Jurnal Teknik Informatika (JUTIF)
Vol. 5, No. 3, June 2024, pp. 747-756

DOI: https://doi.org/10.52436/1.jutif.2024.5.3.1836
p-ISSN: 2723-3863
e-ISSN: 2723-3871

COMPARISON OF JENKINS AND GITLAB CI/CD TO IMPROVE DELIVERY TIME OF BASU DAIRY FARM ADMIN WEBSITE

Alif Babrizq Kuncara[*1], Dana Sulistyo Kusumo[2], Monterico Adrian[3]

[1,2,3]Telkom University, Indonesia
Email: [1]alifbabrizq@students.telkomuniversity.ac.id, [2]danakusumo@telkomuniversity.ac.id, [3]monterico@telkomuniversity.ac.id

(Article received: February 5, 2024; Revision: February 20, 2024; published: May 28, 2024)

Abstract

The Basu Dairy Farm admin website is a web-based information system developed using monolithic architecture. The delivery process of source code changes from the GitLab repository on the "main" branch (development) to the main server (production) takes a long time because the build and deploy process is done manually. This causes the delivery time to be long. To overcome this, this research applies Continuous Integration/Continuous Deployment (CI/CD) as a solution. The CI/CD tools used are Jenkins and GitLab CI/CD because they are open source and the most popular. In this study, a comparison of the delivery time of the two tools was carried out. Delivery time is obtained when the build process starts to run until the deploy process is completed. The analysis includes the time required to run the build and deploy process of the CI/CD tool. The results of this research show that Jenkins and GitLab CI/CD are successfully implemented and can automate the build and deploy process. In terms of implementation, Jenkins requires in-depth configuration, so it looks complicated, while GitLab CI/CD offers simple and easy configuration. In the three experiments conducted, Jenkins showed a faster average time in completing the build and deploy process, so Jenkins has a better delivery time than GitLab CI/CD in the context of the Basu Dairy Farm admin website development process.

Keywords: *build, ci/cd, delivery time, deploy, gitlab ci/cd, jenkins.*

1. INTRODUCTION

Continuous Integration/ Continuous Deployment (CI/CD) is a modern software development practice that revolutionizes development by automating the build and deploy process of software development [1]. By implementing CI/CD, development teams can identify and fix problems early on, thereby reducing the risk of missed errors during software delivery to the server [1]. As such, CI/CD is a modern software development practice that is essential for improving and maintaining the quality and speed of software delivery [2]. CI/CD is also considered to be a critical point for QA to apply such automation methods and tools within a dynamic environment [3].

The research uses a case study on the Basu Dairy Farm admin website development process. The Basu Dairy Farm admin website is developed using monolithic architecture. According to research [4], monolithic architecture is still widely used because of its ease of development and does not involve many problems related to integration, connection, and configuration with all functionality encapsulated into one single application [5]. The delivery process of source code changes from the GitLab repository on the “main” branch (development) to the main server (production) takes a long time because the build and deployment process is done manually by the project manager. This causes the delivery time to be long [6].

In this research, an adaptation of [7] is made so that the deployment process in CI/CD practice. In [7], delivery time is a combination of two phases when doing a pull request, namely the merge phase and the delivery phase. The solution offered is the Basu Dairy Farm website admin information system. Before the implementation of CI/CD, the case study used cloud services for deployment facilities. The application of CI/CD tools with Travis CI in [7] did not necessarily speed up the delivery time of pull requests but improved the contribution processing mechanism on the project. This is done by identifying feasible pull requests, thus reducing the burden on developers.

In its implementation, CI/CD practices require tools to automate the build and deploy process. This tool is the key to facilitating efficient software development based on how easy it is to identify mistakes quickly [8]. We proposed the research object using Jenkins, Jenkins is a CI tool that executes Maven scripts and shell scripts both for Windows and Unix/Linux environments [9]. We proposed the research second object for comparison using GitLab, GitLab is a web-based DevOps workflow application for managing CI/CD [10]. In [11], Jenkins and Gitlab CI/CD were implemented in a microservice-based application by creating an experimental setup and evaluating the deployment time of both tools. The

results show that Jenkins is more efficient in deploying applications to the server.

We proposed to use a cloud platform for this research that was introduced by Microsoft called Azure, a Software as a Service (SaaS) platform that helps organizations manage end-to-end software development and deployment processes effectively [12]. Microsoft Azure automatically builds and tests code projects by combining continuous integration (CI) and continuous delivery (CD) [13]. In this research, we use Microsoft Azure as a Virtual Machine based on Azure features that use virtual networks and policies to allow CI/CD from this machine [14]. Microsoft Azure is very suitable for the project team that uses Microsoft technology for CI/CD purposes [15].

In this research, the most popular open-source CI/CD tools, namely Jenkins and GitLab CI/CD, are implemented on a monolithic architecture and compared against the delivery time obtained from the length of time to run the build and deploy process that has been automated. The selection of CI/CD tools is based on how popular and widely used in the software development industry. The comparison needed to be done so that the tool that has the most advantage for deployment can be implemented on the Basu Dairy Farm admin website. Furthermore, research on the effect of CI/CD tools on delivery time obtained from the length of time to run the build and deploy process to the server on monolithic-based applications has not been explained. Delivery time is focused on the pull request process, while this study focuses on calculating delivery time on CI/CD practices.

2. METHOD

2.1. Existing Conditions

The development process of the admin website at Basu Dairy Farm uses GitLab with two main branches, namely ”dev” and ”main”. The ”dev” branch is used for development, while the ”main” branch is used to deploy to the main server. The development workflow used is as follows.

