Implementation and Analysis of Container Image Optimization Using Alpine Linux and Multi-Stage Builds

Mochamad Rizal Fachrudin[1*], Arif Saivul Affandi[2]

[1,2]Department of Information System, Faculty of Information Technology, Universitas Merdeka Malang, Malang, East Java, Indonesia
E-mail: [1*]mrizalf.email@gmail.com, [2]fandi@unmer.ac.id

(Received: 31 Oct 2024, revised: 12 Nov 2024, accepted: 13 Nov 2024)

Abstract

Containerization enables isolation within a host, with Docker being a popular tool for packaging applications and their dependencies in container images. However, challenges like slow build processes and bloated image sizes can consume resources, slow down builds, and pose security risks. This study optimizes Docker images by combining the Alpine base image with multi-stage builds, analyzing size, build speed, and security across different combinations and environments to identify and propose the most efficient combination solution. The approach used is a quantitative quasi-experiment with a within-subject design. The sample used was a JavaScript framework, with the main experimental group being the combination of Alpine and multi-stage builds, while the comparison group included combinations of Node and Node-Alpine, both in single-stage and multi-stage configurations, as well as single-stage Alpine. Data was obtained from CI/CD, container registry, and Trivy reports. Analyzed by descriptive analysis, One-Way ANOVA or Kruskal Wallis test, and post-hoc test. The results show that combining multi-stage builds with Alpine is considered best practice because it produces the smallest image size, reducing it by up to 94% compared to single-stage Node. It also achieves the shortest build times across all environments and presents low vulnerability issues. However, it is important to note that while the Alpine multi-stage combination offers the most efficient build times, it experiences a 1.3x increase in duration in low-spec environments.

Keywords: Optimization, Docker Image, Size, Building Time, Vulnerability.

I. INTRODUCTION

Containerization is a technological advancement that enables system isolation within a host. Several containerization technologies exist, such as Linux Containers (LXC), Docker, and Podman. These technologies differ from RunC and Containerd, as the latter are runtime engines used to execute containers [1]. In research conducted by Tarasiuk et al., LXC containerization technology yielded the best results across several resource aspects, particularly in CPU and memory performance, demonstrating its efficiency compared to other technologies [2]. However, each of these technologies has a different focus. For instance, LXC is used for OS-level container virtualization, while Docker and Podman are designed for application-level container virtualization, with Docker offering the best performance in image processing compared to the others [3]. Although applications running on Podman have shown better performance than those on Docker, Docker provides more efficient resource utilization compared to Podman [4], [5]. According to a survey by the international forum StackOverflow [6], Docker is the most popular container technology in use today, and its flexibility in Dockerfile configuration can also be applied to other application-level container technologies like Podman, making it a suitable technology for further discussion. With Docker, applications, along with all their dependencies and environments, can be packaged and run within a single container [7]. Docker also ensures that applications run consistently across different environments by creating a virtualized layer where the application operates [8]. Additionally, Docker's container technology provides ease and speed in the deployment process, making it an ideal choice for implementation [9].

Docker provides a more advanced and lightweight mechanism compared to hypervisor-based virtualization software [10]. Due to the isolated nature of containerization, Docker allows multiple applications to run simultaneously on a single server, reducing the need for traditional servers and lowering hardware costs [11]. With these numerous advantages, the implementation of containerization using Docker has become widespread, including in Indonesia. The adoption of modern lifecycle practices such as DevSecOps is

DOI: 10.34148/teknika.v14i1.1118

another driving factor, as containerization is a key component in the successful implementation of these practices.

However, in practice, since this technology uses images to run containers, the main challenge, especially when using Docker, is the large image size. Large images can increase storage usage and slow down build times. By default, running a Docker container requires pulling a base image from Docker Hub, so the size of the base image used will also affect the build speed [12]. Beyond base image size, the bloating of Docker images is also caused by the scale of large projects and the storage of unnecessary artifacts. As image sizes increase, so does the potential for vulnerabilities, as more dependencies or files with security risks may be packaged within the container. This issue must be addressed before the application is deployed since security is a critical aspect of any application.

One approach to addressing this issue is using lightweight base images, such as Alpine Linux [13]. Alpine Linux has recently gained popularity due to its small size and strong security features [14]. In a study by Tipantuña et al., using the Alpine Linux base image successfully reduced resource usage in a Raspberry Pi environment, while another study by Fava et al. found that Alpine Linux provided 20% better memory savings compared to Debian images [15], [16]. Another approach demonstrated in a study by Badisa et al., utilized multi-stage builds, successfully reducing image size by up to 97% [17]. The entire research has observed the approaches separately, without conducting a deeper examination of image processing by combining both approaches. However, Docker itself is known to be superior to other container technologies due to its fast image processing. Therefore, it is important to conduct a more in-depth analysis of image processing, considering both size and time, by combining the two approaches previously employed. Additionally, it should be examined whether combining these two approaches provides greater efficiency compared to other combinations. Furthermore, when combining these two approaches, it is essential to evaluate whether the best results in terms of security are still achieved, as reducing image size may also lower vulnerability risks.

This research aims to optimize container images by combining Alpine Linux as the base image with a multi-stage build technique, and analyzing its efficiency by comparing the images in terms of size, build time, and vulnerabilities across various combinations and environmental conditions. The approach used in this study is experimental, with descriptive and statistical analysis to evaluate the results obtained. It is hoped that this research will provide deeper insights into Docker image optimization, particularly in terms of efficiency. Furthermore, the findings are expected to offer solutions or best practice recommendations for real-world cases and serve as a reference for similar future research, contributing to the development of literature in the fields of containerization and image management, especially in the context of Docker.

