Accredited SINTA 2 Ranking
Decree of the Director General of Higher Education, Research, and Technology, No. 158/E/KPT/2021
Validity period from Volume 5 Number 2 of 2021 to Volume 10 Number 1 of 2026

Published online at: **http://jurnal.iaii.or.id**

JURNAL RESTI
(Rekayasa Sistem dan Teknologi Informasi)
Vol. 8 No. 3 (2024) 368 - 376 e-ISSN: 2580-0760

Implementing Continuous Integration and Deployment Strategy: Cloversy.id RESTful API Development

Eric Prima Wijaya[1*], Sandy Kosasi[2], David[3]
[1,3]Teknik Informatika, STMIK Pontianak
[2]Sistem Informasi, STMIK Pontianak
[1]ericrimaw@stmikpontianak.ac.id, [2]sandykosasi@stmikpontianak.ac.id, [3]david@stmikpontianak.ac.id

Abstract

The software development cycle involves testing and deployment stages that can be laborious and time-consuming, especially in collaborative projects that involve several developers. Implementing Continuous Integration (CI) and Continuous Delivery (CD) offers a solution to streamline this process. This study presents a case study of the Cloversy.id RESTful API project, highlighting challenges encountered during development and the implementation of a new system using GitHub Actions as the DevOps tool. The research resulted in the adoption of a new system, replacing the conventional practices previously employed by the Cloversy.id development team. Employing flowcharts, the study meticulously mapped out the development flow, pinpointing bottlenecks and areas for optimization within the cycle. Notably, the implementation of a CI/CD pipeline resulted in a notable improvement, with a 35% increase in speed for CI and a remarkable 39% enhancement for CD. GitHub Actions played a pivotal role in automating critical tasks, reducing reliance on manual intervention, and minimizing dependency on team leaders. The platform's features, including detailed logs and email notifications, empowered team leaders and developers alike to take informed actions swiftly. Furthermore, the study highlights the novelty of integrating CI/CD by considering factors such as branching strategy, code review practices, testing methodologies, deployment methods, and infrastructure.

Keywords: *automation; DevOps; continuous integration; continuous delivery*

How to Cite: Eric Prima Wijaya, Sandy Kosasi, and David, "Implementing Continuous Integration and Deployment Strategy: Cloversy.id RESTful API Development", *J. RESTI (Rekayasa Sist. Teknol. Inf.)*, vol. 8, no. 3, pp. 368 - 376, Jun. 2024.
DOI: https://doi.org/10.29207/resti.v8i3.5527

1. Introduction

Software development in the entire cycle can be divided into five primary stages to produce high-quality and low-cost software: planning, creating, developing, testing, and deploying [1]. Testing and deployment are the last 2 stages that have an essential role right before the application goes to market. Software testing is a critical aspect of development to ensure software modules work as intended, which helps minimize the chance of failure reaching production or client level [2]. Meanwhile, the deployment process continues the testing stage by launching the application to the staging environment to satisfy User Acceptance Testing (UAT) before moving on to production deployment [3].

Testing and deployment stages could be laborious, manual, and time-consuming if several developers are working on the same project; challenges often occur when merging multiple development branches to the main branch and the process of deployment stage preparation [4]. One practical solution to overcome those challenges is to implement DevOps with continuous integration (CI) and continuous delivery (CD) flow [5]. CI is a development practice that requires application developers to integrate, such as merging their code changes, into a collaborative repository version control system such as GitHub. Meanwhile, CD is the incremental delivery of software in a production environment, or it could be said that CD is the automation of changes to software so that it can be immediately deployed to a system that can be reached by target users [6]. Simultaneously CI/CD means the process of automating recurring tasks related to deployment and testing in the application development process, either on cloud or on-premise systems environment [7].

DevOps can significantly improve time to deployment and provide a much smoother experience for developers to maintain code integrity at all stages of development,

Received: 03-11-2024 | Accepted: 24-04-2024 | Published Online: 05-06-2024

which results in much more efficient outcomes [8]. Implementation of CI/CD is also proven to significantly improve repository commit velocity by 141.19% [9]. Commit velocity directly affects engineering productivity level to perform smoother and more efficient development cycle.

There is a wide range of CI/CD reliable tools to be used by developers to design and perform CI/CD flow, such as Github Actions, Jenkins, CircleCI, GitLab CI/CD, Azure DevOps, and Travis, which come with its own set of advantages and disadvantages over the implementation [10]–[12]. One of the popular tools used is Github Actions due to its direct integration with GitHub as the most significant social coding platform, providing a collaborative repository ecosystem for over 94 million developers in 2022 [13]. Developers prefer GitHub Actions due to its easy-to-use and flexible behavior. Additionally, GitHub Actions offers more than 12,000 reusable components on its marketplace which is 4 times more than CircleCI and 6 times higher than Jenkins [14]. GitHub has excellent support, especially if the application development process involves Git as version control, and GitHub remote repository place to host the software code in the development process.

Most of the time, the process of integrating CI/CD into existing projects is time-consuming caused by the CI/CD tools implementation and architecture, which sometimes are overcomplicated. For example, integration using Jenkins requires developers to learn about Master/Slave Architecture which is implemented on the Jenkins system [15]. Besides that, migration from the conventional approach to CI/CD in software development could be challenging and sometimes require a dedicated DevOps engineer with sufficient knowledge. That complexity is often formed by the ecosystem of the DevOps field and additional tools for specific cloud platforms or software architecture [16], [17]. Creating a complex DevOps ecosystem will require much effort and time, especially in terms of selecting suitable tools, flow planning, monitoring systems, and compatibility with company requirements.