The Basu Dairy Farm admin website was developed using the monolithic architecture shown in Figure 2. The main characteristic of this approach is that the entire application consists of an integrated whole of various functional parts, running on a single technological environment. Therefore, this architecture does not involve many issues related to integration, connection, and configuration.

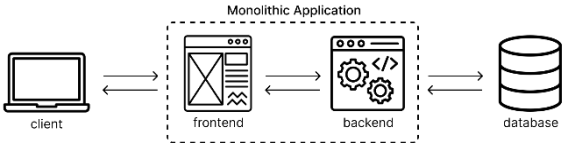

Figure 2. Monolithic Architecture

The delivery process is initiated whenever there are changes or new commits made to the "main" branch. This involves executing the build and deploy process. According to the workflow depicted in Figure 1, whenever there's a modification in the source code, such as additions or deletions, developers push these changes to the GitLab upstream repository within the "dev" branch. Subsequently, the developer merges the latest source code alterations from the "dev" branch into the "main" branch. This merge is then reviewed or checked by a designated team member acting as a reviewer. Once approved, the code is merged from the "dev" branch to the "main" branch.

Frequent small changes to the "main" branch necessitate the project manager to manually pull the latest source code for deployment on the main server. This process is time-consuming, error-prone, and dependent on the manager's availability.

2.2. Architectural Design

The architecture design is a proposed workflow architecture that is used to overcome the problems in the build and deploy process that are still done manually. This workflow architecture applies CI/CD practices to overcome the problems that occur in Figure 1. This architecture replaces the build and deploy (delivery) process that was previously done manually by the project manager (Figure 1) to be automated by using CI/CD tools shown in Figure 3.

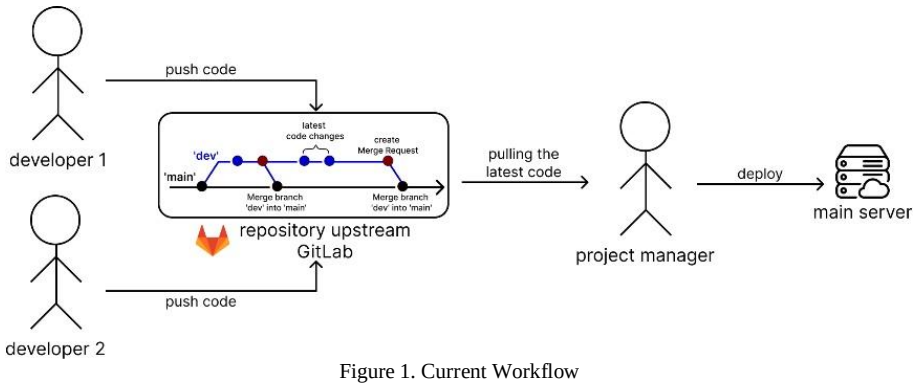

Figure 1. Current Workflow

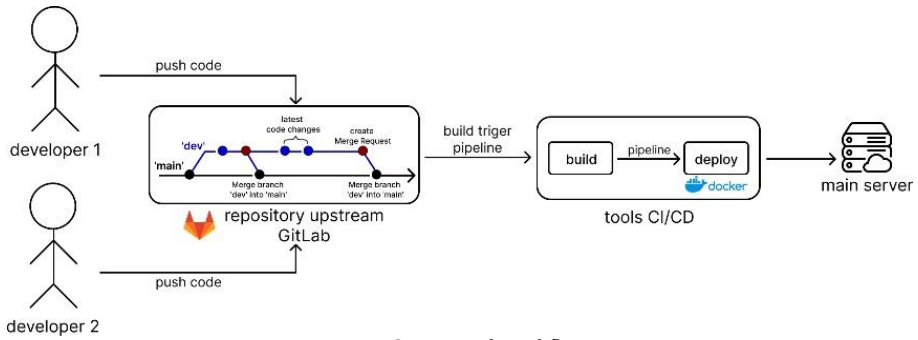

Figure 3. Proposed Workflow

Based on the proposed workflow (Figure 3), the process performed in implementing CI/CD is as follows:

1. Source Code Push: Developers push changes to the "dev" branch in GitLab, which are then merged into the "main" branch.
2. Pipeline Activation: GitLab monitors changes in the "main" branch and triggers the CI/CD pipeline if no issues are detected.
3. Automated CI/CD: The pipeline automates build and deployment using Docker, ensuring the integrity of the system. Failure at any stage halts the process to prevent potential issues.

2.3. Implementation

This stage is to implement the website development workflow with CI/CD practices as in Figure 3 to replace the website development workflow in Figure 1. This implementation uses two CI/CD tools, namely: Jenkins and GitLab CI/CD. Both tools run on a virtual machine server running Linux operating system with the same specifications. The following are the server specifications used to run each tool.

Table 1. Server Specifications

Property	Value
Cloud Platform	Microsoft Azure
Operating System	Linux (Ubuntu 20.04 LTS Gen 2)
Size	Standard D2s v3
vCPUs	2
RAM	8 GiB

1. Jenkins

Jenkins is a popular Java-based and open-source CI/CD tool for Continuous Integration/Continuous Deployment (CI/CD) [11]. Jenkins will perform the deployment process after a code change in the Git repository. In this process, Jenkins will retrieve the latest code from the relevant branch and proceed with the deployment steps according to predefined instructions [17].

