hệ thống thu thập và lưu trữ số liệu

Prometheus là một hệ thống mã nguồn mở, đã trở thành tiêu chuẩn de facto cho việc giám sát trong thế giới Cloud Native. Chức năng chính của nó là thu thập và lưu trữ các số liệu (metrics) dưới dạng chuỗi thời gian (time-series). Các metrics này là bất kỳ dữ liệu số nào có thể đo lường được, ví dụ: mức sử dụng CPU của một tiến trình, dung lượng bộ nhớ còn trống trên một máy chủ, số lượng yêu cầu HTTP thất bại mỗi giây, hoặc độ trễ của một API.

A logo of a fire

AI-generated content may be incorrect.

Hình . Biểu tượng của Prometheus

A logo of a fire

AI-generated content may be incorrect.

Hình . Biểu tượng của Prometheus

Prometheus hoạt động chủ yếu theo mô hình "kéo" (pull). Nó được cấu hình để định kỳ kết nối đến các "điểm cuối" (endpoints) mà các ứng dụng hoặc thành phần hệ thống (gọi là exporters) cung cấp. Tại các điểm cuối này, Prometheus sẽ thu thập các số liệu mới nhất và lưu chúng vào cơ sở dữ liệu chuỗi thời gian (TSDB) của mình. Kiến trúc này giúp Prometheus trở nên linh hoạt và đáng tin cậy. Nó cũng đi kèm với một ngôn ngữ truy vấn mạnh mẽ gọi là PromQL, cho phép người dùng thực hiện các phân tích phức tạp và thiết lập các quy tắc cảnh báo (alerting) tinh vi.

### Grafana: Nền tảng trực quan hóa và phân tích

Dữ liệu số liệu thô trong Prometheus, mặc dù rất giá trị, nhưng lại khó hiểu đối với con người nếu chỉ nhìn vào các con số. Grafana là một nền tảng mã nguồn mở chuyên dụng để trực quan hóa dữ liệu, giúp biến những con số phức tạp đó thành các biểu đồ, đồ thị và bảng điều khiển (dashboards) đẹp mắt, tương tác và dễ hiểu.

A logo with a spiral in the center

AI-generated content may be incorrect.

Hình .Biểu tượng của Grafana

A logo with a spiral in the center

AI-generated content may be incorrect.

Hình .Biểu tượng của Grafana

Grafana có khả năng kết nối với rất nhiều nguồn dữ liệu khác nhau, trong đó Prometheus là một trong những nguồn phổ biến nhất. Sau khi kết nối, người dùng có thể sử dụng giao diện của Grafana để xây dựng các dashboard. Mỗi dashboard bao gồm nhiều panel, và mỗi panel hiển thị dữ liệu từ một hoặc nhiều truy vấn PromQL. Grafana cho phép người dùng tùy chỉnh mọi thứ, từ loại biểu đồ, màu sắc, cho đến các chú thích, giúp tạo ra một "phòng điều khiển" trung tâm để theo dõi toàn bộ hệ thống.

Grafana được sử dụng để hiển thị các thông số cơ bản về "sức khỏe" của cụm K3s (CPU, RAM, Network) thông qua một dashboard có sẵn, giúp người quản trị có cái nhìn tổng quan nhanh chóng về tình trạng hoạt động của hạ tầng.

# THIẾT KẾ HỆ THỐNG

## Bối cảnh và vấn đề

Một đội ngũ phát triển phần mềm được giao nhiệm vụ xây dựng một hệ thống website thương mại điện tử hiện đại theo kiến trúc Microservices. Hệ thống gồm nhiều thành phần độc lập như: dịch vụ người dùng, dịch vụ sản phẩm, dịch vụ đặt hàng và thanh toán. Mỗi thành phần được phát triển, kiểm thử và triển khai riêng biệt.

Trong bối cảnh đó, việc triển khai thủ công từng dịch vụ, theo dõi trạng thái hoạt động, kiểm thử tích hợp và đảm bảo đồng bộ phiên bản sẽ gây ra rất nhiều khó khăn, đặc biệt khi nhóm mở rộng quy mô hoặc cần triển khai thường xuyên.

Các vấn đề chính được đặt ra gồm:

* Làm thế nào để tự động hóa quy trình triển khai cho từng Microservice?
* Làm nào để quản lý tính nhất quán môi trường triển khai nhanh, chính xác và lặp lại được.
* Làm thế nào để đảm bảo mỗi lần cập nhật đều có thể triển khai nhanh, chính xác?
* Làm thế nào để triển khai hệ thống lên cụm nội bộ, không phụ thuộc vào các dịch vụ cloud thương mại?

## Yêu cầu giải pháp

Để giải quyết vấn đề trên, đề tài cần xây dựng một giải pháp kỹ thuật đáp ứng các yêu cầu sau:

* Về ứng dụng: Phải được thiết kế theo kiến trúc Microservices, bao gồm các dịch vụ chức năng: Frontend (Giao diện người dùng), User Service (Quản lý thông tin người dùng, các chức năng đăng nhập, đăng ký), Product Service (Quản lý thông tin sản phẩm), Order Service (Quản lý các đơn hàng) và Payment Service (Dịch vụ thanh toán).
* Về Quy trình: Phải thiết lập được một quy trình DevOps tự động hóa hoàn toàn việc xây dựng và triển khai các dịch vụ.
* Về tính nhất quán: Sử dụng công nghệ container (Docker) để đóng gói ứng dụng, đảm bảo tính nhất quán môi trường.
* Về hạ tầng: Triển khai trên một hệ thống điều phối container để quản lý ứng dụng một cách hiệu quả.