II. RESEARCH METHOD

This research adopts a quantitative approach using a quasi-experimental method with a within-subject design. This design was chosen because the study only includes an experimental group, with a non-random sample, followed by a posttest or sequential testing after treatment. In this study, optimization results from combining Alpine Linux with a multi-stage build will be compared to various base image combinations and other build methods. Node and Node-alpine are the base images selected for comparison, with the default Node image version chosen as it serves as the standard runtime for Node.js applications. The Alpine version of Node, as studied by Hakue et al., was selected because it effectively reduces Docker image size and aligns with containerization goals [18]. Thus, the Alpine Node image is appropriate as a comparative group.

In this study, there is one main experimental group that receives the optimization treatment using a combination of the Alpine base image and multi-stage build, along with five comparative experimental groups that receive treatments using other combinations of base images and build methods besides Alpine and multi-stage. Thus, a total of six combinations will produce images that will be compared based on aspects such as size, build speed, and vulnerability. The research flow can be seen in Figure 1 below.

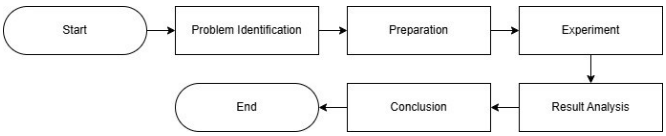

Figure 1. Research Flow

A. Problem Identification

This stage is crucial as it serves as the initial step in identifying the problem and setting the focus of the discussion. The problem formulation or focus of this research is to explore how optimization can be achieved by combining Alpine Linux with a multi-stage build and to analyze the outcomes by comparing other base image and build method combinations. The objective is to present applicable optimizations and identify the most efficient combination in terms of image size, build speed, and vulnerability issues.

B. Preparation

The initial preparation in this study involves determining how optimization will be applied. Since Docker is used as the containerization tool, optimization will be implemented in the Dockerfile for all samples and base images using a multi-stage approach to ensure fair results. The build process will use Docker BuildKit, chosen for its speed and ability to leverage caching, as well as its enhanced build performance through efficient parallelization, caching, and layering [19], [20]. For execution, the `no-cache` command will be applied to prevent caching, and `docker system prune` will be used in CI/CD to clear the cache after each build completes. This approach allows the study to evaluate which treatment group achieves the highest build speed when relying solely on BuildKit

layering. For the optimization implementation, each command will be divided into multiple stages, allowing each layer to leverage artifacts from previous layers or run in parallel to minimize build time. Further details are provided in Figure 2 below.

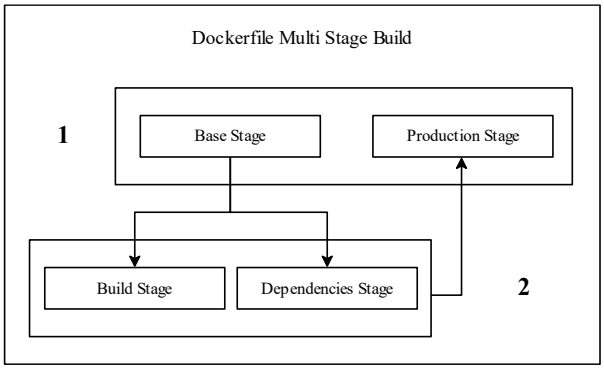

Figure 2. Scheme of Optimization Implementation

Then, three back-end framework samples used Koa JS as the most recommended framework, followed by Express JS as a recommended framework, and Nest JS as a non-recommended framework [21]. For the testing devices in this study, one personal device belonging to the author and three VMs built on different services with varying specifications will be used, as shown in Table 1 below.

Table 1. Specification

Description	Specification
Local Device	Windows OS, 8GB RAM, 500GB Storage, 4CPU
Runner 1 (Azure)	Ubuntu OS, 16GB RAM, 32GB Storage, 4vCPU
Runner 2 (Gitlab Shared Runner)	Saas-linux-small-amd64, 8GB RAM, 30GB Storage, 2vCPU
Runner 3 (AWS)	Ubuntu OS, 1GB RAM, 20GB Storage, 2vCPU

With the differences in specifications and environments, it is anticipated that the data obtained will provide deeper insights, allowing for a more accurate analysis of the results.

C. Experiment

At this stage, implementation is carried out alongside data collection. The data collected corresponds to the aspects of the discussion topic, namely size, build time, and image vulnerability issues. The data collection methods employed include two testing methods and a literature review. In the testing method, this stage will adopt a DevSecOps workflow. Once the optimization implementation is applied, the code will be pushed to the repository, which will automatically trigger the CI/CD process. This CI/CD pipeline will run build tests on each runner/ environment, perform security scanning, and push the resulting image to the container registry. This automation is crucial for avoiding errors, reducing repetitive tasks, and speeding up the data collection process [22]. Build time data will be collected five times at 90-minute intervals. Security data will be gathered from security scanning using Trivy, focusing on high and critical severity vulnerabilities. Trivy was chosen because, in addition to being open-source and supporting CI/CD, Trivy also utilizes the National Vulnerability Database (NVD) published by the Security Division of the National Institute of Standards and Technology (NIST), ensuring that its scanning results comply with the latest standards [23]. Furthermore, Trivy is widely recognized as a standard SBOM (Software Bill of Materials) generator used across various industries, offering well-balanced and detailed vulnerability reporting for container images, operating system packages, application dependencies, and libraries, thereby ensuring thorough security assessments [24], [25]. This feature will be particularly useful, as multi-stage builds are closely tied to application dependencies. Meanwhile, the image size results will be obtained from the container registry after the image push process is completed.