Jenkins version 2.436 is integrated with GitLab for automation with the specifications (Table 1). GitLab sends notifications to Jenkins via webhook when new code changes are made. Jenkins pulls the latest code and runs a predefined pipeline from the Jenkins file. This pipeline includes building the project with "npm run build" and storing the compilation results in "/dist". Once built successfully, Jenkins creates a Docker image and deploys it on the main server using the default built-in agent, which shares the server's specifications.

At the build stage, Jenkins will run the ”npm run build” command to compile the software project. The compilation results will be stored in the ”/dist” folder. The contents of the folder will be used for the deployment process. After the build stage is successful, Jenkins will build a Docker image of the project and run it on the main server.

```
pipeline
  agent any
  tools
     nodejs 'NodeJS 16'
  stages
      stage('Build')
        steps
           script
              sh 'npm install --ignore-scripts'
              sh 'npm run build'
     stage('Deploy')
        steps
           script
              sh "docker build -t image."
              sh "docker stop container || true"
              sh "docker rm -f container|| true"
              sh "docker system prune -f"
              sh "docker run -d --name container -p 9000:9000
image"
```

Code 1. Pipeline on Jenkinsfile

2. GitLab CI/CD

GitLab is an open source web-based version control system with built-in CI/CD capabilities. Pipelines are automatically activated when changes are made, until they can be executed on the server [11]. GitLab CI/CD is one of the features provided by GitLab that fulfills all the basic needs of Continuous Integration/Continuous Deployment (CI/CD) in one environment [18].

The research implements GitLab CI/CD on version 16.6 for project management and automation. Configuration involves defining pipelines in ".yml" files and creating executors known as "Runners." Uploaded source code triggers automated pipelines, which are defined in ".gitlab-ci.yml" files and executed by Runners using Docker containers. This setup streamlines the build and deployment processes.

Table 2. Docker Specifications

Property	Value
Operating System	Linux (Ubuntu 20.04.6 LTS)
Architecture	x86_64
CPU	2
Memory	7.704 GiB
Docker Server Version	24.0.7

During the build stage, GitLab CI/CD compiles the project using "npm run build" and stores the result in "/dist". This folder's contents are used for deployment. After a successful build, GitLab CI/CD creates a Docker image of the project and deploys it on the main server according to specified specifications (Table 2).

```
stages:
  - build
  - deploy
build:
  image: node:16
  stage: build
  only:
    - main
  script:
    - npm install --ignore-scripts
    - npm run build
  artifacts:
    paths:
      - dist/
deploy:
  image: docker
  stage: deploy
  only:
    - main
  script:
    - docker build -t image .
    - docker stop conatiner || true
    - docker rm conatiner || true
    - docker system prune -f
    - docker run -d --name conatiner -p 9001:9001 image
```

Code 2. Pipeline on gitlab-ci.yml

3. DELIVERY TIME CALCULATION

The study evaluates delivery time (Dt) from build initiation to deployment completion using Jenkins and GitLab CI/CD. Dt is calculated as the sum of build time (t1) and deployment time (t2), expressed as Dt = t1 + t2.

This statement is based on research [3] which explains the calculation of delivery time from pull requests that have been merged. Delivery time in [3] summarizes the two pull request phases, namely the merge phase (t1) and the delivery phase (t2). The difference between this research and research [3] lies in the focus and method of measuring delivery time. In [3], the delivery time calculation is focused on the pull request process, by dividing the process into two phases: the merge phase (t1) and the delivery phase (t2). Meanwhile, this study focuses the delivery time calculation on the Continuous Integration/Continuous Deployment (CI/CD) process. Here, the delivery time (Dt) is calculated by summing the time required to run the build (t1) and deploy (t2) processes.

The calculation begins with pushing identical source code to different GitLab repositories via merge requests from the "dev" branch to the "main" branch. Changes to the "main" branch trigger pipelines run by Jenkins and GitLab CI/CD (Figure 4). The calculation occurs during the execution of commands in the "script{...}" tag in the build and deploy stages of each pipeline. The start and end times of these processes are recorded to determine their duration. Three experiments were conducted, involving frequent changes to the "main" branch of the Basu Dairy Farm admin website. These changes included additions, additions and deletions, and line subtractions. Each experiment was executed once due to limited server resources.

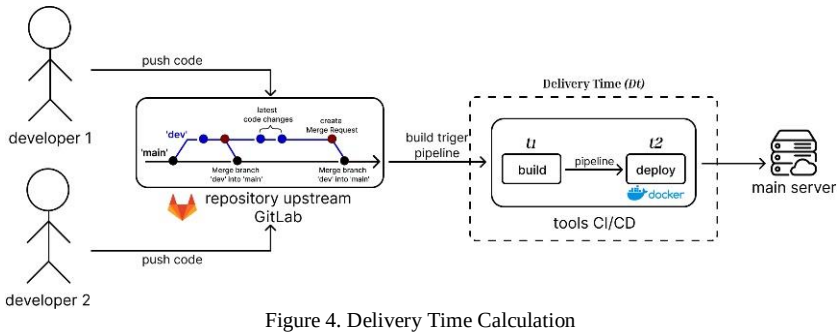

Figure 4. Delivery Time Calculation

1. Delivery Time (Dt) in Jenkins

1) Experiment 1

In experiment 1, a merge request was made containing changes in the form of 22240 code line additions and 0 code line deletions. The result for experiment 1 in Jenkins is represented in Table 3.

