Bachelor’s thesis

Information and Communications Technology

2024

Dieu Thu Vu

CI/CD Automation's Impact on Microservices Project Management

Bachelor’s Abstract

Turku University of Applied Sciences

Degree Programme in Information and Communications Technology

2024 | 31 pages

Dieu Thu Vu

CI/CD Automation's Impact on Microservices Project Management

This thesis topic will focus on the impact of Continuous Intergration/ Continuous Deployment automation on the management of microservices architectures. Due to the implementation of microservices, the traditional approaches cannot be used singularly since each of the projects has become complex. As a result, this paper seeks to examine how the CI/CD pipeline automation can improve the basic aspects of project management including resource management, monitoring, risk management, and teamwork. In particular, it specifies how it augments visibility, resource utilization, risk mitigation with early detection, or reduces time-to-market by efficient deployment. The findings are useful to business stakeholders including project managers, DevOps professionals, software developers, IT leaders, and educational institutions in integrating CI/CD pipelines to improve project outcomes. Combining literature review, hands-on work, case study evaluation and theoretical framework, this thesis outlines the best practices to improve CI/CD automation in microservices project management. The outcomes advance the knowledge base about how the automation of the development process may improve productivity and cooperation in microservices initiatives.

Keywords:

CI/CD, DevOps, Project Management, Microservices, Pipelines, Continuous Intergration, Continuous Deployment.

Content

List of abbreviations **5**

1 Introduction **6**

2 Literature Review **8**

2.1 Monolithic Architecture 8

2.2 Microservice Architechture 9

2.3 CI/CD 11

2.4 CI/CD Impact on Project Management in Microservices 12

2.5 DevOps 12

2.6 Project Management in Microservices 13

3 Hands-On CI/CD Pipeline Implementation for Microservices **15**

3.1 Project Overview and Architecture 15

3.2 CI/CD Pipeline Setup with GitHub Actions 16

3.1.1 Build Automation 17

3.1.2 Testing Automation 19

3.1.3 Deployment Automation 20

3.3 Observation 21

4 Case Studies of CI/CD Automation in Microservices Project Management **23**

4.1 Netflix: Driving Scalability with CI/CD 23

4.2 Spotify: Enhancing Developer Productivity with CI/CD 24

4.3 Insights from Netflix and Spotify’s CI/CD Practices 24

5 Best Practices for Integrating CI/CD in Project Management **26**

6 Conclusion **28**

References **30**

Pictures

Picture 1. Monolithic architecture (Hidayat, 2020). 9
Picture 2. Microservice architecture (Hidayat, 2020). 10
Picture 3. CI/CD (Black Duck, n.d.). 11
Picture 4. DevOps tools (Kulyk, 2019). 13
Picture 5. Architecture of the product-service microservice 15
Picture 6. Architecture of the customer-service microservice. 16
Picture 7. Application is pushed on Github. 17
Picture 8. .yml file configuration for automatically building Docker images on repository changes. 17
Picture 9. Dockerfile of product-service. 18
Picture 10. .yml file configuration for tagging and storing built images in Docker Hub. 18
Picture 11. .yml file - Installing dependencies and running tests. 19
Picture 12. Adding a testing tool in package.json. 19
Picture 13. Docker Compose file. 20
Picture 14. CI/CD Pipeline is built successfully. 20
Picture 15. Images of microservices are pushed to Dockerhub. 20
Picture 16. index.test.js file - Starting and stopping the server properly during tests to avoid resource conflicts. 21
Picture 17. Modifying the pipeline to check if code changes affect the product-service or customer-service directories. 22

List of abbreviations

API	Application Programming Interface
CI/CD	Continuous Integration and Continuous Delivery/Deployment
DevOps	Development Operations
Golang	‘Go’ language
HTTP	Hypertext Transfer Protocol
IT	Information Technology
JSON	JavaScript Object Notation
YAML	yet another markup language

1 Introduction

In recent years, microservices architecture and Continuous Integration and Continuous Delivery/Deployment (CI/CD) pipelines of software development industry have been evolving. Instead of building monolithic applications as before, with a microservices architecture, an application is built as independent components that run each application process as a service. This structure enables development teams to scale applications in a much simpler and flexible manner in a fast-growing and dynamic technology market. However, the complexity of managing microservices comes with project management challenges, such as coordinating multiple services, maintaining consistency across deployments, and effectively tracking the lifecycle of each service.

CI/CD pipelines have also become an important part of modern software engineering as they automate the processes of building, testing, and deploying applications. CI/CD enables faster and more reliable delivery of new features and updates by automating these phases. This thesis will delve into the impacts of CI/CD automation on project management practices, specifically in the microservices architecture, by analyzing its role in streamlining workflows, enhancing collaboration, and addressing challenges in deployment and service management. The objectives of this thesis are to explore how CI/CD is used in managing microservices, to develop a CI/CD pipeline for a microservices application, and to evaluate the process, ensuring a comprehensive analysis of its impacts.

The reason for choosing this topic is the current trend in the industry, which focuses on microservices and Development-Operation (DevOps) to increase the speed and reliability of software delivery. While organizations integrate these practices into their projects, the understanding of how CI/CD impacts project management can guide more effective decisions.