The literature review collection method is also conducted to gather information from journals, books, and other scientific articles that are relevant to the topic being discussed. The information obtained can provide a foundational understanding of the research, including implementable workflows, best practices, and more. This approach ensures that the research proceeds more systematically, yielding results that are more precise and accurate.

D. Result Analysis

After all the data is collected, data analysis will be conducted. The techniques used in data analysis include descriptive analysis and various statistical tests, such as classical assumption testing, difference testing, and advanced testing, which provide a more detailed and in-depth comparison of each combination. Descriptive analysis is employed here to describe the results obtained from the vulnerability issues, where the collected data is not numerical. For numerical data, statistical tests will be performed, including normality and homogeneity tests, as prerequisites for conducting difference testing. The normality test used is the Shapiro-Wilk test, which is suitable for small-scale data [26]. The homogeneity test applied is Levene's test, which is more appropriate for assessing the homogeneity of population variances [27]. The difference test will be conducted to determine whether there is a significant difference among all groups/combinations. Since six combinations are being compared in this study, the appropriate test for detailed calculations and determining the significance of differences across several groups/combinations is One-Way ANOVA, assuming the data meet the assumptions of normality and homogeneity and the Kruskal-Wallis test will be used as an alternative if the normality assumption is not met. In addition to identifying the most efficient combination and examining the significance of differences between all combinations, this test will also serve as the basis for conducting subsequent post-hoc tests. The post-hoc tests will be used to determine the significance of differences when comparing each group/combination one by one. All calculations and analyses of the numerical data were conducted using IBM SPSS

Statistics software to facilitate and enhance the accuracy and efficiency of the data analysis process.

E. Conclusion

In the conclusion phase, an overview of the analysis results and findings will be presented. This will help determine whether the optimization combination of Alpine Linux and multi-stage build yields the best results. This stage will also outline the findings and recommendations based on those findings. It is hoped that readers will be able to assess their needs according to the case studies presented or even explore further based on the findings discussed.

III. RESULTS AND DISCUSSIONS

The optimization implementation, by the previously designed scheme, is carried out in a Dockerfile divided into four stages. In the first stage, the Alpine base image is downloaded, and a working directory is created. Next, NPM is installed to globally install PNPM, using the `no-cache` tag to prevent cache storage. Then, the package.json and pnpm-lock.yml files are copied to the working directory. For further details, please refer to Pseudocode 1.

Pseudocode 1. Stage Base

```
1.# Stage 1: Base
2.FROM alpine: x.xx.x AS base
3.WORKDIR /app
4.RUN apk add --no-cache npm && npm
install -g pnpm
5.COPY package.json pnpm-lock.yaml ./
```

The second stage involves the project build process, using the first stage as a foundation. Development dependencies are installed, the project is copied, and the build is executed. The details of the second stage can be found in Pseudocode 2.

Pseudocode 2. Stage Build

```
6.# Stage 2: Build
7.FROM base AS build
8.WORKDIR /app
9.RUN pnpm install --frozen-lockfile --
prefer-frozen-lockfile
10.COPY . .
11.RUN pnpm run build
```

In the third stage, production dependencies will be installed to ensure that only those dependencies are used in the subsequent stages. Like the second stage, this one is also run in parallel after the first stage is completed. For further details about the third stage, please refer to Pseudocode 3.

Pseudocode 3. Stage Deps

```
12.# Stage 3: Deps Production
13.FROM base AS deps-prod
14.WORKDIR /app
15.RUN pnpm install --prod --frozen-
lockfile --prefer-frozen-lockfile
```

In the final stage, the Alpine image is used to install Node.js, create a working directory, and copy the results from the second and third stages. After that, the project is run with a non-root user. For further details, please refer to Pseudocode 4.

Pseudocode 4. Stage Production

```
16.# Stage 4: Production
17.FROM alpine:x.xx.x AS production
18.RUN apk add --no-cache nodejs && rm
-rf /var/cache/apk/*
19.WORKDIR /app
20.COPY --from=deps-prod
/app/node_modules ./node_modules
21.COPY --from=build /app/dist ./dist
22.EXPOSE 3000
23.RUN addgroup -S appgroup && adduser
-S appuser -G appgroup
24.USER appuser
25.CMD ["node", "dist/main.js"]
```

Then, the Dockerfile configuration is built using Docker BuildKit with the `no-cache` tag and applied to each sample in every runner. When the tests are executed, a CI/CD pipeline workflow will run as shown in Figure 3 below.

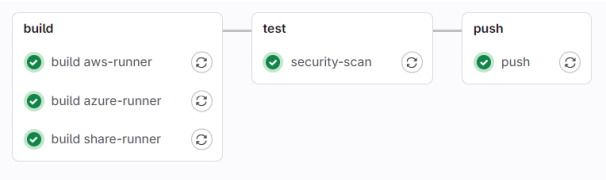

Figure 3. CI/CD Workflow

In the above figure, there are three stages, build, test, and push. In the first stage, three jobs will run to perform the build on the three runners, followed by stage two, which conducts security scanning using Trivy. The final stage is for pushing to the container registry.