Table 3. Experiment 1 on Jenkins

Process	Start	End	Result (s)
Build (t1)	15:29:57	15:30:14	17s
Deploy (t2)	15:30:14	15:30:18	4s
Delivery Time (Dt)			21s

2) Experiment 2

In experiment 2, a merge request was made containing changes in the form of 4486 additions of lines of code and 4486 deletions of lines of code. The result for experiment 2 in Jenkins is represented in Table 4.

Table 4. Experiment 2 on Jenkins

Process	Start	End	Result (s)
Build (t1)	16:52:17	16:52:30	13s
Deploy (t2)	16:52:30	16:52:33	3s
Delivery Time (Dt)			16s

3) Experiment 3

In experiment 3, a merge request was made containing changes in the form of 0 additions of lines of code and 22240 deletions of lines of code. The result for experiment 3 in Jenkins is represented in Table 5.

Table 5. Experiment 3 on Jenkins

Process	Start	End	Result (s)
Build (t1)	17:45:13	17:45:24	11s
Deploy (t2)	17:45:24	17:45:28	4s
Delivery Time (Dt)			15s

2. GitLab CI/CD

1) Experiment 1

In experiment 1, a merge request was made containing changes in the form of 22240 code line additions and 0 code line deletions. The result for experiment 1 in GitLab CI/CD is represented in Table 6.

Table 6. Experiment 1 on GitLab CI/CD

Process	Start	End	Result (s)
Build (t1)	15:49:34	15:49:54	20s
Deploy (t2)	15:50:32	15:50:36	4s
Delivery Time (Dt)			24s

2) Experiment 2

In experiment 2, a merge request was made containing changes in the form of 4486 additions of lines of code and 4486 deletions of lines of code. The result for experiment 2 in GitLab CI/CD is represented in Table 7.

Table 7. Experiment 2 on GitLab CI/CD

Process	Start	End	Result (s)
Build (t1)	17:08:11	17:08:38	27s
Deploy (t2)	17:10:02	17:10:07	5s
Delivery Time (Dt)			32s

3) Experiment 3

In experiment 3, a merge request was made containing changes in the form of 0 additions of lines of code and 22240 deletions of lines of code. The result for experiment 3 in GitLab CI/CD is represented in Table 8.

Table 8. Experiment 3 on GitLab CI/CD

Process	Start	End	Result (s)
Build (t1)	17:58:53	17:59:13	20s
Deploy (t2)	17:59:39	17:59:43	4s
Delivery Time (Dt)			24s

4. RESULTS AND DISCUSSION

4.1. CI/CD Implementation Results

This is used to determine the results of applying the workflow proposed in Figure 3 to the Basu Dairy Farm admin website development process. The workflow was tested using Jenkins and GitLab CI/CD.

1. Jenkins

The developer pushes the changes to the ”dev” branch with the message ”Update README.md”. Then the developer makes a merge request from the ”dev” branch to the ”main” branch (Figure 5). SHA ”71a14d37ff7dc66d25cf05d34de35d328506c33c".

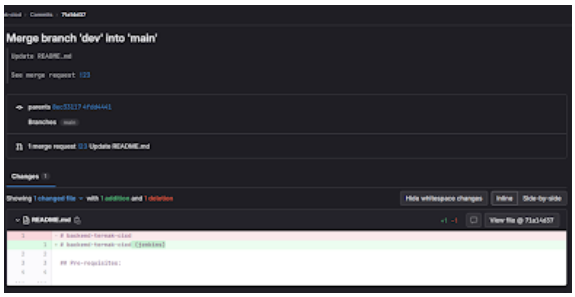

Figure 5. MR on Jenkins

Changes in the ”main” branch trigger a pipeline in Jenkins via a webhook, and Jenkins will run the pipeline with the set of commands defined in Code 1 (Figure 6). The pipeline automates the build and deploy process. Here is an overview of the successful pipeline.

Figure 6. Jenkins Pipeline

2. GitLab CI/CD

The developer pushes the changes to the ”dev” branch with the message ”Update README.md”. Then the developer makes a merge request from the ”dev” branch to the ”main” branch (Figure 7). The following is an overview of the merge request that has been merged with the SHA commit ”4fd7838b24b2459c5dd173f1141efb30f1551073".

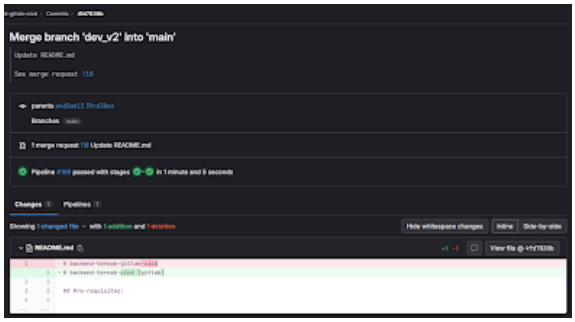

Figure 7. MR GitLab CI/CD

Changes in the ”main” branch trigger the pipeline in GitLab CI/CD, then the pipeline will be run by the ”Runner”. The ”Runner” will automatically run the pipeline with the set of commands defined in Code 2 (Figure 8). The pipeline automates the build and deploy process. Here is an overview of the successful pipeline.

From the results of CI/CD implementation (Figure 6, Figure 8), it can be concluded that the application of the workflow proposed in Figure 3 in the Basu Dairy Farm admin website development

process has been successfully implemented. In other words, the implementation of the CI/CD process using Jenkins and GitLab CI/CD was successfully implemented. Both tools can automate the build and deploy process in the Basu Dairy Farm admin website development process. This is evidenced when there is a source code change in the ”main” branch, Jenkins and GitLab CI/CD automatically run the pipeline to perform the build and deploy process. So that any source code changes can be automatically uploaded and run on the main server.