## Phân tích và đưa ra kiến trúc tổng quan cho quy trình

Hệ thống được triển khai theo mô hình kiến trúc Microservices,trong đó mỗi chức năng chính được tách thành các dịch vụ nhỏ độc lập, đóng gói bằng container và triển khai trên môi trường điều phối Kubernetes. Các thành phần chính bao gồm:

* Frontend (ReactJS): giao diện người dùng cho toàn hệ thống, gọi API từ backend thông qua Nginx Gateway.
* User Service: xử lý đăng ký, đăng nhập người dùng.
* Product Service: quản lý và hiển thị danh sách sản phẩm.
* Order Service: xử lý đơn hàng từ người dùng.
* Payment Service: mô phỏng quy trình thanh toán đơn hàng bằng phương thức momo.
* Dữ liệu: sử dụng file CSV ứng với mỗi service để lưu trữ dữ liệu, đóng vai trò như một hệ cơ sở dữ liệu đơn giản để thử nghiệm.

Tất cả mã nguồn của hệ thống được lưu trữ tập trung trong một repository trên GitHub. Quy trình triển khai DevOps được thiết kế để tự động hóa toàn bộ vòng đời phát triển và triển khai phần mềm, thông qua chuỗi các giai đoạn như sau:

Giai đoạn 1: Phát triển ứng dụng

Các lập trình viên thực hiện xây dựng chức năng cho từng service, đảm bảo hệ thống có thể hoạt động ổn định khi chạy ở môi trường local. Đây là bước chuẩn bị ban đầu để kiểm thử tính năng và xác định giao diện API.

Giai đoạn 2: Đóng gói từng dịch vụ bằng Docker.

Mỗi dịch vụ sẽ có một Dockerfile riêng để xây dựng image tương ứng. Việc container hóa từng microservice là điều bắt buộc để đảm bảo tính nhất quán trong triển khai, đồng thời tách biệt môi trường giữa các service.

Giai đoạn 3: Đẩy Docker image lên Docker Hub.

Sau khi build xong image, hệ thống sẽ đẩy chúng lên Docker Hub — nơi đóng vai trò như kho lưu trữ container trung gian, phục vụ cho quá trình triển khai thực tế ở giai đoạn tiếp theo.

Giai đoạn 4: Thiết lập cụm Kubernetes bằng K3s.

Hệ thống được triển khai trên cụm K3s gồm 3 máy ảo:

* 1 master node: quản lý triển khai toàn bộ cụm Kubernetes.
* 2 worker node: thực thi các Pod chứa các container tương ứng với từng service.

Các Pod sẽ được triển khai từ các file deployment.yaml, quản lý bởi các Deployment Controller của Kubernetes.

Giai đoạn 5: Tự động hóa bằng CI/CD (GitHub Actions)\

GitHub Actions được sử dụng để tự động hóa toàn bộ quy trình triển khai. Mỗi khi lập trình viên đẩy mã nguồn mới lên nhanh main, pipeline sẽ tự động thực hiện các bước:

* Build Docker image cho từng service từ mã nguồn mới.
* Push image lên Docker Hub sử dụng các thông tin xác thực bảo mật.
* Giải mã file kubeconfig từ GitHub Secret để truy cập cụm K3s trên máy ảo.
* Thực hiện triển khai với từng file YAML khai báo deployment và service.
* Theo dõi rollout và trạng thái hệ thống, đảm bảo mọi thành phần được cập nhật thành công.

## Phân tích và thiết kế kiến trúc ứng dụng Microservices

Hệ thống TMĐT mẫu với kiến trúc Microservices được thiết kế với các dịch vụ User Service, Product Service, Order Service, Payment. Mỗi dịch vụ là một ứng dụng Node.js độc lập, được đóng gói trong một container Docker riêng biệt và sử dụng cơ sở dữ liệu là file CSV của riêng mình để thay thế cho cơ sở dữ liệu hoàn chỉnh trong môi trường demo.

Trong kiến trúc Microservices, việc cho phép client (trình duyệt web) giao tiếp trực tiếp qua từng microservice sẽ gây ra nhiều vấn đề: client phải quản lý nhiều địa chỉ IP/cổng, khó xử lý các vấn đề chung như xác thực, giới hạn tần suất truy cập (rate limiting), và làm lộ cấu trúc nội bộ của hệ thống.

Để giải quyết vấn đề này, đề tài áp dụng mẫu thiết kế API Gateway. API Gateway hoạt động như một cổng vào duy nhất cho tất cả các yêu cầu từ client. Nó đóng vai trò là một lớp trung gian, nhận tất cả các yêu cầu, sau đó định tuyến chúng đến các microservice backend phù hợp.

