Performance Optimization of CI/CD Pipelines through Redundant Operation Elimination

Saba Gotsiridze

MSc (Information Systems), Business and Technology University, Tbilisi, Georgia.

Email: *saba.gotsiridze.1@btu.edu.ge*

Giorgi Kuchava

PhD (Engineering of Informatics), Associate professor, Georgian Technical University, Tbilisi, Georgia.

Email: *kuchavagiorgi08@gtu.ge*

Teimuraz Sturua

Candidate of Technical Sciences, Associate professor, Georgian Technical University, Tbilisi, Georgia.

Email: *t.sturua@gtu.ge*

Ana Khinchikashvili

BSc (Information Technologies), Business and Technology University, Tbilisi, Georgia.

Email: *ana.khinchikashvili.1@btu.edu.ge*

Ekaterine Somkhishvili

BSc (Information Technologies), Assistant, Business and Technology University, Tbilisi, Georgia.

Email: *ekaterine.somkhishvili.1@btu.edu.ge*

Ioseb Kartvelishvili

Candidate of Technical Sciences, Professor, Georgian Technical University, Tbilisi, Georgia.

Email: *s.kartvelishvili@gtu.ge*

Maka Mantskava

PhD (Biology), Professor, European University, Tbilisi, Georgia.

Email: *maia.mantskava@eu.edu.ge*

Shalva Chokhonelidze

MSc (Computer Science), Senior Scientist, I. Beritashvili Center of Experimental Biomedicine, Tbilisi, Georgia.

Email: *tchokhonelidze.shalva@atsu.edu.ge*

Nana Momtselidze

PhD (Biology), Senior Scientist, I. Beritashvili Center of Experimental Biomedicine, Tbilisi, Georgia.

Email: *nana.momtselidze@eu.edu.ge*

Abstract

The proposed approach provides actionable insights for DevOps practitioners and supports data-driven decision making in CI/CD pipeline design. This article presents an experimental analysis of Continuous Integration and Continuous Delivery/Deployment (CI/CD) pipelines with a focus on performance optimization through the identification and elimination of redundant operations. Based on controlled experiments conducted in virtual machine environments and real-world CI/CD pipelines, the study demonstrates that unnecessary duplication of build and restore steps significantly increases execution time and resource consumption. A mixed-methods research approach is employed, combining quantitative performance measurements with a practical case study implemented in Azure DevOps. The results indicate a measurable reduction in average build and integration time after applying targeted optimization strategies, including framework-specific configurations, caching mechanisms, and pipeline restructuring. The findings confirm that systematic CI/CD optimization improves software delivery efficiency and supports faster, more reliable release cycles. The study contributes practical guidelines for DevOps teams seeking to enhance pipeline performance while maintaining code quality and system stability [1,2].

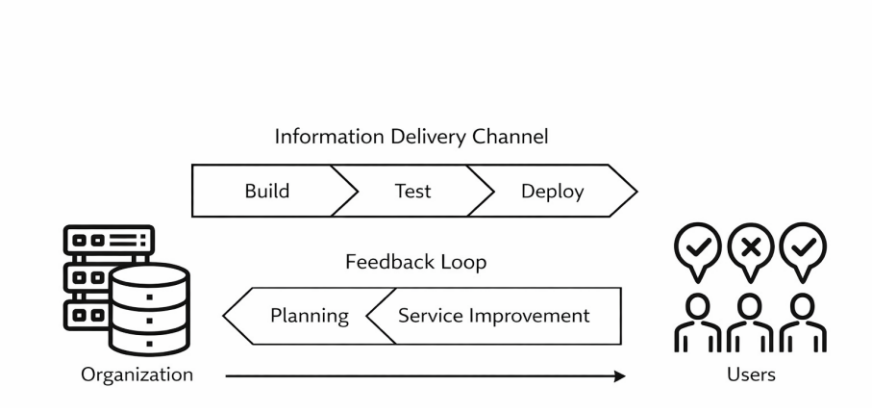
Keywords: CI/CD, DevOps Optimization, Pipeline Performance, Continuous Integration, Build Time Reduction