This study includes a review of literatures, case studies, and prototypes using GitHub Actions to offer practical solutions for project managers working with microservice architecture. Chapter 1 introduces the research topic, its

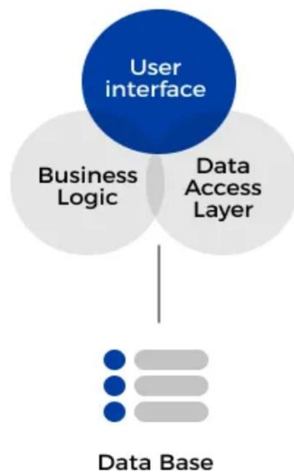
significance, and the objectives of the study. Chapter 2 provides a comprehensive literature review, discussing key concepts such as microservices architecture, CI/CD, and their intersection with project management. Chapter 3 describes the implementation of a CI/CD pipeline for a microservices application and the tools used. Chapter 4 presents case studies of industry leaders, such as Netflix and Spotify, analyzing their CI/CD practices and how they relate to the findings of this study. Chapter 5 discusses best practices for integrating CI/CD into project management, focusing on strategies for workflow design, risk mitigation, and team collaboration. Finally, Chapter 6 concludes the thesis, summarizing the key findings, discussing their implications for project management in microservices, and proposing recommendations for future research.

2 Literature Review

This chapter provides an overview of the key concepts and practices that are central to understanding the study. It starts by looking at monolithic architecture, a traditional approach to software development, and contrasts it with the more modern microservices architecture, which has revolutionized how applications are built and managed. The discussion then moves to Continuous Integration and Continuous Delivery (CI/CD), focusing on its role in streamlining development processes and its impact on project management in microservices. The chapter also explores the concept of DevOps, which complements CI/CD and microservices by fostering collaboration and automation. Finally, the chapter examines project management in microservices, addressing the unique challenges and strategies needed to manage decentralized and complex systems effectively.

2.1 Monolithic Architecture

In the past, systems were developed using monolithic architecture, where enterprise software applications were designed to address a wide range of business requirements within a single unified structure. Therefore, the software provided hundreds of features and all these features were bundled in a monolithic application (Amazon Web Services, n.d.a) (Picture 1).



Picture 1. Monolithic architecture (Hidayat, 2020).

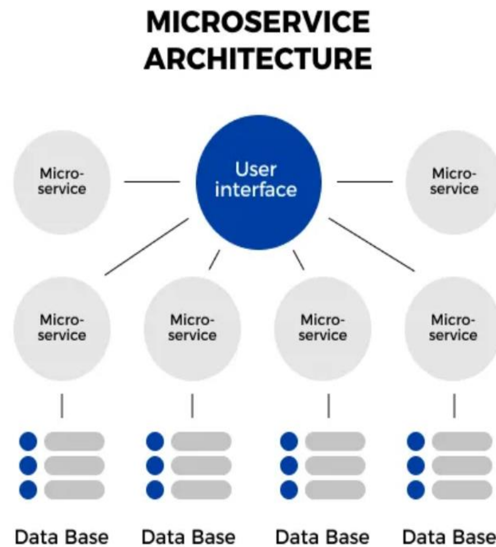
Monolithic applications are typically huge – usually exceeding 100,000 lines of code, and in some cases, they can contain even over a million lines of code. When developing software using a monolithic architecture, all modules are grouped together into one large project. During the deploying phase, the code is uploaded to the server and configured to run. This architecture works well because it is simple and easy to code (Hidayat, 2020).

However, as the software grows and complexity, its disadvantages are revealed. Upgrading a module requires redeploying the entire codebase, which means that users will temporarily lose access to the full functionality of the system during deployment. Additionally, when scaling to serve many users, the server itself must be upgraded (Amazon Web Services, n.d.a).

2.2 Microservice Architechture

Unlike monolithic architecture, with microservice architecture, modules are separated into micro-serivices. Each microservice runs on a different server or

instance and communicates with other microservices over a network, typically using HTTP protocols or message queues (Amazon Web Services, n.d.a) (Picture 2).



Picture 2. Microservice architecture (Hidayat, 2020).

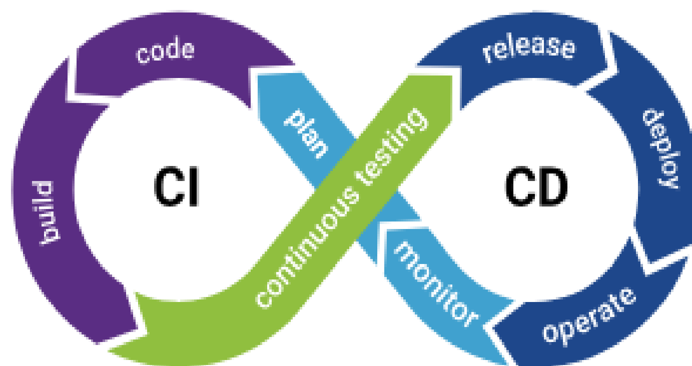
Nowadays, applications are often very large and constantly updated, typically social networking platforms such as Youtube, Tiktok, etc. With a monolithic architecture, upgrading the application will be very difficult and time-consuming. To solve that problem, large applications need to be separated into small services. Each service manages a separate database, located on a separate server (Dragoni et al., 2017).

Because the services are separate, they can be developed in different programming languages (one service is developed with Python and the other is developed with Golang), deployed and run on separate containers. Moreover, the development team can apply automation processes, such as building, deploying, monitoring (Amazon Web Services, n.d.a; Dragoni et al., 2017).