Trong đề tài này, Nginx được sử dụng như một Reverse Proxy để triển khai mẫu API Gateway.

A diagram of a software flow

AI-generated content may be incorrect.

Hình .Sơ đồ kiến trúc ứng dụng, minh họa các Microservices và luồng giao tiếp thông qua API Gateway

A diagram of a software flow

AI-generated content may be incorrect.

Hình .Sơ đồ kiến trúc ứng dụng, minh họa các Microservices và luồng giao tiếp thông qua API Gateway

Trong đề tài, nginx được sử dụng làm công cụ Reverse Proxy, đóng vai trò là API Gateway. Luồng tương tác chi tiết:

* Người dùng truy cập: Người dùng sử dụng trình duyệt để tương tác với Website với Frontend (ReactJS)
* Client-side Request: Khi người dùng thực hiện một hành động, Frontend sẽ tạo một yêu cầu và gửi đến địa chỉ của hệ thống (ví dụ: /api/users/login)
* Chuyển tiếp đến Gateway: Yêu cầu này được gửi đến API Gateway (Nginx), đóng vai trò là điểm vào duy nhất cho hệ thống backend.
* Phân tích và Định tuyến: Gateway phân tích đường dẫn của yêu cầu. Sau khi phân tích sẽ chuyển đến service tương ứng.
* Giao tiếp nội bộ: Việc chuyển tiếp này được thực hiện trong mạng riêng của cụm K3s, API Gateway sẽ gửi yêu cầu đến tên DNS nội bộ của service tương ứng.
* Xứ lý nghiệp vụ: Service nhận được yêu cầu, xử lý logic xác thực người dùng bằng cách truy vấn cơ sở dữ liệu của riêng nó.
* Phản hồi (Response): Sau khi xử lý xong, Service gửi phản hồi người lại cho API Gateway.
* Trả về Client: Cuối cùng, API Gateway nhận phản hồi từ microservice và chuyển tiếp nó trở lại cho ứng dụng Frontend, kết thúc chu trình yêu cầu.

Với kiến trúc này, client chỉ cần biết đến địa chỉ duy nhất của API Gateway, còn taonf bộ cấu trúc phức tạp của các Microservices phía sau được che giấu hoàn toàn, giúp hệ thống trở nên linh hoạt và an toàn hơn.

## Thiết kế kiến trúc hạ tầng.

Hệ thống được triển khai trên một cụm Kubernetes nhẹ (K3s) hoạt động trong môi trường mạng nội bộ (LAN). Hạ tầng được xây dựng hoàn toàn bằng máy ảo tự quản lý, nhằm mô phỏng môi trường thực tế khi triển khai các ứng dụng theo mô hình microservices trên nền tảng container.

A diagram of a computer

AI-generated content may be incorrect.

Hình .Sơ đồ hạ tầng trên một cụm K3s

A diagram of a computer

AI-generated content may be incorrect.

Hình .Sơ đồ hạ tầng trên một cụm K3s

Mô tả các thành phần và luồng hoạt động của hệ thống:

Mạng LAN: Là mạng nội bộ nơi tất cả các máy ảo trong cụm K3s hoạt động. Người dùng cũng kết nối vào mạng LAN này để truy cập dịch vụ. Việc sử dụng mạng LAN giúp đảm bảo độ ổn định và kiểm soát lưu lượng truy cập trong môi trường triển khai.

Cụm K3s (K3s Cluster): Là trung tâm của hạ tầng trên, bao gồm một nhóm các máy ảo hoạt động như một hệ thống thống nhất để chạy ứng dụng.

* 1 Master Node: Đây là nút điều phối chính của cụm K3s, chịu trách nhiệm quản lý trạng thái hệ thống, lịch trình Pod và giám sát tài nguyên. Ngoài vai trò Control Plane, node này còn đồng thời đóng vai trò là NFS Server, cung cấp vùng lưu trữ dùng chung cho các worker node. Điều này giúp giảm chi phí tài nguyên và đảm bảo tính nhất quán của dữ liệu trong toàn hệ thống.
* Worker Node 1 & Worker Node 2: Hai node này chịu trách nhiệm chính trong việc chạy các container ứng dụng. Các Pod sẽ được tự động phân bổ và khởi chạy trên các worker node này dựa trên tài nguyên khả dụng và chính sách phân phối của Kubernetes.

MetalLB: Đây là thành phần cung cấp dịch vụ Load Balancer trong môi trường mạng nội bộ, giúp người dùng có thể truy cập hệ thống thông qua một địa chỉ IP duy nhất mà MetalLB cung cấp. MetalLB đóng vai trò điều phối lưu lượng truy cập từ bên ngoài vào cụm K3s.

Luồng truy cập của người dùng:

* Người dùng từ máy tính mở trình duyệt truy cập vào một địa chỉ duy nhất do MetalLB cung cấp. Đây là cổng vào cho toàn bộ hệ thống
* MetalLB tiếp nhận yêu cầu và chuyển tiếp đến Pod API Gateway (thường là Nginx hoặc một dịch vụ reverse proxy) đang chạy trên một trong các worker node.
* API Gateway thực hiện chức năng định tuyến yêu cầu đến các microservice backend tương ứng (như User Service, Product Service, Order Service, Payment).
* Các dịch vụ backend sẽ xử lý yêu cầu và trả về phản hồi thông qua API Gateway về phía người dùng.