Drawing on numerous past research studies, various use cases and approaches have been employed to achieve successful integration of CI/CD, whether on existing projects or those in their initial stages. Raut et al. implemented a CI/CD pipeline using Jenkins on AWS Cloud architecture alongside additional tools such as Docker, SonarCube, and Trivy [18]. Their research led to the development of a new workflow, triggered by GitHub commits, which initiates Jenkins to run Maven tests and deploy using a shell script and DockerHub. Similarly, Kavya and Smitha conducted research in an AWS environment, employing Jenkins, Docker, and Kubernetes to create a single-container application integrated with a CI/CD flow [19]. Widiyanto et al. presented a slightly different CI/CD for non-AWS server environments by also implementing integration tests and using Gitea instead of GitHub [20]. Alam et al. implemented a CI/CD flow for smaller teams using Jenkins combined with Nexus Repository, Ansible, and Nagios on an AWS server environment [21]. Meanwhile, Railic and Savid's research covered CI/CD implementation in a microservices architecture, resulting in improved efficiency and structure across various software development stages [22]. Reflecting on these studies, the majority opted to use Jenkins as the main CI/CD tool, combined with Docker and various additional tools, to implement CI/CD in the AWS cloud environment. While much of the research focused on implementation details of selected technologies, certain edge cases remained unexplored, such as the influence of repository branching strategy, code review practices, team roles, and the actual benefits of CI/CD integration in the development cycle.

The goal of this research is to showcase a comprehensive exploration of the CI/CD implementation process and its profound impact on established software development methodologies. To achieve this, the Cloversy.id RESTful API project will be utilized as a real-world case study. This research emphasizes the unexplored edge cases particularly branching strategy utilization, the impact of CI/CD on developer responsibility, and its correlation with code review practices, testing processes, and notifications or alert systems. To conclude the research results, a series of tests will be conducted to assess the efficiency difference between development processes with and without CI/CD integration over time. Additionally, an analysis of additional influencing factors of CI/CD will be carried out to support the final research findings.

2. Research Methods

Adapting to the research objective and the implementation of the current system, the phases in this research were organized and divided into 4 primary stages to keep this research on track which is shown in Figure 1.

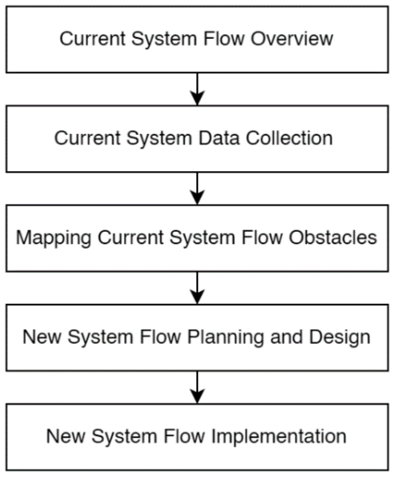

Current System Flow Overview
Current System Data Collection
Mapping Current System Flow Obstacles
New System Flow Planning and Design
New System Flow Implementation

Figure 1. Research Phases

2.1 Current System Flow Overview

To provide an overview of the in-use branching strategy and its relationship with the development team and provide a general view regarding the system implementation, the current development system flow used by the Cloversy.id team RESTful API is shown below in Figure 2.

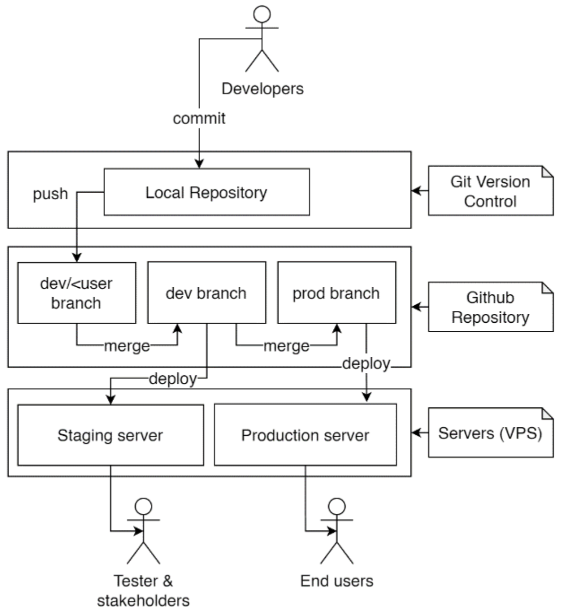


Figure 2. Current Branching Strategy Flow

Figure 2 illustrates the comprehensive delivery process of the Cloversy.id RESTful API application. The development team employs a branching strategy consisting of three types: Firstly, individual developer branches follow the naming convention "dev/<developer_name>", assigned to each developer to monitor individual progress and changes. Secondly, the "dev branch" houses the staging app codebase, while the "prod branch" stores the final production application. Initially, developers make changes to the codebase and commit them locally using Git for version control.

Once a feature is completed, including passing unit and integration testing, developers must push their local changes to the remote repository hosting, which in this case is GitHub. Changes are pushed to the individual developer remote branch "dev/<developer_name>", followed by creating a pull request to the development branch. Every pull request must be well-documented, starting with a descriptive title that indicates the intention of the desired change. Additionally, the pull request should include a detailed description listing all code modifications made by the application developer. The standard for writing pull requests used by the Cloversy.id developer team is straightforward: it involves specifying the type of pull request intent, such as Hotfix, Update, or New Feature, followed by outlining the major changes made to represent this pull request.

Afterward, the pull request will be reviewed by the team leader by running unit tests and integration tests locally to ensure code changes meet the company code standard and fulfill the module's requirements. It's worth noting that the tests applied to this project have been configured to be executed with a single command, facilitating ease of use and efficiency in the testing process.

After the pull request has been reviewed, it is merged into the development branch. Subsequently, the team leader manually deploys the code by accessing the staging Virtual Private Server (VPS) using an SSH connection. The staging app hosted on the staging server is then utilized for stakeholders to conduct testing and undergo User Acceptance Testing (UAT). Once all features on the staging app have been thoroughly tested, passed all tests, and confirmed to be valid, the developer creates another pull request to merge the code from the development branch to the production branch.