2.3 CI/CD

The concept of CI/CD often refers to automation in the software development and product delivery process, helping integration take place faster and the finished product is delivered to users in the shortest time (Red Hat, 2023).

Currently, CI/CD has been widely applied to the workflow of businesses working in the IT field, along with DevOps and Agile. A complete CI/CD process can be understood as follows: The developer commits code (pushes code to the server), after which the CI/CD process automatically runs the build, runs the tests, deploys the product, and finally continues to deliver the product to users (Black Duck, n.d.) (Picture 3).



Picture 3. CI/CD (Black Duck, n.d.).

CI/CD helps reduce errors by automatically testing code after each commit. All stakeholders, from developers to testers, can see the progress of builds, allowing for better coordination to ensure the highest quality software (Red Hat, 2023). In addition, CI/CD helps maintain consistency in software releases by ensuring that all builds follow a pre-established standard testing process. This helps ensure consistent software quality (Black Duck, n.d.). In large software systems, scaling and maintaining the system can become very complex. CI/CD

plays an important role in offloading and automating these processes, ensuring the stability and scalability of the system (Red Hat, 2023).

2.4 CI/CD Impact on Project Management in Microservices

By automating code integration, testing, and deployment, CI/CD pipelines reduce manual work. From that, development teams will be able to deliver updates quickly and frequently (Codefresh, 2024). From a project management perspective, this automation enables faster development cycles, improved visibility into the state of each service, and the ability to identify and resolve issues earlier in the development process (Microsoft Learn, n.d.). CI/CD pipelines also help developers better manage risk by standardizing human error implementations and ensuring that each service is tested and deployed independently (Vehvaria, n.d.; Mosyan, 2023).

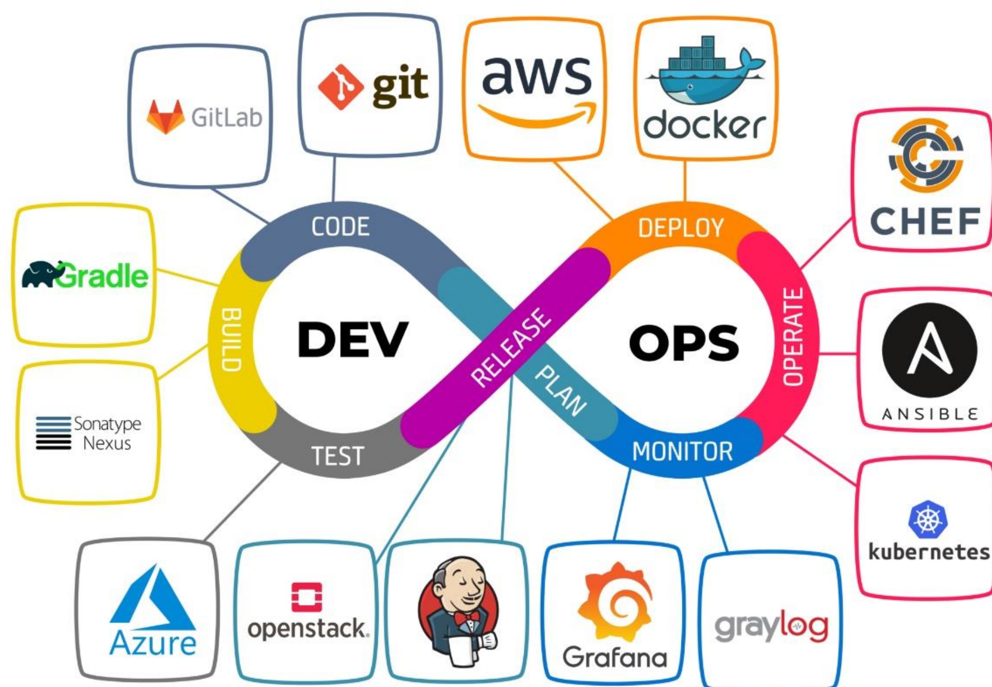
However, applying CI/CD to microservices also brings challenges, including the need for specialized tooling, configuration management, and the skills to effectively maintain automated pipelines. Understanding the balance between automation and manual monitoring is essential for project managers when adopting CI/CD (Mosyan, 2023; Vehvaria, n.d.).

2.5 DevOps

DevOps is a combination of principles, practices, processes and tools that help automate the software development and delivery process. With DevOps, companies can release small features very quickly and incorporate the feedback they receive quickly. These two stages are relatively separate, especially in companies of medium size and above (Amazon Web Services, n.d.b). Therefore, the concept of DevOps was born to optimize the software

development cycle, helping software products to be released faster and more frequently (SHALB, 2024).

DevOps is an important component in the software development process along with the Agile method. It helps to complete the transformation of the software development and operation process from the waterfall model to the continuous development/release model (CI/CD). All serve the ultimate purpose of improving the ability to deploy software quickly. Thereby, increasing the competitiveness of products/businesses (Kulyk, 2019; SHALB, 2024). Many tools have been developed to serve the application of DevOps (Picture 4).



Picture 4. DevOps tools (Kulyk, 2019).