Luồng lưu trữ và truy xuất dữ liệu:

* Trong quá trình xử lý, nếu một Pod cần ghi hoặc đọc dữ liệu bền vững (ví dụ: các tệp CSV), nó sẽ thực hiện truy cập đến hệ thống tệp thông qua giao thức mạng nội bộ.
* Yêu cầu truy xuất sẽ được chuyển đến **NFS Server** đang hoạt động trên Master Node.
* NFS Server sẽ xử lý yêu cầu và cung cấp vùng lưu trữ chia sẻ cho các Pod, đảm bảo rằng dữ liệu không bị mất mát ngay cả khi Pod gặp sự cố hoặc bị khởi động lại.

## Thiết kế quy trình CI/CD với GitHub Actions

Quy trình CI/CD là một phần cốt lõi trong việc áp dụng DevOps vào hệ thống, giúp tự động hóa toàn bộ quá trình xây dựng, kiểm thử, kiểm tra bảo mật và triển khai ứng dụng. Trong đồ án này, quy trình CI/CD được xây dựng bằng GitHub Actions và triển khai trực tiếp vào cụm Kubernetes nhẹ K3s.

A screenshot of a computer

AI-generated content may be incorrect.

Hình .Sơ đồ kiến trúc CI/CD của GitHub Actions với cụm K3s

A screenshot of a computer

AI-generated content may be incorrect.

Hình .Sơ đồ kiến trúc CI/CD của GitHub Actions với cụm K3s

Khi lập trình viên hoàn tất chỉnh sửa mã nguồn và thực hiện thao tác đẩy mã (git push) lên nhánh main trên GitHub, một workflow CI/CD sẽ được tự động kích hoạt thông qua GitHub Actions. Đây là khởi đầu cho toàn bộ quy trình kiểm thử, đóng gói, kiểm tra bảo mật và triển khai ứng dụng.

Ngay khi workflow được kích hoạt, GitHub Actions Runner sẽ thực hiện checkout mã nguồn từ repository để sử dụng trong quá trình xử lý. Hệ thống tiếp theo sẽ chuẩn bị môi trường cần thiết, bao gồm cài đặt các công cụ như Trivy – công cụ quét bảo mật cho Docker Image – và các phần mềm phụ thuộc cần thiết cho quá trình test và build.

Sau khi môi trường đã được chuẩn bị, hệ thống sẽ thực hiện chạy các Unit Test. Đây là bước quan trọng giúp phát hiện lỗi trong các chức năng riêng lẻ của ứng dụng, đảm bảo rằng logic chương trình vẫn hoạt động đúng đắn sau mỗi lần cập nhật mã nguồn. Nếu bất kỳ test nào không đạt, workflow sẽ dừng lại, giúp lập trình viên nhận biết và sửa lỗi kịp thời trước khi ảnh hưởng đến môi trường triển khai.

Khi các Unit Test đều vượt qua thành công, quá trình đóng gói ứng dụng thành Docker Image sẽ được tiến hành. Docker Image tạo ra sẽ chứa toàn bộ mã nguồn và môi trường chạy cần thiết để triển khai ứng dụng một cách đồng nhất.

Tiếp theo, hệ thống sẽ dùng Trivy để quét bảo mật Docker Image. Nếu Trivy phát hiện ra các lỗ hổng bảo mật nghiêm trọng, quy trình sẽ tự động dừng lại để đảm bảo rằng image không an toàn sẽ không được đưa vào hệ thống.

Nếu image vượt qua quá trình quét bảo mật, hệ thống sẽ đẩy Docker Image lên Docker Hub, kho lưu trữ công khai hoặc riêng tư để phục vụ triển khai. Sau đó, GitHub Actions sẽ sử dụng lệnh kubectl apply -f để triển khai ứng dụng lên cụm Kubernetes nhẹ K3s thông qua các tệp YAML khai báo.

Khi cụm K3s nhận được lệnh triển khai, nó sẽ kéo Docker Image mới nhất từ Docker Hub về và cập nhật lại các Pod đang chạy, đảm bảo rằng hệ thống luôn sử dụng phiên bản ứng dụng mới nhất mà không ảnh hưởng đến các thành phần khác.

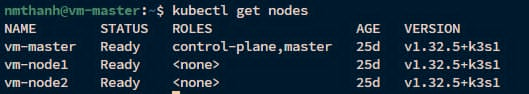
# KẾT QUẢ TRIỂN KHAI HỆ THỐNG

## Kết quả xây dựng và cấu hình hạ tầng

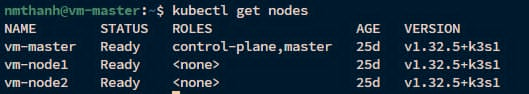
Để hiện thực hóa quy trình DevOps, bước đầu tiên và quan trọng nhất là xây dựng một nền tảng hạ tầng vững chắc, linh hoạt và có khả năng tự động hóa. Trong khuôn khổ đề tài, một hạ tầng Cloud Native hoàn chỉnh, mô phỏng môi trường production thực tế, đã được xây dựng thành công trên nền tảng các máy ảo tự quản lý. Đây là lớp nền tảng cốt lõi, cung cấp các tài nguyên về tính toán, mạng và lưu trữ cần thiết để quy trình CI/CD có thể triển khai và vận hành ứng dụng Microservices một cách tự động.