A. Analysis of Images Size

In the first analysis, the sizes of the obtained images are compared in terms of size and percentage reduction across all treatments and samples. The percentage reduction in image size is calculated using the largest value as a reference, which corresponds to the base image of the single-stage Node. The results of the percentage reduction can be seen in Table 2 below.

Table 2. Images Size Reduction Percentage

Sample	Combination	Size (MB)	Reduced
Koa	Node Single Stage	401.45	0.00%
	Node Multi Stage	386.08	3.80%
	Node-alpine Single Stage	67.06	83.30%

DOI: 10.34148/teknika.v14i1.1118

Sample	Combination	Size (MB)	Reduced
	Node-alpine Multi Stage	51.69	87.10%
	Alpine Single Stage	42.57	89.40%
	Alpine Multi Stage	**24.10**	**94.00%**
Express	Node Single Stage	405.10	0.00%
	Node Multi Stage	386.07	3.80%
	Node-alpine Single Stage	70.71	82.40%
	Node-alpine Multi Stage	51.67	87.10%
	Alpine Single Stage	46.22	88.50%
	Alpine Multi Stage	**24.09**	**94.00%**
Nest	Node Single Stage	425.29	0.00%
	Node Multi Stage	387.57	3.50%
	Node-alpine Single Stage	90.90	77.40%
	Node-alpine Multi Stage	53.18	86.80%
	Alpine Single Stage	66.41	83.50%
	Alpine Multi Stage	**25.60**	**93.60%**

From the image size results, it is evident that the combination of Alpine Linux and multi-stage builds provides the highest percentage reduction, reaching approximately 94% based on the largest value of each sample. A sufficiently high and satisfactory percentage was observed in terms of size. To conduct a deeper analysis of the significance of the differences among all combinations, a normality test must first be performed as a prerequisite. The results of the normality test are presented in Table 3.

Table 3. Normality of Images Size

Combination	Shapiro-Wilk		
	Statistic	df	Sig
Node Single Stage	.693	15	<.001
Node Multi Stage	.607	15	<.001
Node-alpine Single Stage	.693	15	<.001
Node-alpine Multi Stage	.611	15	<.001
Alpine Single Stage	.693	15	<.001
Alpine Multi Stage	.607	15	<.001

The results above indicate that all combinations have a non-normal data distribution, with a p-value (Sig) of less than 0.001, which is below the 0.05 threshold. Because the normality assumption was not met, the parametric One-Way ANOVA test was not used, and the non-parametric Kruskal-Wallis test was applied instead. The significance value from the Kruskal-Wallis test is shown in Table 4 below.

Table 4. Kruskal-Wallis of Images Size

Test Statistics[a,b]	
	Images Size
Kruskal-Wallis H	64.169
Df	5
Asymp. Sig.	<.001

It is noted that there is a significant difference between the combinations/groups, with a significance value (Asymp. Sig) of 0.001, which is below the 0.05 threshold. This indicates a highly significant result. When each combination was compared individually using the post-hoc test, a pairwise comparison diagram was produced, as shown in Figure 4 below.

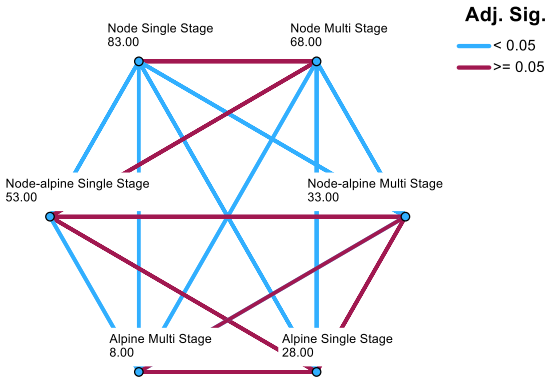

Figure 4. Pairwise Comparison of Images Size

According to the post-hoc test results, the Alpine multi-stage combination ranked first as the combination that produced the smallest image size, with a mean rank score of 8.00. This was followed by Alpine single-stage, Node-Alpine multi-stage, Node-Alpine single-stage, Node multi-stage, and finally Node single-stage. Although comparisons indicate that the Alpine multi-stage combination does not show a significant difference compared to the Alpine single-stage and Node-Alpine multi-stage, as indicated by the red line, this combination still provides the best results. This is because, in practice, companies using container technologies like Docker often run multiple applications within a single VM, which can become problematic if even a simple JavaScript application consumes hundreds of megabytes of storage. When many applications need to be run, companies must allocate substantial storage space, which can be challenging for both small and large companies. Therefore, implementing the Alpine multi-stage combination provides the best solution for saving storage space, as even when compared to the Node version of Alpine with multi-stage, the Alpine multi-stage combination still results in the most efficient image size.

B. Analysis of Building Time

In the analysis of build time, before further analyzing and comparing with post-hoc tests, the normality assumption must be met. After performing the normality test using the Shapiro-Wilk test, the results are presented in Table 5.

Table 5. Normality of Building Time

Runner	Combination	Shapiro-Wilk		
		Statistic	df	Sig
Azure	Node Single Stage	.855	15	.020
	Node Multi Stage	.785	15	.002
	Node-alpine Single Stage	.799	15	.004
	Node-alpine Multi Stage	.812	15	.005
	Alpine Single Stage	.827	15	.008

Runner	Combination	Shapiro-Wilk		
		Statistic	df	Sig
	Alpine Multi Stage	.746	15	<.001
Gitlab Share Runner	Node Single Stage	.857	15	.022
	Node Multi Stage	.775	15	.002
	Node-alpine Single Stage	.789	15	.003
	Node-alpine Multi Stage	.793	15	.003
	Alpine Single Stage	.839	15	.012
	Alpine Multi Stage	.774	15	.002
AWS	Node Single Stage	.921	15	.198
	Node Multi Stage	.814	15	.006
	Node-alpine Single Stage	.836	15	.011
	Node-alpine Multi Stage	.767	15	.001
	Alpine Single Stage	.840	15	.012
	Alpine Multi Stage	.755	15	.001

It was found that almost all data exhibit a non-normal distribution, with results (Sig) less than 0.05 in each testing runner. Therefore, the subsequent analysis was conducted using the non-parametric Kruskal-Wallis test, yielding the following results in Table 6.