2.6 Project Management in Microservices

Project management practices in software development have changed significantly as many applications have moved from monolithic architectures to

microservices architectures. Microservices require decentralized management due to the independence of each service, which can lead to complex dependencies and the need for specialized project management strategies (Dragoni et al., 2017). Traditional project management approaches may not align well with microservice architecture, where independent teams often need to work closely together to maintain cohesion across the entire application (Codefresh, 2024).

Effective project management in microservices requires more communication, coordination between service teams, and tracking of the lifecycle of each service. To scale and deploy microservices better, project managers must have a well-planned strategy, and the development team must possess the right DevOps skillset (OpsTree Solutions, 2021; Dragoni et al., 2017).

3 Hands-On CI/CD Pipeline Implementation for Microservices

This part details the practical setup and implementation of a CI/CD pipeline for a simple microservices application, which is designed to examine the project management impact of CI/CD in real-world scenarios. With GitHub Actions, this hands-on project sets up a CI/CD pipeline to automate building, testing, and deploying two services and assesses the effectiveness and challenges that development team might face in each stage.

3.1 Project Overview and Architecture

This section provides an overview of the architecture and purpose of the microservices used in the study, highlighting their role in the CI/CD pipeline implementation.

Product Service: Responsible for the product information data and has endpoints for creating and listing a product (Picture 5).

Picture 5. Architecture of the product-service microservice

Customer Service: Handles customer information with similar functionalities for the customer record creation and retrieval (Picture 6).

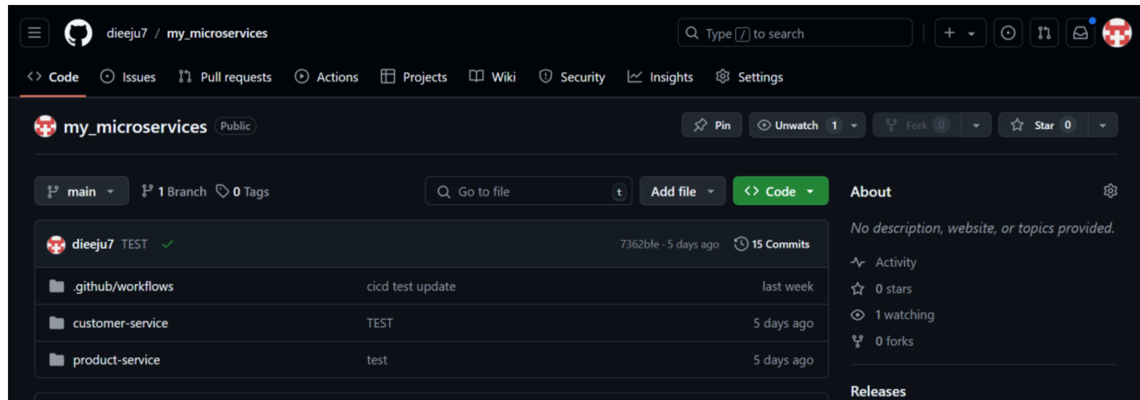
```
const express = require('express');
const app = express();
const port = process.env.PORT || 3002;

let customers = [{ id: 1, name: "Customer A", email: "customerA@example.com"

app.use(express.json()); // To parse JSON bodies

// Get list of customers
app.get('/customers', (req, res) => {
  res.json(customers);
});
```

Picture 6. Architecture of the customer-service microservice.

Each service is developed as an independent Node.js application and containerized with Docker for independent deploying. This microservices will highlight the need for reliable CI/CD processes to streamline integration and deployment across multiple, distributed services.

3.2 CI/CD Pipeline Setup with GitHub Actions

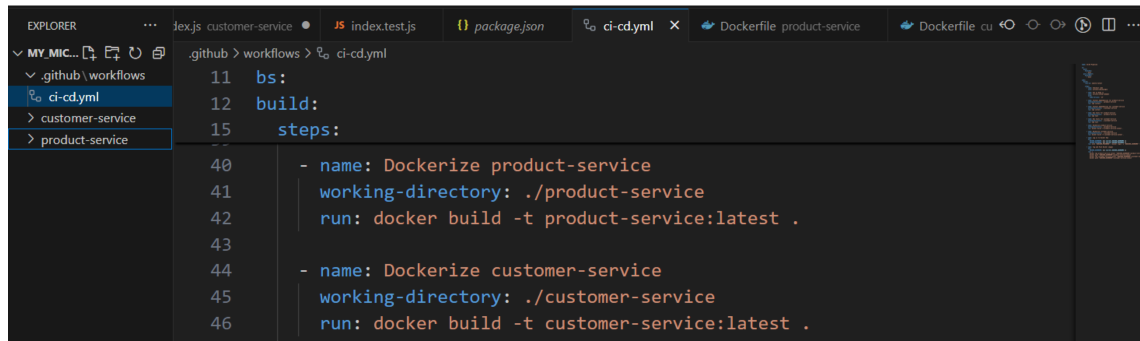
GitHub Actions was chosen for the CI/CD pipeline due to its integration with GitHub repositories, flexibility, and extensive support for automation. The pipeline is designed to trigger on every commit or pull request, automating the build, test, and deployment phases to simulate a production-like environment. Configuration files (.yml) are stored in each repository to define workflow actions for both services (Picture 7).



Picture 7. Application is pushed to Github.

3.1.1 Build Automation

The first stage in the CI/CD pipeline setup is build automation. The pipeline automatically builds Docker images for each service whenever changes are pushed to the repository (Picture 8).