Hoàn thành cụm Kubernetes với K3s – Nền tảng điều phối tập trung:

* Hiện trạng: Một cụm K3s bao gồm 1 master node và 2 worker nodes đã được thiết lập và xác minh hoạt động ổn định. Kết quả lệnh kubectl get nodes cho thấy tất cả các node đều ở trạng thái "Ready", chứng tỏ chúng đã gia nhập cụm thành công và Control Plane có thể giao tiếp, điều phối các tác vụ.
* Ý nghĩa và Tầm quan trọng: Kết quả này đã biến 3 máy ảo riêng lẻ thành một nền tảng tài nguyên tính toán hợp nhất. Thay vì phải suy nghĩ về việc triển khai ứng dụng lên từng máy chủ cụ thể, giờ đây chúng ta có thể xem toàn bộ cụm như một "siêu máy tính" duy nhất. Đây là điều kiện tiên quyết cho việc tự động hóa, cho phép Kubernetes tự do phân bổ, quản lý và phục hồi các container ứng dụng trên toàn bộ tài nguyên có sẵn, tạo ra một môi trường triển khai linh hoạt và có khả năng chịu lỗi.



Hình .Kết quả cụm K3s



Hình .Kết quả cụm K3s

Hệ thống có khả năng cấp phát IP ngoại vi – Mở cổng giao tiếp ra bên ngoài:

* Hiện trạng: MetalLB đã được cấu hình và tích hợp thành công vào cụm, giải quyết được bài toán kết nối mạng cho môi trường bare-metal. Kết quả là các dịch vụ (services) loại LoadBalancer có thể tự động nhận được một địa chỉ IP ổn định từ dải IP đã định nghĩa trong mạng LAN.
* Ý nghĩa và Tầm quan trọng: Trong môi trường Kubernetes tự quản lý, việc đưa ứng dụng ra ngoài cho người dùng truy cập là một thách thức. Nếu không có MetalLB, các dịch vụ sẽ bị kẹt ở trạng thái "Pending" và không thể truy cập được. Việc tích hợp thành công MetalLB không chỉ cho phép người dùng bên ngoài có thể truy cập vào ứng dụng một cách dễ dàng mà còn mô phỏng chính xác cách các bộ cân bằng tải trên nền tảng đám mây hoạt động. Điều này tạo ra một điểm vào (entrypoint) duy nhất và chuyên nghiệp cho hệ thống, là một thành phần thiết yếu cho bất kỳ ứng dụng web nào.



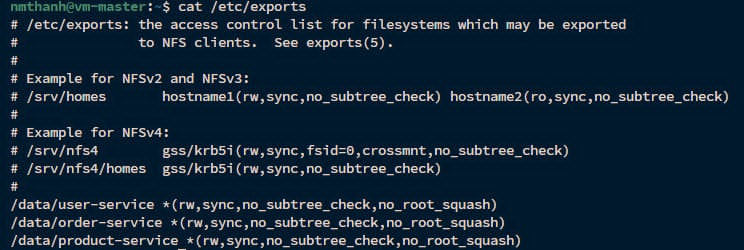
Hình .Kết quả khi sử dụng MetalLB



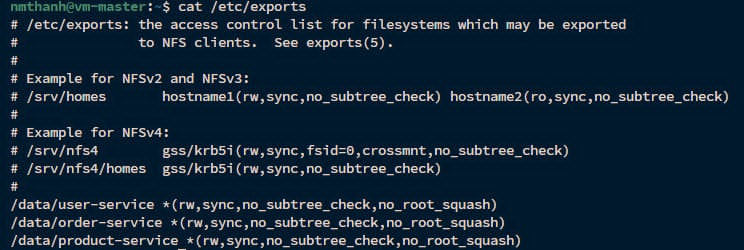
Hình .Kết quả khi sử dụng MetalLB

Thiết lập thành công vùng lưu trữ bền vững – Đảm bảo an toàn dữ liệu

* Hiện trạng: Một NFS Server đã được cấu hình để cung cấp một vùng lưu trữ dùng chung qua mạng, và được tích hợp vào Kubernetes thông qua các đối tượng PersistentVolume và PersistentVolumeClaim.
* Ý nghĩa và Tầm quan trọng: Kết quả này giải quyết được vấn đề cố hữu về vòng đời tạm thời của container. Nếu không có giải pháp lưu trữ bền vững, mọi dữ liệu quan trọng của ứng dụng (như thông tin người dùng, danh sách sản phẩm, đơn hàng) sẽ bị mất vĩnh viễn mỗi khi một Pod được di chuyển, khởi động lại hay cập nhật. Bằng cách sử dụng NFS, dữ liệu giờ đây được lưu trữ an toàn trên một hệ thống độc lập, đảm bảo tính toàn vẹn và bền vững cho các ứng dụng có trạng thái (stateful), giúp ứng dụng có thể hoạt động ổn định và đáng tin cậy như trong thực tế.