Table 6. Kruskal-Wallis of Building Time

Runner	Test Statistics[a,b]	
		Building Time
Azure	Kruskal-Wallis H	63.530
	Df	5
	Asymp. Sig.	<.001
Gitlab Share Runner	Kruskal-Wallis H	64.169
	Df	5
	Asymp. Sig.	<.001
AWS	Kruskal-Wallis H	40.979
	Df	5
	Asymp. Sig.	<.001

From the table above, the build times for all runners show significant differences across all groups/combinations, with an Asymp.Sig value of < 0.001, which is below 0.05. Therefore, when comparing each combination individually using the post-hoc test in the first runner, Azure, with the highest specifications, the pairwise comparison diagram is shown in Figure 5 below.

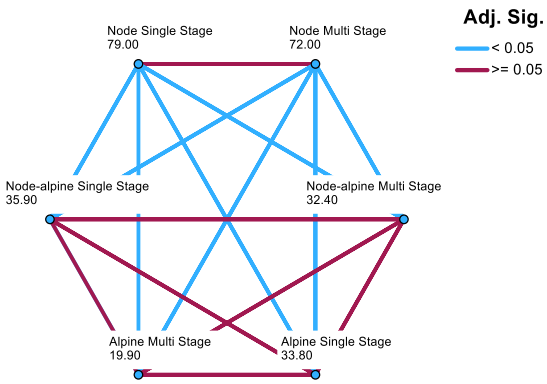

Figure 5. Pairwise Comparison of Azure Building Time

Based on the results above, the Alpine multi-stage combination obtained the lowest mean rank score, representing the shortest duration, with a score of 19.90, followed by Node-Alpine multi-stage, Alpine single-stage, Node-Alpine single-stage, Node multi-stage, and last Node single-stage. Although the Alpine multi-stage combination did not show significant differences compared to three other combinations including Node-Alpine multi-stage, Alpine single-stage, and Node-Alpine single-stage, it still proved to be the most efficient in terms of build time in this runner. For the comparison of combinations in the second runner or environment, the GitLab Shared Runner with medium specifications, the pairwise comparison diagram is shown in Figure 6 below.

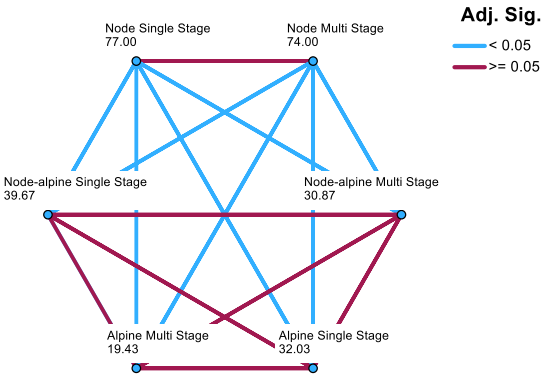

Figure 6. Pairwise Comparison of Gitlab Building Time

In the results from the second runner, which has medium specifications, the Alpine multi-stage combination still holds the top position as the most efficient combination, with the ranking order remaining the same as in the results from the previous runner. Although the significance comparison between combinations remains consistent, the mean rank score slightly decreases but stays around ±19.00. In the last runner, AWS, which has the lowest specifications, a pairwise comparison diagram from the post-hoc test is shown in Figure 7 below.

DOI: 10.34148/teknika.v14i1.1118

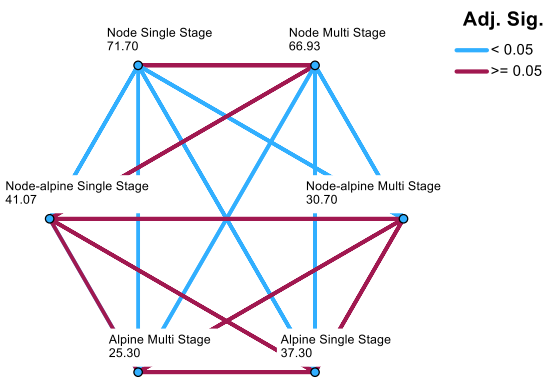

Figure 7. Pairwise Comparison of AWS Building Time

In this runner, based on the results above, the Alpine multi-stage combination consistently remains the most efficient, achieving the lowest mean rank score compared to other combinations. The build duration ranking is also the same as before, with Alpine multi-stage in first place, followed by Node-Alpine multi-stage, Alpine single-stage, Node-Alpine single-stage, Node multi-stage, and lastly, Node single-stage. The significant comparison results are also similar, where the Alpine multi-stage combination across all three runners shows no significant difference compared to Node-Alpine multi-stage, Alpine single-stage, and Node-Alpine single-stage, as indicated by the red line.