Picture 8. .yml file configuration for automatically building Docker images on repository changes.

Dockerfiles for each service define the base image, dependencies, and configurations required to containerize the application (Picture 9).

```
# Use an official Node.js runtime as a parent image
FROM node:14

# Set the working directory in the container
WORKDIR /app

# Copy the current directory contents into the container at /app
COPY . .

# Install dependencies
RUN npm install

# Make port 3001 available to the world outside this container
EXPOSE 3001

# Run the app
CMD ["node", "index.js"]
```

Picture 9. Dockerfile of product-service.

The built images are tagged and stored in Docker Hub, ensuring that the newest version is available for deployment. This stage ensures the services can be deployed across various environments (Picture 10).

```
jobs:
  build:
    steps:
      - name: Log in to Docker Hub
        env:
          DOCKER_USERNAME: ${{ secrets.DOCKER_USERNAME }}
          DOCKER_PASSWORD: ${{ secrets.DOCKER_PASSWORD }}
        run: echo "$DOCKER_PASSWORD" | docker login -u "$DOCKER_USERNAME" --password-stdin

      - name: Tag and Push Docker images
        env:
          DOCKER_USERNAME: ${{ secrets.DOCKER_USERNAME }}
        run: |
          docker tag product-service:latest "$DOCKER_USERNAME"/product-service:latest
          docker push "$DOCKER_USERNAME"/product-service:latest
          docker tag customer-service:latest "$DOCKER_USERNAME"/customer-service:latest
          docker push "$DOCKER_USERNAME"/customer-service:latest
```

Picture 10. .yml file configuration for tagging and storing built images in Docker Hub.

3.1.2 Testing Automation

Testing automation is the second focus of the pipeline. Unit tests, implemented using Jest and Supertest, validate the core functionality of each service. For example, the tests check whether the GET and POST endpoints of both services return the correct responses and status codes. The pipeline is configured to run these tests automatically with each push to the repository. This ensures that errors are caught early in the development cycle, providing immediate feedback to developers. The test results are logged in the pipeline workflow, offering visibility into any issues that arise during the integration phase (Picture 11; Picture 12).

```yaml
11  jobs:
12    build:
15      steps:
24        - name: Install dependencies for product-service
25          working-directory: ./product-service
26          run: npm install
27
28        - name: Install dependencies for customer-service
29          working-directory: ./customer-service
30          run: npm install
31
32        - name: Run tests for product-service
33          working-directory: ./product-service
34          run: npm test
35
36        - name: Run tests for customer-service
37          working-directory: ./customer-service
38          run: npm test
```

Picture 11. .yml file - Installing dependencies and running tests.

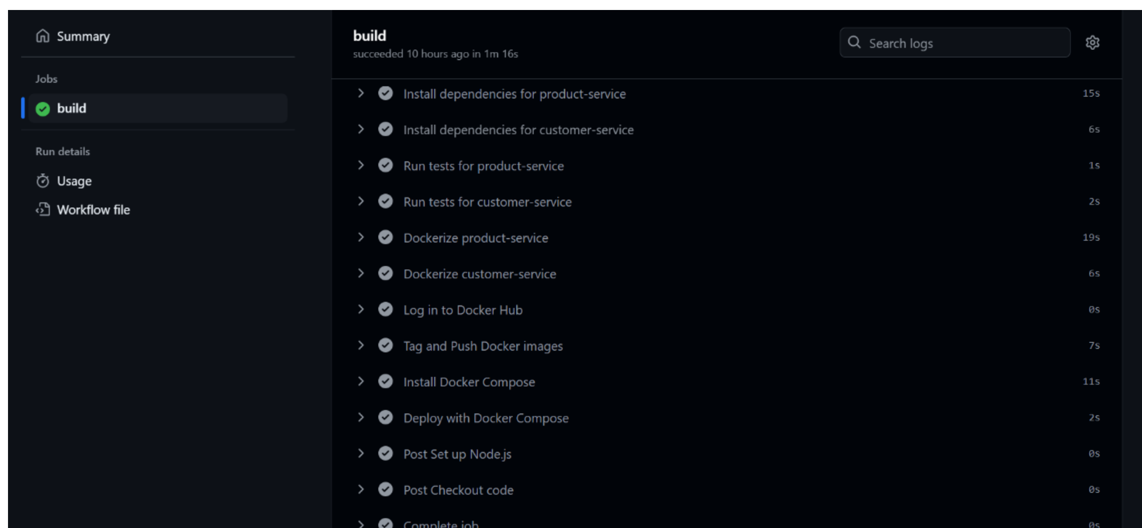
```json
"scripts": {
  "test": "jest --detectOpenHandles",
  "test:watch": "jest --watch"
},
```

Picture 12. Adding a testing tool in package.json.