Finally, the team leader performs another manual deployment by accessing the VPS with an SSH connection, this time utilizing the production branch and targeting the production server. As a result, at the conclusion of the development cycle, the changes are effectively applied to live applications accessible by end-users. The current system flow architecture employed by the Cloversy.id team for internal app development is illustrated in Figure 3.

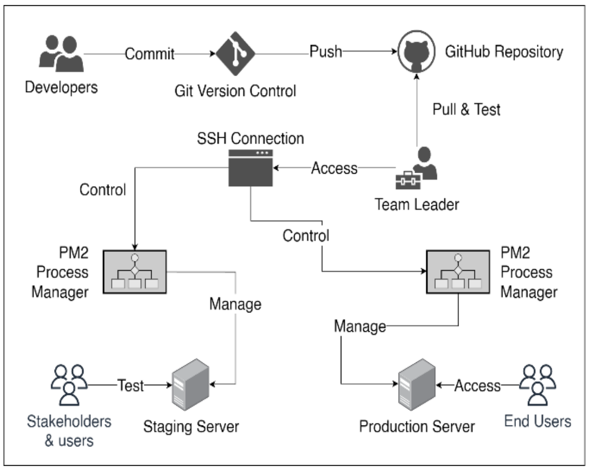


Figure 3. Current System Architecture Flow

Figure 3 shows the current system architecture flow of Cloversy.id RESTful API. There are 4 actors involved, namely developers, team leaders, stakeholders, and end users. Git is used by developers as version control which is connected to the GitHub remote repository, the team leader has the role of controlling which version of the app to deploy and serve in either the staging server or the production server. To manage all of the application deployments, the team leader uses direct SSH connection access to each server, and then the applications on each server are managed using the PM2 process manager.

2.2 Current System Data Collection

The deployment time data for the application on the current system is collected using time-tracking software. The tracker is activated at the start of the deployment process and deactivated once the application is up and running on the server, signifying the completion of the deployment process.

This dataset comprises 10 deployment trials overseen by the developers. Time data, originally recorded in various units by the time tracker application, has been standardized into seconds. This dataset enables the assessment of differences in the total deployment time, serving as a baseline for comparison against post-CI/CD implementation metrics.

2.3. *Mapping Current System Flow Obstacles*

Current development system flowchart of Cloversy.id RESTful API is shown below in Figure 4.

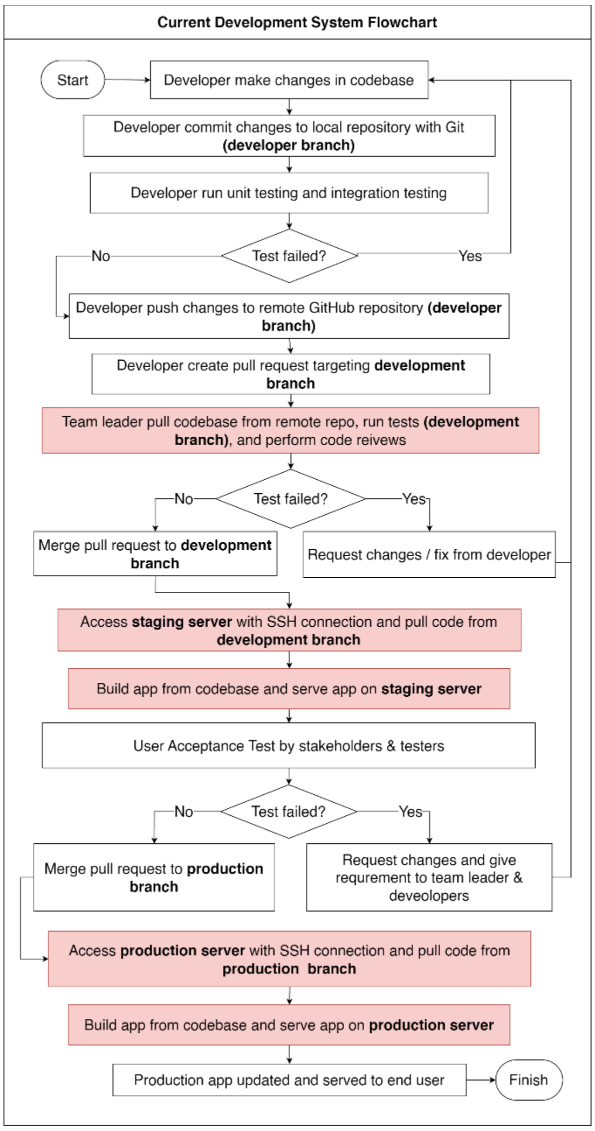

Current Development System Flowchart
Start
Developer make changes in codebase
Developer commit changes to local repository with Git (developer branch)
Developer run unit testing and integration testing
Test failed?
No
Yes
Developer push changes to remote GitHub repository (developer branch)
Developer create pull request targeting development branch
Team leader pull codebase from remote repo, run tests (development branch), and perform code reivews
Test failed?
No
Yes
Merge pull request to development branch
Request changes / fix from developer
Access staging server with SSH connection and pull code from development branch
Build app from codebase and serve app on staging server
User Acceptance Test by stakeholders & testers
Test failed?
No
Yes
Merge pull request to production branch
Request changes and give requrement to team leader & deveolopers
Access production server with SSH connection and pull code from production branch
Build app from codebase and serve app on production server
Production app updated and served to end user
Finish

Figure 4. Current Development System Flowchart

Figure 4 illustrates the flowchart of the current development cycle, showcasing the sequential process of implementing codebase changes and transformations within the system. It delineates the stages starting from developers making modifications to the codebase, including developing new features or rectifying bugs, followed by the team leader conducting code reviews and tests, merging between branches, and ultimately culminating in deployment. The flowchart highlights several activities marked in red, indicating obstacles commonly encountered in the development cycle.

Upon thorough examination of the flowchart, it becomes apparent that several identified constraints manifest as repetitive tasks, notably including pulling codebase from a remote repository, conducting unit and integration tests, accessing VPS, and executing the building and deployment of applications onto their respective servers. These recurring obstacles often consume significant time and effort during the development cycle.