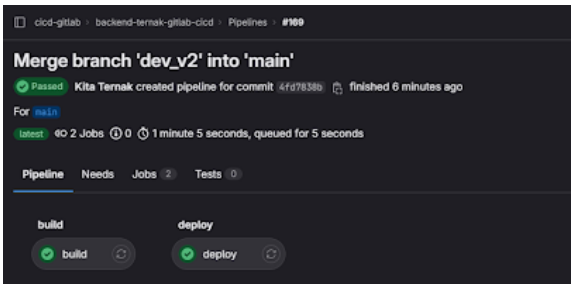

Figure 8. GitLab CI/CD Pipeline

4.2. Delivery Time Calculation Result

A graph is presented from the results of the Jenkins and GitLab CI/CD delivery time (Dt) calculations that have been carried out in the Delivery Time Calculation chapter. This graph presents a visual depiction of the length of time from the build (t1), deploy (t2), and delivery time (Dt) processes. Here is a visual depiction of the delivery time (Dt) calculation in 3 experiments.

1. Experiment 1 Results

During the first experiment, Jenkins and GitLab CI/CD executed a pipeline incorporating alterations totaling 22,240 lines of added code, while no lines were removed. The resulting delivery time (Dt) calculation outcomes for experiment 1 are visually depicted. The result for experiment 1 will be represented in Figure 9 and Figure 10.

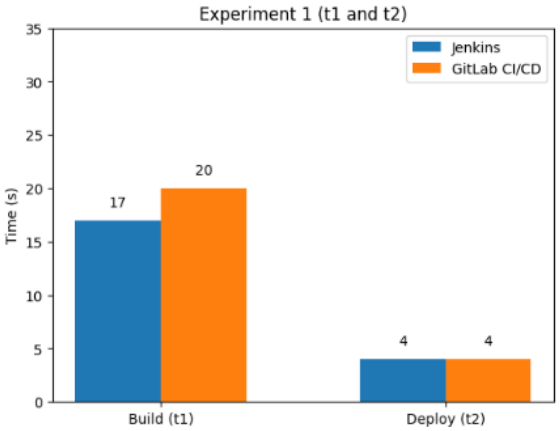

Figure 9. Experiment 1 (t1 and t2)

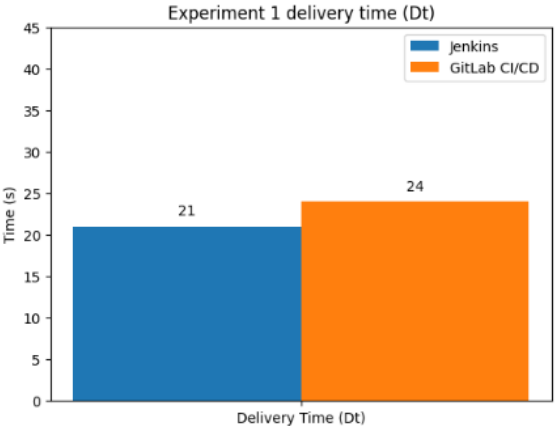

Figure 10. Experiment 1 (delivery time)

2. Experiment 2 Results

In the second experiment, Jenkins and GitLab CI/CD processed a pipeline involving modifications comprising 4,486 lines of code additions and an equal number of deletions. The delivery time (Dt) calculation results from experiment 2 are illustrated for analysis. The result for experiment 2 will be represented in Figure 11 and Figure 12.

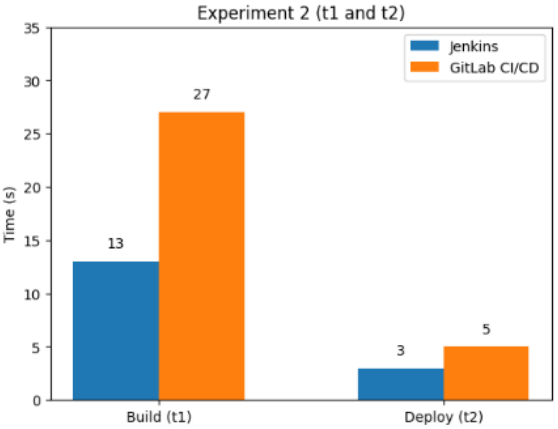

Figure 11. Experiment 2 (t1 and t2)

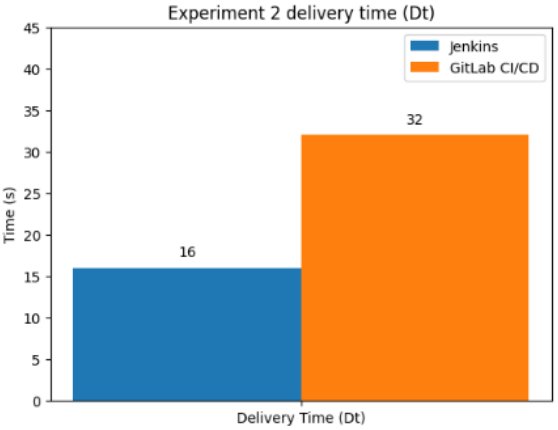

Figure 12. Experiment 2 (delivery time)

3. Experiment 3 Results

Experiment 3 witnessed Jenkins and GitLab CI/CD executing a pipeline with alterations characterized by the removal of 22,240 lines of code

without any additions. The delivery time (Dt) calculation findings from this experiment are presented graphically. The result for experiment 3 will be represented in Figure 13 and Figure 14.