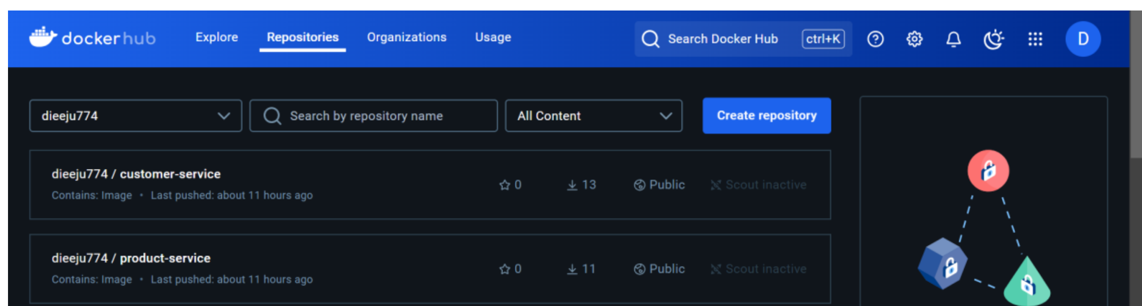
3.1.3 Deployment Automation

Docker images are pushed to Docker Hub and deployed locally using Docker Compose. Docker Compose configuration defines each service, its ports, and links, simplifying multi-container deployment (Picture 13; Picture 14; Picture 15).

Picture 13. Docker Compose file.



Picture 14. CI/CD Pipeline is built successfully.

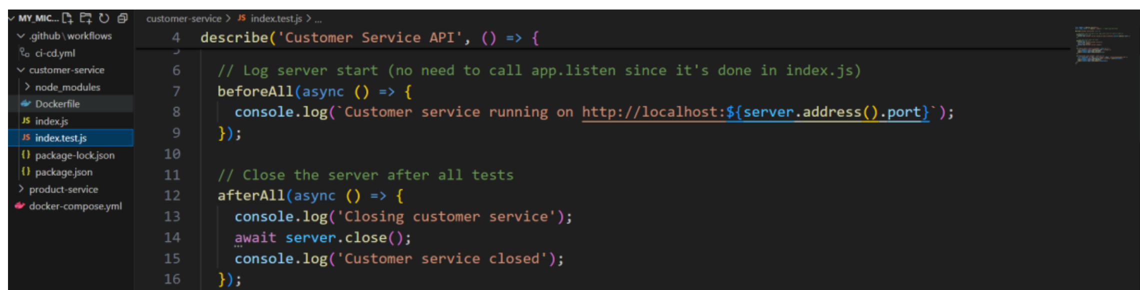


Picture 15. Images of microservices are pushed to Dockerhub.

3.3 Observation

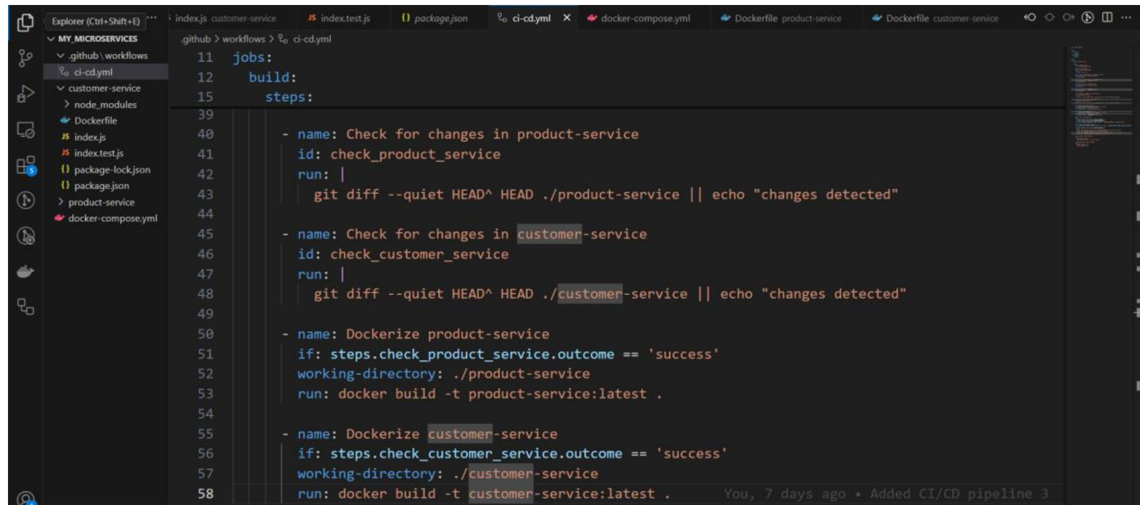
During the implementation, several challenges appeared. The first problem was that services shared the same port on which they started, conflicting with each other. This was solved by giving each service its own port; the product-service having a port number of 3001 and the customer-service having a port number of 3002 This adjustment ensured that both services could run simultaneously without interfering with each other.

Another challenge involved ensuring that the tests ran independently. During testing, conflicts arose due to improper handling of server connections, leading to resource contention. This issue was addressed by carefully managing the server lifecycle during tests, specifically starting and stopping the server properly to avoid conflicts (Picture 16).



Picture 16. index.test.js file - Starting and stopping the server properly during tests to avoid resource conflicts.

Additionally, Docker Hub rate limits occasionally posed a problem due to frequent pushes of updated images during the pipeline's execution. This issue was mitigated by modifying the workflow to push images only when relevant code changes were detected in the product-service or customer-service directories. This optimization reduced the frequency of unnecessary image pushes, ensuring compliance with Docker Hub's rate limits (Picture 17).



Picture 17. Modifying the pipeline to check if code changes affect the product-service or customer-service directories.