However, these challenges present an opportunity for improvement through the implementation of a CI/CD pipeline. By automating critical processes such as testing, code retrieval, and deployment, organizations can streamline their development workflows, enhance efficiency, and reduce manual intervention. Moreover, CI/CD pipelines facilitate consistent and reliable software delivery, mitigating the risk of human error and accelerating time-to-market for new features and updates.

2.4 New System Flow Planning and Design

The current development system flow used by the Cloversy.id team for RESTful API is shown below in Figure 5.

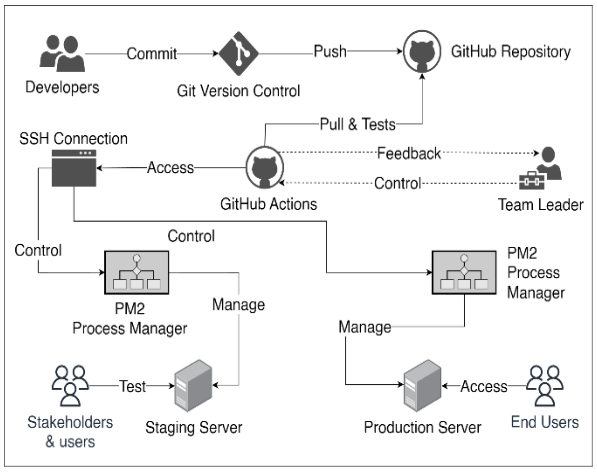


Figure 5. Current System Architecture Flow

Figure 5 shows the design of the new system architecture flow, showcasing the integration of additional tools, notably GitHub Actions, to orchestrate CI/CD pipelines. In the illustration, GitHub Actions assumes the responsibilities previously handled by the team leader, encompassing tasks such as code retrieval, testing, linting, server access, process manager management, and application deployment.

During the CI/CD process, Github actions require full control from the team leader to confirm the merging branch after the code is confirmed to meet the code review requirements. Apart from that, GitHub Actions also provides feedback during the CI/CD process in the form of emails and stack tracks on the GitHub Actions page.

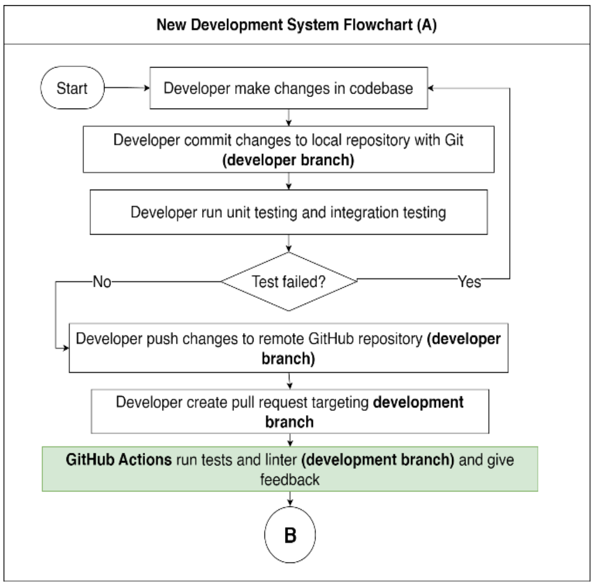

New Development System Flowchart (A)
Start
Developer make changes in codebase
Developer commit changes to local repository with Git (developer branch)
Developer run unit testing and integration testing
Test failed?
No
Yes
Developer push changes to remote GitHub repository (developer branch)
Developer create pull request targeting development branch
GitHub Actions run tests and linter (development branch) and give feedback
B

Figure 6. New Development System Flowchart Design (A)

Figure 6 shows a flowchart illustrating the design of a new system that incorporates Continuous Integration and Continuous Deployment (CI/CD) practices. This part of the design, referred to as Part A of the development flow, represents a notable shift in responsibilities between the team lead and GitHub Actions. In this reimagined system architecture, GitHub Actions assumes a pivotal role, effectively replacing the traditional tasks performed by the team lead. Specifically, GitHub Actions takes charge of pulling remote repositories from the development branch, executing unit tests and integration tests, and generating comprehensive feedback. This feedback is disseminated via email to the team lead, ensuring timely notification of test outcomes, while detailed task logs are accessible within the GitHub Actions menu for comprehensive review and analysis.

The integration of CI/CD brings about significant advantages, particularly in streamlining development workflows and enhancing efficiency. With GitHub Actions handling the code retrieval and testing processes, team leads are relieved of the burden of manual tasks performed on local machines. Consequently, they can allocate more time and resources toward application development and strategic decision-making, fostering greater productivity and innovation within the development team. Figure 7 below shows the continuation of development flow part A, namely development flow part B.

Figure 7 provides a detailed illustration of four processes highlighted in green, these activities are processes that were initially handled by the team leader and in the design were taken over by GitHub Actions. These processes cover critical tasks such as accessing the staging and production servers via SSH connections, retrieving code from both the development and production branches, executing the application build process, and orchestrating deployment using the process manager.