However, in this runner, it was noted that the mean rank score for the Alpine multi-stage combination increased to 25.30, approximately 1.3 times higher than in the previous runner. This increase only occurred in the runner with the lowest specifications, indicating that runner specifications have a significant impact and should be carefully considered when choosing the Alpine multi-stage combination. Interestingly, in the low-specification AWS runner, Node multi-stage showed a decrease in the mean rank score, resulting in 8 combinations without significant differences, up from 7 combinations in the other two runners.

Overall, the Alpine multi-stage combination still provides the most efficient build duration compared to other combinations, and this result is consistent across different runners. Therefore, the Alpine multi-stage combination is the best choice for companies or organizations that implement agile development cycles. This speed improvement boosts team productivity in application development by enabling faster feedback through rapid application releases. For commercial companies, this has a positive impact by meeting the need for fast application releases. In organizations applying DevOps or DevSecOps life cycles, this combination brings efficiency to the entire CI/CD process. However, considering runner specifications remains important based on previous findings.

C. Analysis of Vulnerability Issues

In the security scanning report results with Trivy, the findings are divided into two parts, the base image layer and the application layer. The results of the security scanning for the base image layer are as follows in Table 7.

Table 7. Results of the Security Scan on Image Layers

Level	Node	Node-alpine	Alpine
High	93 Vulnerability	-	-
Critical	14 Vulnerability	-	-

Based on the results above, it is evident that high and critical vulnerabilities are present only in the base Node image, unlike the Node-alpine and Alpine images, which do not have high or critical-level vulnerabilities. The absence of vulnerabilities significantly reduces the risk of exploitation and conflicts within the container. Fourteen critical vulnerabilities found in the Node image include packages such as CVE-2024-32002 in git, CVE-2023-6879 in libaoem3, CVE-2024-45490 in libexpat, CVE-2023-5841 in libopenexr, CVE-2024-38428 in wget, and CVE-2023-45853 in zlib. All of these packages pose significant security risks, as these vulnerabilities could allow attackers to execute remote malicious code, corrupt data, or cause system issues. Furthermore, these problems could disrupt the application services running within the container. Therefore, it is crucial to choose an image with minimal vulnerability risks, such as the Node-alpine or Alpine images. The results of the application layer scan are presented in Table 8 below.

Table 8. Results of the Security Scan on Application Layers

Method	Sample	Package	Level	VulnID
Single Stage	Koa	path-to-regexp	HIGH	CVE-2024-45296
	Express	body-parser	HIGH	CVE-2024-45590
		path-to-regexp	HIGH	CVE-2024-45296
	Nest	body-parser	HIGH	CVE-2024-45590
		path-to-regexp 0.1.7	HIGH	CVE-2024-45296
		path-to-regexp 3.1.2	HIGH	CVE-2024-45296
Multi Stage	Koa	path-to-regexp	HIGH	CVE-2024-45296
	Express	body-parser	HIGH	CVE-2024-45590
		path-to-regexp	HIGH	CVE-2024-45296
	Nest	body-parser	HIGH	CVE-2024-45590
		path-to-regexp 0.1.7	HIGH	CVE-2024-45296
		path-to-regexp 3.1.2	HIGH	CVE-2024-45296

Based on the results above, no significant difference was found between single-stage and multi-stage builds. The Koa

sample had 1 high-severity vulnerability, the Express sample had 2 high-severity vulnerabilities, and the Nest sample had 3 high-severity vulnerabilities, with 2 of them present in the same package but in different versions. Although using a multi-stage build is expected to reduce vulnerabilities in development dependencies by isolating them and only including production dependencies in the final image, this benefit was not observed in the samples tested. This is due to the default or simplified configurations and the limited number of dependencies, leading to no notable difference between single-stage and multi-stage builds in terms of vulnerability detection, with only 2 types of vulnerable packages identified. Thus, when applied to a larger scale, the results are likely to differ. However, even with only 2 types of vulnerabilities, the vulnerabilities in the body-parser and regex packages are significant because they can cause Denial of Service (DoS) attacks, posing the risk of slowing down the server or causing it to hang. Therefore, developers must address such vulnerabilities as early as possible to prevent potential future threats.

Using an Alpine base image in combination with a multi-stage build allows developers to focus more on application-layer vulnerabilities, such as the 2 detected package vulnerabilities mentioned earlier. Alpine image has proven to provide better security, as shown in the previous results where no high or critical vulnerabilities were found. Furthermore, as applications grow larger and more complex, combining multi-stage builds can help reduce vulnerabilities in development dependencies at the application level. This approach is essential and can become a best practice, as in real-world scenarios, application performance and security have a direct impact on a company's operations and reputation, especially for large enterprises. The release process for applications and new features can also proceed faster due to the reduced number of unnecessary vulnerabilities, ensuring that the product released is more stable and reliable. For companies adopting modern lifecycles such as DevSecOps, where security is integrated into every stage of the development cycle, this approach supports more responsive and secure development.

D. Comparison of Research Findings

The findings of this research largely align with and support previous studies, with some reinforcing existing conclusions and others providing new perspectives. In a study by C. Tipantuña et al. [15], the use of Alpine was found to successfully save storage resources, which is further supported by findings that Alpine-based images have the smallest size. Additionally, research by Haque et al. [18] found that the Alpine and Node base images are secure with zero vulnerabilities. This finding is reinforced by results indicating that both images do not have any high or critical-level vulnerabilities. Another study by N. Badisa et al. [17] revealed that using multi-stage builds can reduce image size by up to 97%. This finding is supported by the results of this research, which show that multi-stage builds can reduce image size compared to single-stage builds, although the reduction is only in the range of 3% to 10%, depending on the base image used. This study also proves that combining the Alpine Linux base image with multi-stage builds results in the most efficient build duration compared to various other combinations of Node and Node-alpine base images. This is evidenced by build duration tests on the three JavaScript framework samples, where the duration of the combination of Alpine with multi-stage consistently yielded the lowest mean rank compared to other combinations. However, it should be noted that this combination can result in longer build durations on virtual machines with lower specifications.