There are several advantages that the CI/CD pipeline demonstrates in the context of managing microservices applications. It also reduced work fraught at build and test, improved its quality, and testing speed as well as brought drastic changes in the deployment phases. Automated testing reduced the risk while integrating systems and the Docker images made deployment consistent across environments.

The pipeline has described the basic life cycle phases of build-test-deployment and given the vision of the possibilities and limitations involved in the CI/CD automation of microservices projects. This experiment provides a starting point of understanding practicalities of CI/CD project management specifically in a microservices setting and as such presents the basis of future work to be done in terms of CI/CD scalability to more complex architectures.

4 Case Studies of CI/CD Automation in Microservices Project Management

4.1 Netflix: Driving Scalability with CI/CD

One of the streaming giants Netflix has also achieved a reliable CI/CD pipeline that helps contain over a thousand microservices scattered across different regions. Integral to this approach is Spinnaker, an open-source, multi-cloud continuous delivery tool that enables the submission and distribution of updates and enhancements (Seyhunak, 2020). It is important to understand that one of the key and well-established strategies of Netflix is the use of automated and gradual implementations. By employing Spinnaker, Netflix deploys changes discretely for the region to minimize the impact and provide a more stable base in the actual live environment.

Netflix uses high level of deployment strategies like blue/green and canary deployments. These enable the company to try changing its features or updates for a few of its users before doing it to all of its users. In a similar way, this approach reduces periods when some clients may not be attended to and also increases early detection of any possible problems. Another principle also fielding for CI/CD pipeline includes several forms of testing whereby Netflix emulate unit tests, integration tests, and regression tests. Furthermore, the adaptation of principles covering activities related to chaos engineering aids in keeping the system robust through application of stress tests (Seyhunak, 2020).

While deployments bring problems, Netflix discourages having complicated features and protocols and focuses mainly on fast rollback procedures to be able to accomplish timely service delivery. From such practices, Netflix has been able to deploy features quickly, expand internationally and maintain excellent system availability. Thus, it allows the platform to provide as often as possible updates throughout its services and at the same time keeping high availability of the portal that is expected by millions of users (Seyhunak, 2020).

4.2 Spotify: Enhancing Developer Productivity with CI/CD

Spotify, a global music streaming platform, has implemented a CI/CD pipeline tailored to its unique squad-based development model, which promotes team autonomy and ownership. This structure enables the teams to independently develop, test, and deploy the microservices, meaning little interdependency, and thus faster releases (Spotify Engineering, 2020).

Backstage is an open source developer portal which has been built in-house and is a main enabler for Spotify's CI/CD. Through Backstage, CI/CD pipelines can be hosted and viewed as unique items that can be easily navigated, enhancing collaboration between the corporate Spotify teams and the enterprise's squads. This tool makes deployment easier and directly puts it into the hands of developers to develop new features (Spotify Engineering, 2020).

An area is with regards to testing known as shift left testing, Spotify has integrated it into CI/CD pipeline to ensure testing is done early in the development process. Moreover, feature toggles help deliver new features gradually to particular sets of users in the case with Spotify. This allows for live response and also affects only a select few users rather than almost all of the populace (Spotify Engineering, 2020).

Given the complexity of this pipeline, Spotify has been able to make several improvements in increasing developer efficiency, reliability, and velocity in the development-releasement process. This agility is important given the continuously evolving business environment in the music streaming business (Spotify Engineering, 2020).

4.3 Insights from Netflix and Spotify's CI/CD Practices

The case studies in the Netflix's CI/CD pipeline and Spotify's CI/CD pipeline also show that it is crucial to choose the right pipeline strategy according to the company context.

Both organizations rely on incremental deployments, comprehensive testing, and automation to streamline workflows and reduce errors. Their use of proprietary tools - Spinnaker at Netflix and Backstage at Spotify - further reflects the importance of customizing CI/CD pipelines to meet specific requirements.

These practices provide valuable lessons for organizations implementing CI/CD pipelines and reinforce the relevance of CI/CD to the present study's exploration of its impact on project management practices in microservices architectures (Seyhunak, 2020; Spotify Engineering, 2020).

5 Best Practices for Integrating CI/CD in Project Management

Based on the hands-on experience and insights from case studies, several best practices emerge for integrating CI/CD into project management, particularly for microservices-based projects:

First, adopting a modular workflow is critical. Breaking an application into smaller, independent services allows teams to work on multiple components in parallel, which not only speeds up development but also makes debugging and maintenance more manageable.

Second, automation is key to mitigating risk. Incorporating automated tests - from unit tests to regression tests - ensures that problems are caught early in the process. Incremental deployment strategies, such as blue/green or canary deployments, can help minimize disruption to live environments. Adopting robust recovery mechanisms increases the system's ability to recover quickly from failed updates.

Third, fostering collaboration within the team is essential. Version control systems integrated with the CI/CD pipeline promote transparency and allow team members to track progress in real time. Regular commits and updates help detect issues earlier, while appropriate training ensures that all team members are comfortable navigating the tools and workflows.

Finally, continuous monitoring and optimization are essential. Regularly reviewing pipeline performance helps identify and address bottlenecks, such as long build times or configuration issues. Using monitoring tools to track system health and deployment metrics ensures the pipeline remains efficient and reliable over time.

By combining these activities, project managers can unlock the full potential of the CI/CD pipeline. This approach not only improves the speed and quality of software delivery, but also creates a more cohesive and efficient development

environment, ultimately aligning engineering workflows with broader project goals.