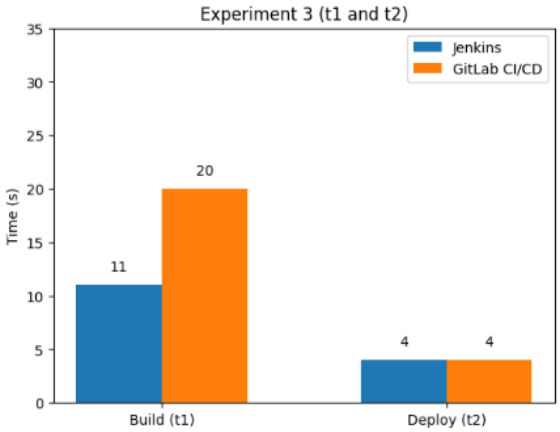

Figure 13. Experiment 3 (t1 and t2)

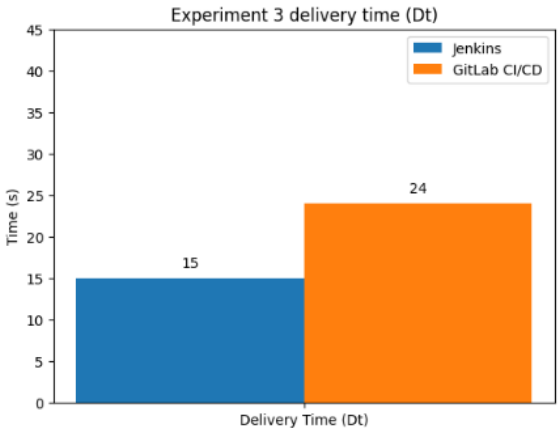

Figure 14. Experiment 3 (delivery time)

Regarding the analysis of the results of the delivery time (Dt) calculation in three experiments, Jenkins proved to have a faster delivery time than GitLab CI/CD. The results of this analysis are presented in graphical form in Figure 15.

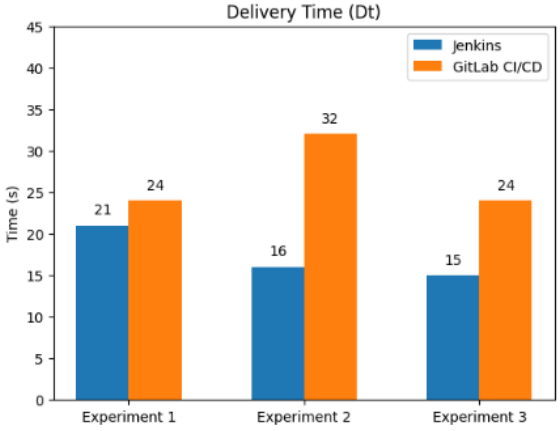

Figure 15. Delivery time (Dt) in 3 experiments

On the first experiment, Jenkins took 21s, while GitLab CI/CD took 24s. On the second experiment, Jenkins took 16s, while GitLab CI/CD took 32s. And in the third experiment, Jenkins completed the process in 15s, while GitLab CI/CD required 24s.

Below is a graph comparing the average durations of Jenkins and GitLab for completing the build t1, deploy t2, and delivery time (Dt) processes.

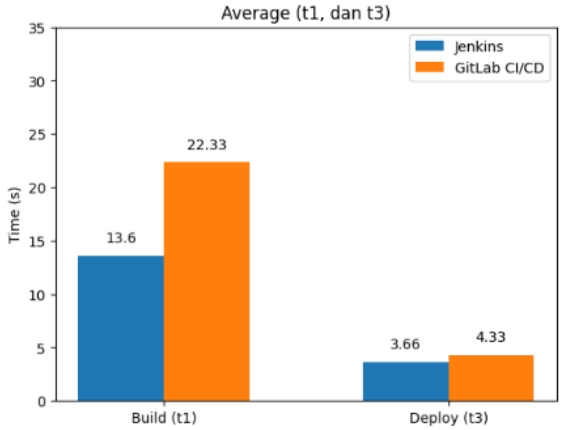

Figure 16. Average (t1 and t2)

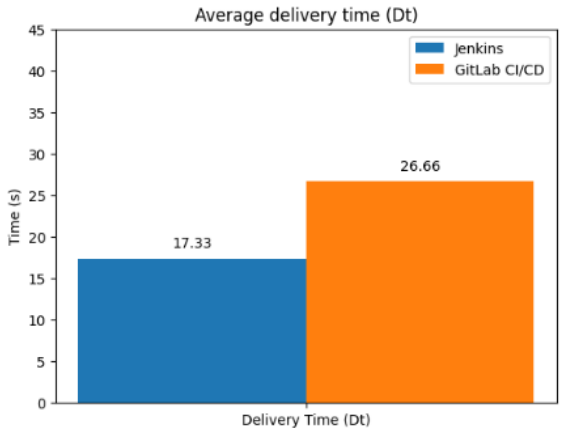

Figure 17. Average (delivery time)

From the data that has been presented, it can be seen that the significant difference in delivery time (Dt) between Jenkins and GitLab CI/CD is due to the considerable difference in the time required to run the build process t1. When running the build process t1, Jenkins and GitLab CI/CD require JS nodes to execute the commands in the build stage. Jenkins utilizes the ”NodeJS” plugin so it already has JS nodes installed in the agent ”built-in”. On the other hand, GitLab CI/CD needs to install the JS node using docker first with the "node" image (node:16) so it takes longer. The time difference is also caused by the different resources (RAM) of the executors used. Jenkins runs the pipeline internally on a ”built-in” agent with a resource capacity (RAM) of 8 GiB, which means that the build and deploy process is done within the Jenkins system itself. Meanwhile, GitLab CI/CD runs the pipeline externally using Docker containers, with a resource capacity (RAM) of 7.704 GiB. In this context, ”run internally” means that Jenkins uses resources integrated in its system, while ”run externally” in GitLab CI/CD indicates the use of

resources from outside the main system, namely through the use of Docker as an executor in a separate container.