This consideration is important when implementing it in real-world company scenarios, as companies may not always allocate high-specification virtual machines. On the other hand, this study also found that the Node-alpine image with multi-stage builds had a faster build time compared to the single-stage Alpine image, even though the Alpine base image is smaller than Node-alpine. This result was consistent across all testing environments. These findings reinforce the statement by Changyuan et al. [14] that smaller image size is a best practice, but it must be accompanied by the application of appropriate technologies and instructions, such as the use of multi-stage builds and BuildKit. Erdenebat et al. [20] research demonstrated that BuildKit can achieve efficient results, which is supported by this study through the use of parallel instructions and efficient layering, allowing the Node-alpine multi-stage image to surpass the build speed of the single-stage Alpine image. From this, it can be concluded that utilizing multi-stage builds and BuildKit can accelerate the image build process.

IV. CONCLUSION

The use of Docker image optimization by combining Alpine with multi-stage builds is considered a best practice. This approach results in optimal size efficiency, achieving up to a 94% reduction compared to the Node single-stage setup, and provides the fastest build time when compared to other combinations, including Node or Node-alpine in both single-stage and multi-stage build, as well as the Alpine single-stage configuration. This result is supported by statistical tests, which showed the lowest mean rank scores across all environments, with 19.90 in Azure, 19.43 in the GitLab Shared Runner, and 25.30 in AWS. These scores consistently represent the lowest mean ranks among all tested combinations. Although the differences are not significant, this approach is still highly recommended. Alpine also demonstrates zero high and critical vulnerability issues, unlike Node, which has more vulnerabilities. However, the specifications of the built environment also play a crucial role, as the Alpine multi-stage combination experienced an increase in duration of up to 1.3 times in AWS, even though it remains overall more efficient in build duration.

Based on these findings, the author hopes that future research will explore the combination of Alpine with multi-stage builds in larger-scale deployments within Kubernetes, Nomad, or Docker Swarm environments, and also investigate

the performance aspect of handling client requests by comparing this combination with other approaches.

REFERENCES

[1] P. Muzumdar, A. Bhosale, G. P. Basyal, and G. Kurian, "Navigating the Docker Ecosystem: A Comprehensive Taxonomy and Survey," *Asian Journal of Research in Computer Science*, vol. 17, no. 1, pp. 42–61, 2024, doi: 10.9734/ajrcos/2024/v17i1411.

[2] S. Tarasiuk, D. Traczuk, K. Szczepaniuk, P. Stoń, and J. Smołka, "Performance evaluation of designated containerization and virtualization solutions using a synthetic benchmark," *Journal of Computer Sciences Institute*, vol. 32, pp. 157–162, 2024, doi: 10.35784/jcsi.6231.

[3] O. I. Alqaisi, A. Şaman Tosun, and T. Korkmaz, "Performance Analysis of Container Technologies for Computer Vision Applications on Edge Devices," *IEEE Access*, vol. 12, pp. 41852–41869, 2024, doi: 10.1109/ACCESS.2024.3376570.

[4] I. P. A. Eka Pratama and I. M. Raharja, "Node.js Performance Benchmarking and Analysis at Virtualbox, Docker, and Podman Environment Using Node-Bench Method," *JOIV : International Journal on Informatics Visualization*, vol. 7, p. 2240, Dec. 2023, doi: 10.30630/joiv.7.4.01762.

[5] C. Mukmin, T. Naraloka, and Q. H. Andriyanto, "Analisis Perbandingan Kinerja Layanan Container As A Service (CAAS) Studi Kasus : Docker dan Podman," *Kumpulan jurnaL Ilmu Komputer (KLIK)*, vol. 08, no. 2, pp. 152–161, 2021, doi: http://dx.doi.org/10.20527/klik.v8i2.

[6] Stack Overflow, "Technology | 2024 Stack Overflow Developer Survey." Accessed: Oct. 07, 2024. [Online]. Available: https://survey.stackoverflow.co/2024/technology/

[7] A. R. Ekaputra and A. S. Affandi, "Pemanfaatan layanan cloud computing dan docker container untuk meningkatkan kinerja aplikasi web," *Journal of Information System and Application Development*, vol. 1, no. 2, pp. 138–147, 2023, doi: 10.26905/jisad.v1i2.11084.

[8] A. M. Potdar, N. D G, S. Kengond, and M. M. Mulla, "Performance Evaluation of Docker Container and Virtual Machine," *Procedia Comput Sci*, vol. 171, pp. 1419–1428, 2020, doi: https://doi.org/10.1016/j.procs.2020.04.152.

[9] S. Prayetno and B. Santoso, "Penerapan Docker Container Guna Mempermudah Deployment Dan Maintenance Aplikasi Web (Studi Kasus PT.Gogomedia Visindo)," *Jurnal Sistem Informasi dan Teknologi Informatika*, vol. 1, no. 1, pp. 37–49, 2023.

[10] S. Dwiyatno, E. Rachmat, A. P. Sari, and O. Gustiawan, "Implementasi Virtualisasi Server Berbasis Docker Container," *PROSISKO: Jurnal Pengembangan Riset dan Observasi Sistem Komputer*, vol. 7, no. 2, pp. 165–175, 2020, doi: 10.30656/prosisko.v7i2.2520.