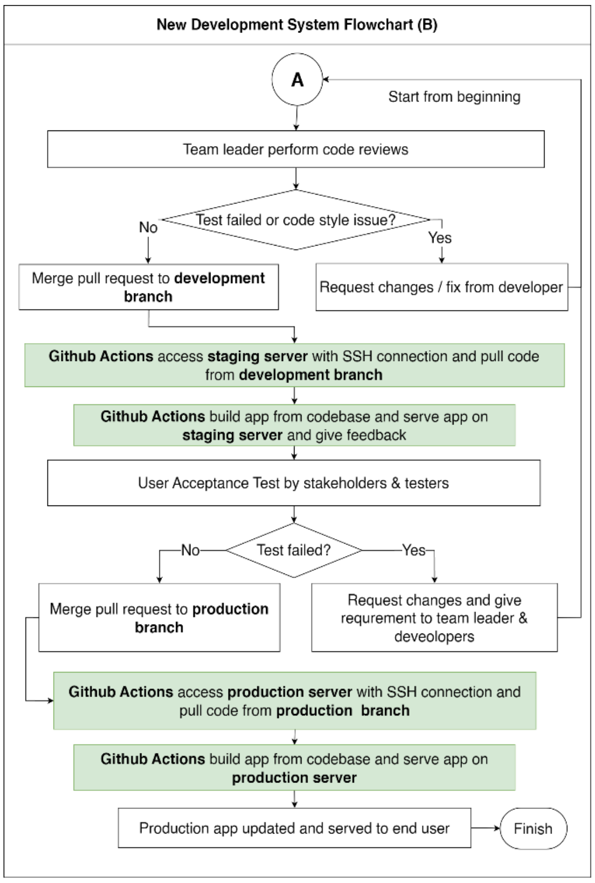

New Development System Flowchart (B)
A
Start from beginning
Team leader perform code reviews
Test failed or code style issue?
No
Yes
Merge pull request to development branch
Request changes / fix from developer
Github Actions access staging server with SSH connection and pull code from development branch
Github Actions build app from codebase and serve app on staging server and give feedback
User Acceptance Test by stakeholders & testers
Test failed?
No
Yes
Merge pull request to production branch
Request changes and give requrement to team leader & deveolopers
Github Actions access production server with SSH connection and pull code from production branch
Github Actions build app from codebase and serve app on production server
Production app updated and served to end user
Finish

Figure 7. New Development System Flowchart Design (B)

The new role assumed by the team leader in this design is confirming the merge after the code quality has been checked, and receiving and monitoring feedback from the GitHub Actions pipeline. The new responsibilities completely reduce the burden and amount of manual work on team leaders to carry out iterative work in the application development process, which can also reduce the possibility of human error in the process.

For the record, current development systems and new designs already implement the DSP model, which divides the development environment into 3 development environments [23]. Namely, development is used by developers to develop and test new features, in this case, local machines. Then the staging environment is used to carry out testing and UAT by stakeholders before the application reaches the end user, and finally, the production environment is used to run and distribute the application to the end user after passing through the previous two environments.

2.5 New System Flow Implementation

In CI/CD implementations using GitHub Actions, developers must create workflows that can be used to define workflow names, task sequences, runner types, and event triggers [24]. The workflow will be run every time a certain event is executed on a certain branch according to the configuration with YAML as a data serialization language. As a solution to the CI/CD implementation of Cloversy.id's RESTFul API system development, 4 workflows are created including 2 workflows for continuous integration and 2 workflows for continuous delivery. Details of workflow usage can be seen in Figure 8.

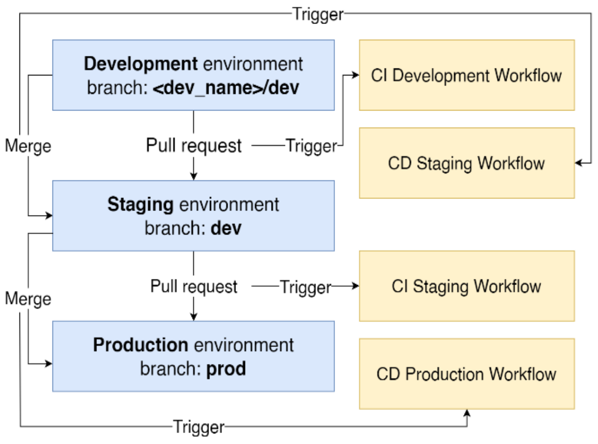


Figure 8. CI/CD GitHub Actions Workflow Implementation

In Figure 8, three types of development environment stages are depicted, represented by processes with a blue background. Each of these environments corresponds to a type of branch created on GitHub following the DSP model. Additionally, the image showcases four GitHub Actions workflows, symbolized by processes with a yellow background. Two types of events trigger these workflows: pull requests and merge actions.

Whenever a developer initiates a pull request with the developer branch as the source and targets the development branch, the "CI Development Workflow" is activated. Subsequently, if the pull request is approved and the merge action is executed, the "CD Staging Workflow" is triggered. Similarly, when a pull request targeting the production branch is created, the "CI Staging Workflow" is activated, followed by the "CD Production Workflow" once the pull request is merged. Detailed information regarding the implementation of the "CI Development Workflow" workflow can be found below.

ci-dev-workflow.yml

```
name: CI Development Workflow
on:
  pull_request:
    branches:
      - dev
jobs:
  test:
    runs-on: ubuntu-latest
    strategy:
      matrix:
        node-version: [14.x]
    steps:
      - uses: actions/checkout@v2
      - name: Use Node.js ${{ matrix.node-version }}
        uses: actions/setup-node@v2
        with:
          node-version: ${{ matrix.node-version }}
      - name: lint and test
        run: |
          echo "$GOOGLE_CLOUD_JSON" > src/config/google-cloud.json
          echo "$FIREBASE_ADMIN_JSON" > src/config/firebase-admin.json
          npm install
          npx eslint .
          npm run test
        env:
          CI: true
          (More env variable…)
```

The code snippet above outlines the configuration for the CI Development workflow. This configuration specifies that the workflow will be triggered whenever a pull request is initiated with the dev branch as the target. GitHub Actions will then execute a sequence of commands to install dependencies and run linters and tests within the Node.js environment, utilizing the specified environment variables and pre-configuration settings. Similarly, the same configuration is utilized in the CI Staging Workflow, with the only distinction being the target branch, which is changed to "prod". For further insight into the implementation details of the CD Staging Workflow, please refer to the following section.

cd-prod-workflow.yml

```
name: CD Production Workflow
on:
  push:
    branches:
      - prod
jobs:
  deploy:
    runs-on: ubuntu-20.04
    steps:
      - name: SSH and deploy app
        uses: appleboy/ssh-action@master
        with:
          host: ${{ secrets.SSH_HOST_PROD }}
          username: ${{ secrets.SSH_USERNAME }}
          password: ${{ secrets.SSH_PASSWORD }}
          port: ${{ secrets.SSH_PORT }}
          script: |
            cd cloversystore
            cd cloversy-store-api
            ${{ secrets.GH_PULL_SCRIPT }}
            npm install
            tsc
            pm2 restart cloversy-store-api
```

The code snippet provides insight into the CD Staging workflow, which is triggered whenever alterations are made to the prod branch, whether through direct pushes as hotfixes or merge actions. This workflow is initiated by establishing an SSH connection to the production server using securely provided credentials. Subsequently, it navigates to the project's working directory, retrieves the latest code from the prod branch, proceeds to install dependencies, performs compilation and building processes, and finally, restarts the process manager to effectively implement the recent changes.