The graph also shows that the deploy t2 process takes less time than the build t1 process. This is because the deploy t2 process uses a project that has been compiled in the build t1 process, so the project size becomes smaller. Jenkins stores the build results t1 in the workspace directory provided by Jenkins. While GitLab CI/CD stores the build results t1 in artifacts.

From the graph that has been presented, it can be concluded that Jenkins has a faster average time in completing the build t1 and deploy t2 processes compared to GitLab CI/CD. In terms of delivery time (Dt), Jenkins also shows a faster delivery time compared to GitLab CI/CD. Jenkins takes an average of about 17.33s to deliver changes to the main server, while GitLab CI/CD takes an average of about 26.66s.

5. DISCUSSION

In this study, Continuous Integration/Continuous Deployment (CI/CD) practices were implemented on the development of the Basu Dairy Farm admin website using popular open source CI/CD tools, namely Jenkins and GitLab CI/CD. The results showed that the CI/CD implementation successfully automated the build and deploy process, replacing the manual way previously done by project managers.

Jenkins requires in-depth configuration for implementation, making it seem complicated for beginners. On the other hand, GitLab CI/CD offers a simpler configuration and is easy to implement because it is directly integrated into the GitLab platform. This shows the advantages of each CI/CD tool in terms of ease of configuration. In the context of delivery time comparison between Jenkins and GitLab CI/CD, the results show that Jenkins has a faster delivery time than GitLab CI/CD. This statement influenced by several factors, such as the installation process of JS nodes using docker required by GitLab CI/CD, as well as the different resources (RAM) of the pipeline executor used by each tool. Jenkins runs the pipeline on a "built-in" agent with larger resources (RAM) compared to GitLab CI/CD which uses Docker containers.

Based on research [11], to determine the use of CI/CD tools from Jenkins or Gitlab CI/CD implemented in microservice architecture applications, it is determined based on the configuration of the two tools. Where Gitlab CI/CD uses configuration in one yml file which is easier to configure than Jenkins. Research [11] also states that Gitlab CI/CD is better for small-scale projects than Jenkins. However, in the following research, we found that the use of Jenkins can be said to be better than Gitlab CI/CD based on the delivery time that we have researched on the Basu Dairy Farm admin website case study which can be said to be a small-scale and monolithic-based project. This can be the main reason for choosing Jenkins as a CI/CD tool over Gitlab CI/CD by considering the delivery time compared from 3 experiments on both tools.

Meanwhile in research [7], with a research focus on the effect of Continuous Integration (CI) using the Travis CI tool on the delivery time of the pull request process obtained from the summation of the time of the two phases when making a pull request, namely the merging phase and the delivery phase. The result is that the application of Continuous Integration (CI) using the Travis CI tool does not speed up the delivery time of the pull request process, but improves the contribution processing mechanism on the project by identifying feasible pull requests, thereby reducing the burden on developers. In this study, researchers focused on the effect of CI/CD practices on delivery time. Delivery time is calculated as the total time required for the build and deploy process. This is different from research [7] which focuses delivery time on the pull request process. We chose to focus on delivery time on CI/CD practices because we wanted to measure the impact of CI/CD practices on the software development process (build and deploy). The results found that using Jenkins compared to Gitlab CI/CD had different delivery times, with Jenkins time being faster than Gitlab CI/CD on a monolithic application (Basu Dairy Farm admin website).

The main strength of this research is the proof of CI/CD tool Jenkins, despite requiring extensive configuration, Jenkins shows faster delivery time compared to GitLab CI/CD in the context of developing a monolithic Basu Dairy Farm admin website. Although previous studies, such as [11], show a preference for GitLab CI/CD for small-scale microservice-based projects, our findings highlight that Jenkins can be a better choice for smaller-scale monolithic-based projects. This research also enriches the understanding of the impact of CI/CD practices on the software development process, by focusing on delivery time as a performance indicator, as opposed to focusing on the pull request process as done in previous research [7]. Thus, the results of this study provide valuable insights for software developers in selecting CI/CD tools that suit project needs, as well as to deepen the understanding of CI/CD effectiveness in the context of monolithic-based software development.

6. CONCLUSIONS

Based on the research conducted, CI/CD practices were successfully implemented in the development of the Basu Dairy Farm admin website using Jenkins and GitLab CI/CD. CI / CD practices can automate the build and deploy process, so that it can replace the manual method carried out by project managers. Based on this research, Jenkins requires in-depth configuration in terms of implementation, so it

looks complicated for beginners. On the other hand, GitLab CI/CD offers simple configuration and easy implementation. Jenkins needs to configure and install plugins that suit the needs of the project. Whereas in GitLab CI/CD, there is no need to install anything for CI/CD, because this feature has been integrated directly in the GitLab platform. So it can be concluded that GitLab CI/CD has a simpler configuration than Jenkins in the context of CI/CD tool implementation.