[11] R. Felani, M. N. Al Azam, D. P. Adi, A. Widodo, and A. B. Gumelar, "Optimizing Virtual Resources Management Using Docker on Cloud Applications," *IJCCS (Indonesian Journal of Computing and Cybernetics Systems)*, vol. 14, no. 3, p. 319, 2020, doi: 10.22146/ijccs.57565.

[12] S. Gholami, H. Khazaei, and C.-P. Bezemer, "Should you Upgrade Official Docker Hub Images in Production Environments?," in *2021 IEEE/ACM 43rd International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*, Institute of Electrical and Electronics Engineers, 2021, pp. 101–105. doi: 10.1109/ICSE-NIER52604.2021.00029.

[13] N. Zhao *et al.*, "Large-Scale Analysis of Docker Images and Performance Implications for Container Storage Systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 4, pp. 918–930, 2021, doi: 10.1109/TPDS.2020.3034517.

[14] C. Lin, S. Nadi, and H. Khazaei, "A Large-scale Data Set and an Empirical Study of Docker Images Hosted on Docker Hub," in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Institute of Electrical and Electronics Engineers, 2020, pp. 371–381. doi: 10.1109/ICSME46990.2020.00043.

[15] C. Tipantuña, A. Yazán, and J. Carvajal-Rodriguez, "Containers-Based Network Services Deployment: A Practical Approach," *Enfoque UTE*, vol. 15, no. 1, pp. 36–44, 2024, doi: 10.29019/enfoqueute.1005.

[16] F. B. Fava *et al.*, "Assessing the Performance of Docker in Docker Containers for Microservice-Based Architectures," in *2024 32nd Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, Institute of Electrical and Electronics Engineers, 2024, pp. 137–142. doi: 10.1109/PDP62718.2024.00026.

[17] N. Badisa, J. K. Grandhi, L. Kallam, M. R. Eda, S. Nulaka, and S. Bulla, "Efficient Docker Image Optimization using Multi-Stage Builds and Nginx for Enhanced Application Deployment," 2023. doi: 10.21203/rs.3.rs-3276965/v1.

[18] M. U. Haque and M. A. Babar, "Well Begun is Half Done: An Empirical Study of Exploitability & Impact of Base-Image Vulnerabilities," in *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, Institute of Electrical and Electronics Engineers, 2022, pp. 1066–1077. doi: 10.1109/SANER53432.2022.00124.

[19] Docker, "Docker Docs | BuildKit." Accessed: Jul. 22, 2024. [Online]. Available: https://docs.docker.com/build/buildkit/

[20] B. Erdenebat and T. Kozsik, "Comparative Analysis of Container Build Methods: A Performance Evaluation," in *2024 47th MIPRO ICT and Electronics Convention (MIPRO)*, Institute of Electrical and Electronics Engineers, 2024, pp. 1960–1966. doi: 10.1109/MIPRO60963.2024.10569255.

[21] I. P. A. Eka Pratama, "Pengujian Performansi Lima Back-End JavaScript Framework Menggunakan Metode GET dan POST," *Jurnal RESTI (Rekayasa Sistem dan Teknologi Informasi)*, vol. 4, no. 6, p. 1216, Dec. 2020, doi: 10.29207/resti.v4i6.2675.

[22] R. D. Marcus, A. S. Ilmananda, L. Indana, and H. A. Aswari, "Optimalisasi Manajemen Jaringan pada Laboratorium Komputer Melalui Implementasi Remote Installation Services," *Jurnal MediaTIK*, vol. 6, no. 3, pp. 79–85, 2023, doi: https://doi.org/10.26858/jmtik.v6i3.51964.

[23] P. R. Perkasa and E. Mailoa, "Adopsi Devsecops Untuk Mendukung Metode Agile Menggunakan Trivy Sebagai Security Scanner Docker Image Dan Dockerfile," *Jurnal Indonesia : Manajemen Informatika dan Komunikasi*, vol. 4, no. 3, pp. 856–863, 2023, doi: 10.35870/jimik.v4i3.291.

[24] S. Yu, W. Song, X. Hu, and H. Yin, "On the Correctness of Metadata-Based SBOM Generation: A Differential Analysis Approach," in *2024 54th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, Institute of Electrical and Electronics Engineers Inc., 2024, pp. 29–36. doi: 10.1109/DSN58291.2024.00018.

[25] S. H. Majumder, S. Jajodia, S. Majumdar, and M. S. Hossain, "Layered Security Analysis for Container Images: Expanding Lightweight Pre-Deployment Scanning," in *2023 20th Annual International Conference on Privacy, Security and Trust (PST)*, Institute of Electrical and Electronics Engineers, 2023, pp. 1–10. doi: 10.1109/PST58708.2023.10320152.

[26] G.- Mardiatmoko, "Pentingnya Uji Asumsi Klasik Pada Analisis Regresi Linier Berganda," *BAREKENG: Jurnal Ilmu Matematika dan Terapan*, vol. 14, no. 3, pp. 333–342, 2020, doi: 10.30598/barekengvol14iss3pp333-342.

[27] R. Sianturi, "Uji homogenitas sebagai syarat pengujian analisis," *Jurnal Pendidikan, Sains Sosial, dan Agama*, vol. 8, no. 1, pp. 386–397, 2022, doi: 10.53565/pssa.v8i1.507.