The identical configuration is employed in the CD Staging workflow, with the exception that the branch configuration shifts from prod to dev, and the host address transitions from the production server address to the staging server address. All configurations are encapsulated within files with the .yml extension and are situated in the .github/workflows directory. GitHub Actions automatically detects all workflows stored within this directory.

3. Results and Discussions

To measure the influence that CI/CD implementation has had on the development of RESTful APIs on Cloversy.id, a series of trials were carried out to assess the accumulated time required to pull and test the code by the team leader (CI), and the time required to deploy the code after declared production ready (CD). After that, a certain accumulated time is compared with the time required after CI/CD implementation. The results of this series of tests are used as a basis for assessing the impact of CI/CD implementation in terms of time metrics. The results can be seen in Figure 9.

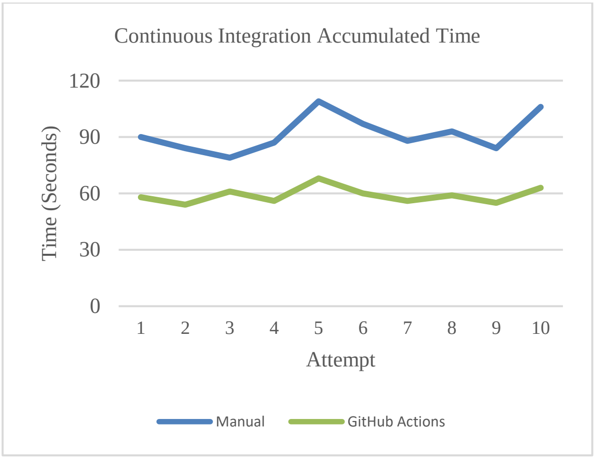


Figure 9. Continuous Integration Time Comparison

Figure 9 provides a comprehensive comparison of the CI (Continuous Integration) process before and after its implementation using GitHub Actions. The results, based on 10 attempts, reveal significant improvements in efficiency following the adoption of automation. On average, the manual CI process, overseen by the team leader, required 92 seconds to complete, while the automated CI process using GitHub Actions reduced the average time to just 59 seconds.

This comparison underscores the tangible benefits of CI implementation with GitHub Actions, showcasing a remarkable 35% reduction in time for tasks such as code pulling, testing, and linting. By automating these essential processes, GitHub Actions expedites the development cycle, enabling faster iterations and enhancing overall productivity. Moreover, Figure 10 offers a glimpse into the accumulated deployment process time (CD) following the implementation of CI/CD practices.

Figure 10 illustrates a comparison of the accumulated time required for application deployment. During 10 trials, manual application deployment averaged 82 seconds, whereas application deployment with GitHub Actions averaged 50 seconds. This demonstrates that leveraging GitHub Actions for CD implementation results in a 39% increase in deployment speed. In addition to the time-based aspect, several supporting factors are evaluated, including the list of workflow execution results displayed in GitHub Actions, as depicted in Figure 11.

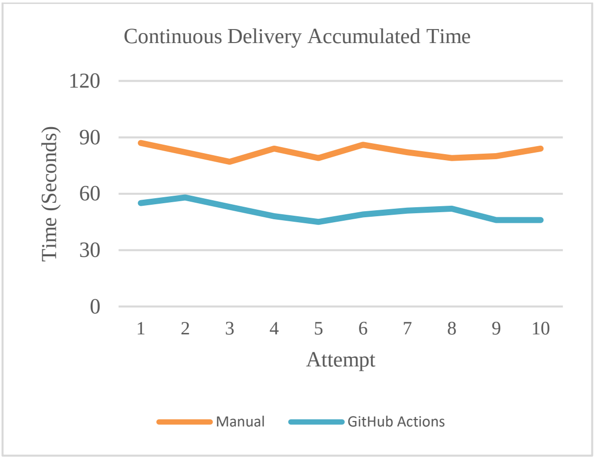
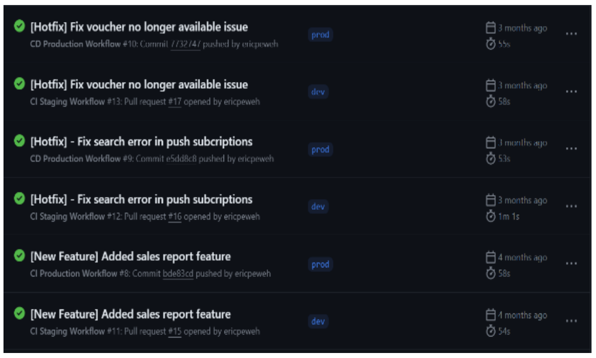


Figure 10. Continuous Delivery Time Comparison


[Hotfix] Fix voucher no longer available issue
CD Production Workflow #10: Commit 7732747 pushed by ericpeweh
prod
3 months ago
55s

[Hotfix] Fix voucher no longer available issue
CI Staging Workflow #13: Pull request #17 opened by ericpeweh
dev
3 months ago
58s

[Hotfix] - Fix search error in push subcriptions
CD Production Workflow #9: Commit e5dd8c8 pushed by ericpeweh
prod
3 months ago
53s

[Hotfix] - Fix search error in push subcriptions
CI Staging Workflow #12: Pull request #16 opened by ericpeweh
dev
3 months ago
1m 1s