In the context of the delivery time comparison, which was calculated in 3 trials due to the limited resources of the server used, Jenkins showed a faster delivery time than GitLab CI/CD. When running the build and deploy process, Jenkins also shows a faster average time compared to GitLab CI/CD. This time difference is influenced because GitLab CI/CD needs to install JS nodes using docker, while Jenkins utilizes plugins that are already installed. The time difference is also caused by the different resources (RAM) of the pipeline executor from each CI/CD tool. Jenkins runs the pipeline on a "built-in" agent with a resource (RAM) of 8 GiB, while GitLab CI/CD runs the pipeline using a Docker container with a resource (RAM) of 7.704 GiB. Therefore, it can be concluded that Jenkins can improve delivery time better than GitLab CI/CD in the context of the Basu Dairy Farm admin website development process. In future research, it is expected to research techniques to shorten the build time and add other aspects to the delivery time calculation such as the automatic testing process. It would be better if you can take measurements more than once for the three experiments, in order to find out whether the measurement time is consistent or there are variations from the measurement results.

REFERENCES

[1] Sheeba, G. Shidaganti, and A. P. Gosar, "A comparison study on various continuous integration tools in software development," in *Lecture notes in networks and systems*, 2020, pp. 65–76. doi: 10.1007/978-981-15-7106-0_7.

[2] Shahin, Mojtaba, M. A. Babar, and L. Zhu. "Continuous integration, delivery and deployment: a systematic review on approaches, tools, challenges and practices." *IEEE access* 5 (2017): 3909-3943.

[3] Bobrovskis, Sergejs, and A. Jurenoks. "A Survey of Continuous Integration, Continuous Delivery and Continuos Deployment." *BIR workshops*. 2018.

[4] K. Gos and W. Zabierowski, "The Comparison of Microservice and Monolithic Architecture," 2020 IEEE XVIth International Conference on the Perspective Technologies and Methods in *MEMS Design (MEMSTECH), Lviv, Ukraine*, 2020, pp. 150-153, doi: 10.1109/MEMSTECH49584.2020.9109514.

[5] Ponce, Francisco, G. Márquez, and H. Astudillo. "Migrating from monolithic architecture to microservices: A Rapid Review." *2019 38th International Conference of the Chilean Computer Science Society (SCCC)*. IEEE, 2019.

[6] S. R. Dileepkumar and J. Mathew. "Optimize Continuous Integration and Continuous Deployment in Azure DevOps for a controlled Microsoft. NET environment using different techniques and practices." *IOP Conference Series: Materials Science and Engineering*. Vol. 1085. No. 1. IOP Publishing, 2021.

[7] J. H. Bernardo, D. A. da Costa, D.A., U. Kulesza, et al. "The impact of a continuous integration service on the delivery time of merged pull requests," *Empir Software Eng*. vol. 28, no. 97. 2023. doi: https://doi.org/10.1007/s10664-023-10327-6.

[8] Bernardo, J. Helis, D. A. da Costa, and U. Kulesza. "Studying the impact of adopting continuous integration on the delivery time of pull requests." *Proceedings of the 15th International Conference on Mining Software Repositories*. 2018.

[9] Mohammad, S. Mohsienuddin. "Continuous integration and automation." *International Journal of Creative Research Thoughts (IJCRT), ISSN* (2016): 2320-2882.

[10] J. Hembrink, and P. G. Stenberg. "Continuous integration with jenkins." *Coaching of Programming Teams (EDA 270), Faculty of Engineering, Lund University, LTH* (2013): 23.

[11] C. Singh, N. S. Gaba, M. Kaur and B. Kaur, "Comparison of Different CI/CD Tools Integrated with Cloud Platform," *2019 9th International Conference on Cloud Computing, Data Science & Engineering (Confluence), Noida, India*, 2019, pp. 7-12, doi: 10.1109/CONFLUENCE.2019.8776985.

[12] Arefeen, Mohammed Shamsul, and Michael Schiller. "Continuous Integration Using Gitlab." *Undergraduate Research in Natural and Clinical Science and Technology Journal* 3 (2019): 1-6.

[13] H. Nguyen, "Continuous Integration for Embedded environment." (2022).

[14] Debroy, Vidroha, S. Miller, and L. Brimble. "Building lean continuous integration and delivery pipelines by applying devops principles: a case study at varidesk." *Proceedings of the 2018 26th ACM*

Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. 2018.

[15] Ferdian, Sendy, et al. "Continuous Integration and Continuous Delivery Platform Development of Software Engineering and Software Project Management in Higher Education." *Jurnal Teknik Informatika dan Sistem Informasi*, vol. 7, no. 1, 2021.

[16] C. Trubiani, P. Jamshidi, J. Cito, W. Shang, Z. M. Jiang, and M. Bor, "Performance Issues? Hey DevOps, Mind the Uncertainty. in *IEEE Software*. vol. 36. no. 2. pp. 110-117. 2019, doi: 10.1109/MS.2018.2875989.

[17] D. Wijayanto, A. Firdonsyah, and F. D. Adhinata, "Implementasi Continous Integration/Continous Delivery Menggunakan Process Manager 2 (Studi Kasus: SIAKAD Akademi Keperawatan Bina Insan)," *Teknika*, vol. 10, no. 3, pp. 181–188, Oct. 2021, doi: 10.34148/teknika.v10i3.400.

[18] GitLab. What is CI/CD?. [Online]. Available at: https://about.gitlab.com/topics/ci-cd/ (accessed Jan. 10, 2024).