Hình .Các thư mục được dùng chung



Hình .Các thư mục được dùng chung

## Kết quả đóng gói và phân phối ứng dụng

Đây là bước then chốt, biến mã nguồn từ trạng thái tĩnh thành các "hiện vật" (artifacts) phần mềm có thể triển khai được. Quá trình này đảm bảo tính nhất quán và tạo ra một "nguồn sự thật duy nhất" cho các phiên bản ứng dụng.

Docker hóa thành công các thành phần ứng dụng

* Hiện trạng: Toàn bộ 5 thành phần của ứng dụng (frontend, user, product, order, payment) đã được đóng gói thành công thành các Docker image riêng biệt. Quá trình này được định nghĩa một cách tường minh thông qua các Dockerfile, đảm bảo rằng mỗi image chứa đựng đầy đủ mã nguồn và các thư viện phụ thuộc cần thiết.
* Ý nghĩa và Tầm quan trọng: Kết quả này xác nhận rằng mỗi thành phần đã thực sự độc lập và có thể tự mình hoạt động trong một môi trường được cô lập. Việc "docker hóa" thành công là một cột mốc quan trọng, tạo ra các đơn vị phần mềm di động và nhất quán. Các image này chính là "vật liệu xây dựng" tiêu chuẩn cho toàn bộ quy trình CI/CD về sau, loại bỏ hoàn toàn vấn đề "nó chạy trên máy tôi" và đảm bảo rằng những gì được kiểm thử cũng chính là những gì sẽ được triển khai.

Các image được lưu trữ và phiên bản hóa trên Docker Hub

* Hiện trạng: Tất cả các image sau khi được xây dựng đã được đẩy thành công lên Docker Hub. Mỗi image được gắn thẻ (tag) một cách có hệ thống, thường là theo phiên bản hoặc mã hash của commit, tạo ra một lịch sử phiên bản rõ ràng.
* Ý nghĩa và Tầm quan trọng: Docker Hub giờ đây đóng vai trò như một kho lưu trữ tập trung, trở thành "nguồn sự thật duy nhất" (single source of truth) cho các artifact có thể triển khai. Việc này tách biệt hoàn toàn giai đoạn xây dựng (CI) khỏi giai đoạn triển khai (CD). Cụm Kubernetes sẽ luôn kéo image từ nguồn đáng tin cậy này, đảm bảo tính toàn vẹn của quy trình. Hơn nữa, việc quản lý phiên bản trên Docker Hub cho phép chúng ta có thể dễ dàng triển khai một phiên bản cụ thể hoặc nhanh chóng hoàn tác (rollback) về một phiên bản ổn định trước đó nếu cần.

A screenshot of a computer

AI-generated content may be incorrect.

Hình . Các image của dự án được lưu trữ và phiên bản hóa trên Docker Hub

A screenshot of a computer

AI-generated content may be incorrect.

Hình . Các image của dự án được lưu trữ và phiên bản hóa trên Docker Hub

## Kết quả tự động hóa CI/CD

Quy trình CI/CD là kết quả trọng tâm, thể hiện việc áp dụng thành công triết lý DevOps vào thực tiễn bằng cách tự động hóa toàn bộ vòng đời phát hành phần mềm, từ mã nguồn đến môi trường sản xuất.

Vận hành thành công pipeline CI/CD end-to-end trên GitHub Actions

* Hiện trạng: Một pipeline CI/CD hoàn chỉnh, từ đầu đến cuối, đã được xây dựng và vận hành thành công bằng GitHub Actions. Khi lập trình viên đẩy mã nguồn lên nhánh main, pipeline sẽ tự động thực hiện một chuỗi các hành động được định nghĩa sẵn: kiểm thử đơn vị (unit test), xây dựng Docker image, quét bảo mật image, đẩy image lên Docker Hub, và cuối cùng là tự động triển khai phiên bản mới lên cụm K3s. Toàn bộ quá trình này diễn ra không cần bất kỳ sự can thiệp thủ công nào.
* Ý nghĩa và Tầm quan trọng: Kết quả này là minh chứng rõ ràng nhất cho việc hiện thực hóa triết lý DevOps. Nó đã biến một quy trình phức tạp, tốn thời gian và dễ sai sót thành một luồng công việc tự động, đáng tin cậy và có khả năng lặp lại. Tốc độ phát hành phần mềm được rút ngắn đáng kể, từ hàng giờ hoặc hàng ngày (nếu làm thủ công) xuống chỉ còn vài phút. Điều này cho phép đội ngũ phát triển có thể phản ứng nhanh chóng với các yêu cầu thay đổi và đưa sản phẩm đến tay người dùng một cách nhanh nhất, tạo ra lợi thế cạnh tranh và hoàn thiện vòng lặp phản hồi.

A screenshot of a computer

AI-generated content may be incorrect.

Hình . Một pipeline CI/CD đã chạy thành công trên GitHub Actions

A screenshot of a computer

AI-generated content may be incorrect.