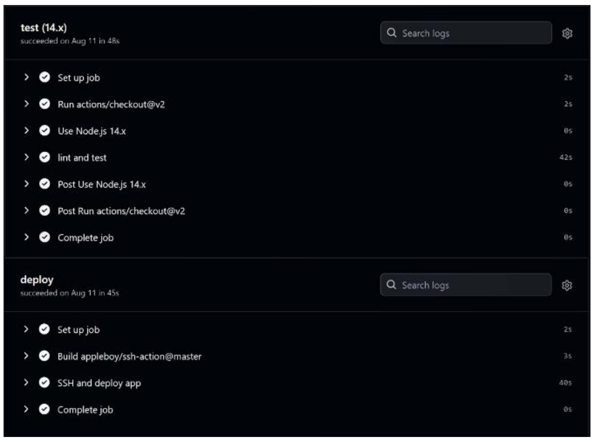
[New Feature] Added sales report feature
CI Production Workflow #8: Commit bde83cd pushed by ericpeweh
prod
4 months ago
58s

[New Feature] Added sales report feature
CI Staging Workflow #11: Pull request #15 opened by ericpeweh
dev
4 months ago
54s

Figure 11. GitHub Actions Workflow History

Through the workflow history shown in Figure 11, developers can effectively perform audits to check whether the workflow is running according to the requirements of the software development Each workflow history item also displays useful information such as the total time required for execution, workflow execution status, target branch, pull request name and description, and others that can be used to evaluate workflow performance. Each workflow history item also displays information that is useful for evaluating workflow performance including logs that could help to locate and trace errors through the execution stack. CI/CD action logs are shown in Figure 12.

Figure 12 shows the execution logs of the CI/CD workflow. If errors or warnings are found, logs can be used as a tool to correct errors and prepare for the next workflow execution. Apart from logs, there are also email notifications sent by GitHub Actions which provide an overview of the status of the workflow being executed. An example of an email sent can be seen in Figure 13.



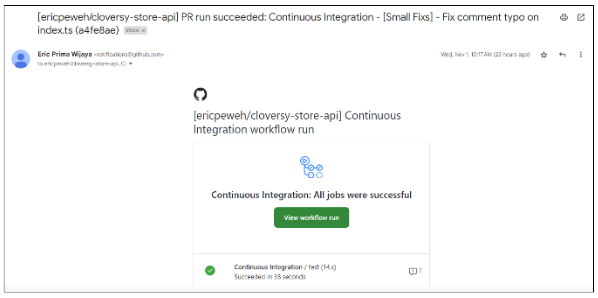
```
test (14.x)
succeeded on Aug 11 in 48s

> Set up job                                2s
> Run actions/checkout@v2                   2s
> Use Node.js 14.x                          0s
> lint and test                            42s
> Post Use Node.js 14.x                     0s
> Post Run actions/checkout@v2              0s
> Complete job                              0s

deploy
succeeded on Aug 11 in 45s

> Set up job                                2s
> Build appleboy/ssh-action@master          3s
> SSH and deploy app                       40s
> Complete job                              0s
```

Figure 12. GitHub Actions Execution Logs



[ericpeweh/cloversy-store-api] PR run succeeded: Continuous Integration - [Small Fixs] - Fix comment typo on index.ts (a4fe8ae)

Eric Prima Wijaya <notifications@github.com>
to ericpeweh/cloversy-store-api, Ci
Wed, Nov 1, 10:17 AM (23 hours ago)

[ericpeweh/cloversy-store-api] Continuous Integration workflow run

Continuous Integration: All jobs were successful

Continuous Integration / test (14.x)
Succeeded in 38 seconds

Figure 13. GitHub Actions Notification Email

The email sent as in Figure 13 is beneficial as a communication tool in the application development process. For example, an email notification reminds the team leader that a new pull request has been created and is waiting for confirmation for merging, or other cases where an email notification notifies that for some reason, the deployment process failed, so developers can immediately follow up regarding the issue.

4. Conclusions

In conclusion, the successful implementation of the CI/CD flow in Cloversy.id's RESTful API development cycle has validated its efficacy in streamlining the software delivery process. This new system leverages CI/CD to automate numerous repetitive tasks previously managed directly by team leaders. The adoption of automated CI/CD flows with GitHub Actions has demonstrated a substantial improvement, achieving a 35% increase in speed for continuous integration and a 39% enhancement for continuous delivery compared to traditional manual methods led by team leaders.

Moreover, the CI/CD implementation addresses various facets of the current development process, including integration with branching strategies, testing procedures, and code reviews, thus bridging critical gaps in the existing literature. Despite the automation, team members continue to fulfill their respective roles post-implementation of the CI/CD pipeline, ensuring collaboration and accountability throughout the development cycle. The adoption of CI/CD introduces several advantageous features such as workflow history, logs, and email notifications, all of which contribute to improving the efficiency of the application development cycle.

There is room for improvement that can be implemented, such as using the conditional GitHub Actions feature to handle exceptions to run fallback tasks or forward notifications using integration with third-party services. From a technical perspective, improvements can be made to address more specific use cases such as handling additional branches such as hotfix branches, team structure mutation, infrastructure transition, and others.

Acknowledgements

This research was carried out with the approval of the owner and development team of Cloversy.id by providing the necessary information to support complete and assist researchers.

References

[1] L. Yuge and T. Badarch, “Research on Contemporary Software Development Life Cycle Models,” *Am. J. Comput. Sci. Technol. Spec. Issue Adv. Comput. Sci. Futur. Technol.*, vol. 6, no. 1, pp. 1–9, 2023, doi: 10.11648/j.ajcst.20230601.11.

[2] S. Kumar, “Reviewing Software Testing Models and Optimization Techniques: An Analysis of Efficiency and Advancement Needs,” *J. Comput. Mech. Manag.*, vol. 2, no. 1, 2023, doi: 10.57159/gadl.jcmm.2.1.23041.