6 Conclusion

The implementation and analysis of a CI/CD pipeline for a microservices-based application have highlighted its profound impact on modern software development and project management. Fully automating features such as building, testing, and deployment proved to enhance development speed, minimizing the chances of manual mistakes, and strengthening system stability. These findings closely align with insights from literature and the case studies of Netflix and Spotify, demonstrating how CI/CD practices can be successfully applied across different contexts and scales.

The hands-on work illustrated the benefits of breaking down applications into smaller, modular services. This approach, combined with an automated pipeline, enabled parallel development and rapid iteration, which in turn boosted productivity and minimized risks. Continuous automated testing helped identify major problems that were being developed, creating only high-quality code. Challenges such as Docker Hub rate limits and deployment conflicts, highlighted the importance of optimization and adaptability, which are key attributes of a successful CI/CD strategy.

The examples of Netflix and Spotify further explained the significance of CI/CD practices. These practices emphasize the role of CI/CD in creating scalable, reliable, and agile development environments while promoting collaboration and accountability across teams.

For project managers, this work provides practical recommendations on the implementation of CI/CD in microservices projects. Other recommendations include the compartmentalization of the processes, use of automation tools to avoid major risks, and use of collaboration tools to enhance team coordination and pipeline monitoring. By adopting these strategies, project managers can achieve faster delivery cycles, improve software quality, and align development activities more effectively with broader project goals.

In conclusion, CI/CD pipelines are far more than just technical tools; they are strategic frameworks that bridge the gap between development and project

management. As organizations increasingly adopt microservices architectures and agile methodologies, CI/CD has become an essential enabler of scalability, reliability, and efficiency. This study serves as a foundation for exploring how CI/CD can be further optimized and its implications for both technical workflows and project management practices in the ever-evolving field of software engineering.

References

Amazon Web Services (n.d.a) What are microservices?. Available at: https://aws.amazon.com/microservices/ (Accessed: 17 November 2024).

Amazon Web Services (n.d.b) What is DevOps?. Available at: https://aws.amazon.com/devops/what-is-devops/ (Accessed: 17 November 2024).

Black Duck (n.d.) What is CI/CD and how does it work?. Available at: https://www.blackduck.com/glossary/what-is-cicd.html (Accessed: 17 November 2024).

Codefresh (2024) CI/CD and Agile: Why CI/CD promotes true agile development. Available at: https://codefresh.io/learn/ci-cd-pipelines/ci-cd-and-agile-why-ci-cd-promotes-true-agile-development/ (Accessed: 17 November 2024).

Dragoni, N., Giallorenzo, S., Lafuente, A., Mazzara, M., Montesi, F., Mustafin, R. & Safina, L. (2017) 'Microservices: Yesterday, Today, and Tomorrow', Present and Ulterior Software Engineering, pp. 195–216. Available at: https://doi.org/10.1007/978-3-319-67425-4_12 (Accessed: 17 November 2024).

Fireship (2020) DevOps CI/CD Explained in 100 Seconds. Available at: https://www.youtube.com/watch?v=scEDHsr3APg (Accessed: 17 November 2024).

Hidayat, T. (2020) 'Microservice versus monolithic architecture. What are they?', Medium. Available at: https://medium.com/javanlabs/micro-services-versus-monolithic-architecture-what-are-they-e17ddc8d3910 (Accessed: 17 November 2024).

Kulyk, A. (2019) 'What is DevOps and where is it applied?', SHALB. Available at: https://shalb.com/blog/what-is-devops-and-where-is-it-applied (Accessed: 17 November 2024).

Microsoft Learn (n.d.) CI/CD for microservices. Available at: https://learn.microsoft.com/en-us/azure/architecture/microservices/ci-cd (Accessed: 17 November 2024).

Mosyan, D. (2023) ‘CI/CD for microservices’, Medium. Available at: https://medium.com/@dmosyan/ci-cd-for-microservices-1b5582f3e1fd (Accessed: 17 November 2024).

Red Hat (2023) What is CI/CD?. Available at: https://www.redhat.com/en/topics/devops/what-is-ci-cd#:~:text=Hat%20can%20help-,Overview,accelerate%20the%20software%20development%20lifecycle (Accessed: 17 November 2024).

Seyhun, A. (2020) ‘Learning best practices from Netflix tech stack CI/CD pipeline’, Medium. Available at: https://medium.com/@seyhunak/learning-best-practices-from-netflix-tech-stack-ci-cd-pipeline-a25a58f46711 (Accessed: 17 November 2024).

Spotify Engineering (2020) ‘How we improved developer productivity for our DevOps teams’, Spotify Engineering. Available at: https://engineering.atspotify.com/2020/08/how-we-improved-developer-productivity-for-our-devops-teams/ (Accessed: 17 November 2024).

Varma, R. (2021) ‘Major DevOps challenges faced while implementing microservices’, OpsTree Solutions. Available at: https://opstree.com/blog/2021/06/02/major-devops-challenges-faced-while-implementing-microservices/ (Accessed: 17 November 2024).

Vehvaria, N. (n.d.) ‘Integrating Agile with Microservices and CI/CD’, LinkedIn. Available at: https://www.linkedin.com/pulse/integrating-agile-microservices-cicd-naeem-vehvaria-mj6nf/ (Accessed: 17 November 2024).