Hình . Một pipeline CI/CD đã chạy thành công trên GitHub Actions

## Kết quả Tích hợp An ninh và Giám sát (DevSecOps)

Để quy trình DevOps trở nên hoàn thiện, các yếu tố về An ninh (Security) và Vận hành (Operations) đã được tích hợp, hình thành một quy trình DevSecOps cơ bản.

Tích hợp An ninh từ sớm (Shift-Left Security)

* Hiện trạng: Quy trình CI/CD đã được tích hợp hai công cụ bảo mật:

Trivy: Để tự động quét lỗ hổng trong Docker image ngay sau khi build.

Kube-bench: Để thực hiện đánh giá cấu hình an toàn cho toàn bộ cụm K3s.

* Ý nghĩa và Tầm quan trọng: Kết quả này đã hiện thực hóa thành công triết lý "Shift-Left", tức là dịch chuyển các hoạt động bảo mật về những giai đoạn sớm nhất. Việc dùng Trivy giống như có một "trạm kiểm soát an ninh" tự động. Thay vì đợi đến khi ứng dụng đã chạy mới phát hiện ra nó có lỗ hổng, chúng ta đã ngăn chặn được các rủi ro này ngay từ khâu "sản xuất", đảm bảo chỉ những "linh kiện" (image) an toàn mới được đưa vào hệ thống. Việc dùng Kube-bench giống như việc mời một "thanh tra xây dựng" đến để kiểm tra xem "ngôi nhà" Kubernetes của chúng ta có được xây dựng theo các tiêu chuẩn an toàn hay không. Điều này giúp "làm cứng" hạ tầng, giảm thiểu các điểm yếu có thể bị khai thác.

Xây dựng được khả năng quan sát (Observability)

* Hiện trạng: Bộ công cụ giám sát gồm Prometheus và Grafana đã được triển khai thành công lên cụm. Một dashboard trực quan đã được thiết lập để theo dõi các chỉ số tài nguyên của hệ thống theo thời gian thực.
* Ý nghĩa và Tầm quan trọng: Kết quả này đã cung cấp "đôi mắt và đôi tai" cho đội ngũ vận hành. Prometheus hoạt động như một hệ thống "cảm biến", liên tục thu thập các thông số sức khỏe của hệ thống (CPU, RAM, mạng...). Grafana là "phòng điều khiển" với các màn hình hiển thị, biến những con số phức tạp từ Prometheus thành các biểu đồ dễ hiểu. Thay vì hoạt động trong một môi trường "hộp đen", giờ đây chúng ta có thể quan sát được "sức khỏe" của toàn bộ hệ thống. Đây là nền tảng cơ bản để phát hiện sự cố sớm, phân tích hiệu năng và đưa ra các quyết định vận hành dựa trên dữ liệu, hoàn thiện vòng lặp phản hồi của DevOps.

A screenshot of a computer

AI-generated content may be incorrect.

Hình . Kết quả giám sát Cluster và các Pods

# KẾT LUẬN

## Tóm tắt kết quả đạt được

Đồ án đã nghiên cứu và ứng dụng thành công quy trình DevSecOps để xây dựng một hệ thống tự động hóa hoàn chỉnh cho việc triển khai website thương mại điện tử kiến trúc Microservices. Thông qua việc tích hợp các công nghệ và phương pháp luận hiện đại, đề tài đã hoàn thành các mục tiêu nghiên cứu đề ra:

* Tự động hóa toàn bộ vòng đời phát hành phần mềm: Đã xây dựng thành công một pipeline CI/CD trên GitHub Actions, tự động hóa các bước từ xây dựng, kiểm thử đơn vị, quét bảo mật cho đến triển khai ứng dụng, đáp ứng mục tiêu tự động hóa quy trình.
* Rút ngắn thời gian triển khai và đảm bảo tính sẵn sàng cao: Quy trình tự động giúp giảm thời gian triển khai xuống còn vài phút. Việc sử dụng cơ chế rolling update của Kubernetes đảm bảo ứng dụng được cập nhật mà không gây gián đoạn dịch vụ, đáp ứng mục tiêu giảm thiểu downtime.
* Xây dựng một mô hình DevOps tham chiếu: Đề tài đã đề xuất và hiện thực hóa một mô hình DevSecOps hoàn chỉnh, bao gồm các thành phần từ ứng dụng (Microservices), hạ tầng (K3s), tự động hóa (CI/CD), bảo mật (Trivy, Kube-bench) cho đến giám sát (Prometheus, Grafana). Mô hình này có tính thực tiễn và có thể làm cơ sở tham khảo cho các dự án tương tự.

Nhìn chung, đề tài không chỉ dừng lại ở mức độ lý thuyết mà đã xây dựng được một quy trình DevSecOps thực tế, chứng minh được tính hiệu quả và các lợi ích mà nó mang lại.