[3] N. Singh, “CI/CD Pipeline for Web Applications,” *Int. J. Res. Appl. Sci. Eng. Technol.*, vol. 11, no. 5, 2023, doi: 10.22214/ijraset.2023.52867.

[4] S. J. Malgund and Sowmyarani C N, “Automating Deployments of The Latest Application Version Using CI-CD Workflow,” *Int. J. Eng. Appl. Sci. Technol.*, vol. 7, no. 5, pp. 99–103, 2022, doi: https://doi.org/10.33564/ijeast.2022.v07i05.017.

[5] A. Mishra and Z. Otaiwi, “DevOps and software quality: A systematic mapping,” *Computer Science Review*, vol. 38. 2020, doi: 10.1016/j.cosrev.2020.100308.

[6] R. Parashar, “Path to Success with CICD Pipeline Delivery,” *Int. J. Res. Eng. Sci. Manag.*, vol. 4, no. 6, pp. 271–273, 2021.

[7] A. Alnafessah, A. U. Gias, R. Wang, L. Zhu, G. Casale, and A. Filieri, “Quality-Aware DevOps Research: Where Do We Stand?,” *IEEE Access*, vol. 9, pp. 44476–44489, 2021, doi: 10.1109/ACCESS.2021.3064867.

[8] S. Ferdian, T. Kandaga, A. Widjaja, H. Toba, R. Joshua, and J. Narabel, “Continuous Integration and Continuous Delivery Platform Development of Software Engineering and Software Project Management in Higher Education,” *J. Tek. Inform. dan Sist. Inf.*, vol. 7, no. 1, 2021, doi: 10.28932/jutisi.v7i1.3254.

[9] J. Fairbanks, A. Tharigonda, and N. U. Eisty, “Analyzing the Effects of CI/CD on Open Source Repositories in GitHub and GitLab,” in *2023 IEEE/ACIS 21st International Conference on Software Engineering Research, Management, and Applications (SERA)*, 2022, pp. 176–181, doi: 10.1109/SERA57763.2023.10197778.

[10] E. Soares, G. Sizilio, J. Santos, D. A. da Costa, and U. Kulesza, “The effects of continuous integration on software development: a systematic literature review,” *Empir. Softw. Eng.*, vol. 27, no. 3, 2022, doi: 10.1007/s10664-021-10114-1.

[11] P. Rostami Mazrae, T. Mens, M. Golzadeh, and A. Decan, “On the usage, co-usage and migration of CI/CD tools: A qualitative analysis,” *Empir. Softw. Eng.*, vol. 28, no. 2, 2023, doi: 10.1007/s10664-022-10285-5.

[12] C. Singh, N. S. Gaba, M. Kaur, and B. Kaur, “Comparison of different CI/CD Tools integrated with cloud platform,” 2019, doi: 10.1109/CONFLUENCE.2019.8776985.

[13] GitHub, “Octoverse - The state of open source software,” 2022.

https://octoverse.github.com/ (accessed Sep. 13, 2023).

[14] A. Decan, T. Mens, P. R. Mazrae, and M. Golzadeh, "On the Use of GitHub Actions in Software Development Repositories," 2022, doi: 10.1109/ICSME55016.2022.00029.

[15] A. S.K, Amrathesh, and D. G. Raju M, "A review on Continuous Integration, Delivery and Deployment using Jenkins," *J. Univ. Shanghai Sci. Technol.*, vol. 23, no. 6, pp. 919–922, 2021, doi: https://doi.org/10.51201/jusst/21/05376.

[16] L. E. Lwakatare *et al.*, "DevOps in practice: A multiple case study of five companies," *Inf. Softw. Technol.*, vol. 114, 2019, doi: 10.1016/j.infsof.2019.06.010.

[17] M. H. Tanzil, M. Sarker, G. Uddin, and A. Iqbal, "A mixed method study of DevOps challenges," *Inf. Softw. Technol.*, vol. 161, 2023, doi: 10.1016/j.infsof.2023.107244.

[18] R. S. W. -, S. B. K. -, K. R. P. -, H. M. T. -, and R. M. R. -, "Implementation of a Continuous Integration and Deployment Pipeline for Containerized Applications in Amazon Web Services Using Jenkins," *Int. J. Multidiscip. Res.*, vol. 5, no. 3, 2023, doi: 10.36948/ijfmr.2023.v05i03.3323.

[19] N. Kavya and P. Smitha, "Deploying and Setting up Ci/Cd Pipeline for Web Development Project on AWS Using Jenkins," *Int. J. Adv. Eng. Manag.*, vol. 4, no. 6, pp. 2325–2332, 2022, doi: 10.35629/5252-040623252332.

[20] A. D. Widiyanto, B. Anindito, and M. N. Al Azam, "Implementation of Docker and Continuous Integration / Continuous Delivery for Management Information System Development," *IJEEIT Int. J. Electr. Eng. Inf. Technol.*, vol. 3, no. 2, 2020, doi: 10.29138/ijeeit.v3i2.1208.

[21] M. M. Alam, A. Arbaz, and S. H. Uddin, "Emerging Continuous Integration Continuous Delivery (CI/CD) For Small Teams," *Math. Stat. Eng. Appl.*, vol. 72, no. 1, pp. 1535–1543, 2023, doi: https://doi.org/10.17762/msea.v72i1.2381.

[22] N. Railic and M. Savic, "Architecting Continuous Integration and Continuous Deployment for Microservice Architecture," 2021, doi: 10.1109/INFOTEH51037.2021.9400696.

[23] A. Hany Fawzy, K. Wassif, and H. Moussa, "Framework for automatic detection of anomalies in DevOps," *J. King Saud Univ. - Comput. Inf. Sci.*, vol. 35, no. 3, 2023, doi: 10.1016/j.jksuci.2023.02.010.

[24] P. Heller, *Automating Workflows with GitHub Actions*. Birmingham: Packt Publishing Ltd, 2021.