## Đánh giá chung

### Ưu điểm

* Tính tự động hóa cao: Toàn bộ quy trình từ code đến production được tự động hóa, giảm thiểu sai sót do con người, tăng tốc độ phát hành sản phẩm.
* Tích hợp an ninh từ sớm: Triết lý "Shift-Left Security" được áp dụng qua việc tích hợp Trivy, giúp phát hiện rủi ro bảo mật ngay trong giai đoạn phát triển.
* Khả năng quan sát tốt: Hệ thống giám sát cung cấp cái nhìn trực quan về hiệu năng và trạng thái của hạ tầng, giúp việc vận hành và xử lý sự cố trở nên dễ dàng hơn.
* Kiến trúc linh hoạt: Việc áp dụng Microservices và Kubernetes giúp hệ thống dễ dàng bảo trì và mở rộng từng thành phần một cách độc lập.

### Hạn chế

Bên cạnh các kết quả đạt được, đề tài cũng có một số hạn chế nằm trong phạm vi nghiên cứu đã xác định:

* Quy trình kiểm thử còn ở mức cơ bản: Pipeline đã tích hợp unit test và quét bảo mật, tuy nhiên các loại kiểm thử phức tạp hơn như kiểm thử tích hợp (integration test) và kiểm thử từ đầu đến cuối (end-to-end test) chưa được tự động hóa.
* Bảo mật mới ở mức độ phát hiện: Các công cụ Trivy và Kube-bench chủ yếu dừng lại ở việc báo cáo. Một hệ thống hoàn thiện hơn cần có cơ chế cảnh báo tự động (alerting) và các chính sách chặn triển khai nghiêm ngặt hơn.
* Giám sát chưa có chiều sâu: Hệ thống mới chỉ giám sát các chỉ số tài nguyên của hạ tầng. Khả năng truy vết yêu cầu (distributed tracing) để theo dõi một request đi qua nhiều microservice khác nhau chưa được triển khai.
* Mô hình dữ liệu và nghiệp vụ được đơn giản hóa: Việc sử dụng file CSV làm cơ sở dữ liệu giúp tập trung vào quy trình DevOps nhưng không phản ánh được các thách thức về quản lý dữ liệu trong một hệ thống thực tế.

## Hướng phát triển trong tương lai

Từ những hạn chế trên, đề tài có thể được cải thiện và mở rộng theo các hướng sau:

* Hoàn thiện quy trình kiểm thử: Tích hợp các framework kiểm thử tự động như Cypress cho end-to-end testing vào pipeline CI/CD.
* Nâng cao quy trình bảo mật (Advanced DevSecOps): Cấu hình Prometheus Alertmanager để gửi cảnh báo tự động khi có sự cố. Thiết lập các chính sách bảo mật mạng (Network Policies) trong Kubernetes để giới hạn giao tiếp giữa các service.
* Triển khai Distributed Tracing: Tích hợp các công cụ như Jaeger hoặc OpenTelemetry để có thể theo dõi và gỡ lỗi các luồng request phức tạp trong hệ thống microservices.
* Áp dụng chiến lược triển khai nâng cao (Progressive Delivery): Sử dụng các công cụ như ArgoCD hoặc Flagger để triển khai các mô hình Blue-Green/Canary, giúp giảm thiểu rủi ro khi phát hành phiên bản mới.
* Nâng cấp kiến trúc ứng dụng: Chuyển đổi từ việc sử dụng file CSV sang các hệ quản trị cơ sở dữ liệu thực thụ (như PostgreSQL, MongoDB) cho từng service để tăng hiệu năng và độ tin cậy.

TÀI LIỆU THAM KHẢO

|  |  |
| --- | --- |
| [1] | M. Fowler, "Monolith First," Martin Fowler, 2015. [Online]. Available: https://martinfowler.com/bliki/MonolithFirst.html. |
| [2] | J. L. a. M. Fowler, "Microservices," Martin Fowler, 2014. [Online]. Available: https://martinfowler.com/articles/microservices.html. |
| [3] | R. Hat, "What is CI/CD?," Red Hat, 2022. [Online]. Available: https://www.redhat.com/en/topics/devops/what-is-ci-cd. |
| [4] | Docker, "Get Started with Docker," Docker, Docker. [Online]. Available: https://docs.docker.com/get-started/. |
| [5] | Kubernetes, "Kubernetes Documentation," CNCF, 2024. [Online]. Available: https://kubernetes.io/docs/home/. |
| [6] | R. Labs, "K3s - Lightweight Kubernetes," Rancher Labs, 2024. [Online]. Available: https://k3s.io/. |
| [7] | M. Project, "MetalLB," MetalLB, 2023. [Online]. Available: https://metallb.universe.tf/. |
| [8] | GitHub, "GitHub Actions Documentation," GitHub, 2024. [Online]. Available: https://docs.github.com/en/actions. |
| [9] | A. Security, "Trivy Documentation," Aqua Security, 2023. [Online]. Available: https://trivy.dev/latest/docs/. |
| [10] | P. Authors, "Prometheus Documentation," CNCF, 2023. [Online]. Available: https://prometheus.io/docs/introduction/overview/. |
| [11] | G. Labs, "Grafana Documentation," Grafana Labs, 2024. [Online]. Available: https://grafana.com/docs/grafana/latest/. |
| [12] | Microsoft, "DevOps Resource Center," Microsoft, 2023. [Online]. Available: https://learn.microsoft.com/en-us/devops/. |