1. Introduction

Before the development of DevOps systems and the widespread availability of personal computers, software throughout its lifecycle was typically developed and maintained by a single team. Subsequent technological advancements contributed to the specialization of software development, testing, and information technology roles, leading to the establishment of structurally distinct positions such as developer, tester, system administrator, network administrator, and project manager. This structural separation introduced a handoff-based workflow, involving the transfer of information and products from one team to another. However, this approach exhibits limited efficiency, as it is practically impossible to fully transfer

the knowledge accumulated by one team to another during the handoff process. To illustrate this limitation in efficiency, the following process is considered: Software engineers write code, while IT operations teams manage servers. Developers place the written code into version control repositories, such as GitHub or Azure DevOps, which are managed through Git-based version control systems. Newly introduced code may cause failures in the software system deployed on servers, leading to malfunctioning client-side applications, even though the same code operates correctly on the developer's local machine. As a result of such inefficiencies, both development and operations teams faced various challenges, primarily related to insufficient information sharing and inadequate testing environments. Teams also occasionally made conflicting decisions regarding the trade-off between software delivery flexibility and operational stability. Consequently, the need to overcome inefficient delivery across defined phases of software development became a critical objective. To address these challenges, the DevOps concept was introduced. DevOps is based on establishing a continuous connection between software service producers and end users through a shared delivery pipeline, incorporating structured feedback loops that relay user feedback back to service producers. This approach has evolved into a widely adopted software development strategy that enables organizations to achieve continuous delivery while simultaneously ensuring stable and reliable software evolution. At the core of DevOps lies CI/CD (Continuous Integration and Continuous Delivery/Deployment) , which operates through an information delivery channel known as the pipeline. The logic of Continuous Integration involves the frequent merging of code changes submitted by developers into a shared repository—effectively a common codebase—where updates are continuously integrated with existing functionality. Following the merge process, the system automatically builds and tests the application. Continuous Delivery/Deployment, in turn, refers to the practice whereby code is promoted to a production environment after a manually approved step, according to the delivery logic. Together, CI/CD form a delivery system (Figure 1) that automates the following processes: code build, testing, and deployment into an operational environment. [1,3]

Figure 1. DevOps Functional Architecture



Despite their widespread adoption, CI/CD pipelines often suffer from performance inefficiencies caused by redundant operations, suboptimal configurations, and insufficient resource utilization. These inefficiencies increase build times, consume excessive computational resources, and negatively impact developer productivity. As organizations scale their DevOps practices, the need for systematic performance optimization of CI/CD pipelines becomes increasingly important. The objective of this research is to analyze the performance characteristics of CI/CD pipelines, identify sources of inefficiency, and evaluate optimization strategies aimed at reducing execution time without compromising software quality. The study focuses on redundant build and restore operations observed in real-world pipelines and proposes practical optimization techniques validated through controlled experiments and a case study conducted using Azure DevOps [3,4].

The evolution of DevOps has been closely associated with Agile and Lean methodologies, which emphasize iterative development, continuous feedback, and waste reduction. Previous studies have highlighted the importance of CI/CD pipelines as a core component of DevOps, enabling frequent integration and automated testing to improve software reliability and delivery speed. Research on CI/CD performance has primarily focused on process automation, tool integration, and organizational factors influencing DevOps maturity. Several authors have demonstrated that automated testing and continuous integration significantly reduce defect rates and improve deployment frequency. However, fewer studies address the internal efficiency of pipeline execution, particularly the impact of redundant operations on build performance. Recent

empirical studies suggest that pipeline optimization through caching mechanisms, parallel execution, and framework-specific configurations can lead to substantial performance gains. Nevertheless, existing literature often lacks detailed experimental validation in controlled environments. This study extends prior work by providing quantitative evidence of performance improvements achieved through the systematic elimination of redundant CI/CD operations, supported by real-world pipeline data [4,5].

2. Methodology

Optimization of CI/CD Process Efficiency: As previously mentioned, Continuous Integration and Continuous Delivery/Deployment (CI/CD) processes play a critical role in modern software development. They automate application build, testing, and deployment activities, thereby accelerating the software delivery lifecycle (SDLC). It is important to note that increases in build, compilation, and testing time within these processes can lead to bottlenecks, slow down development velocity, and reduce responsiveness to change. As observed in the literature review, performance evaluation metrics help identify areas for improvement. One of the key metrics is lead time, which encompasses the time spent on code build and testing activities. The conducted study reveals the impact of optimizing repetitive steps in CI/CD processes on reducing build and testing time. A more detailed examination, specifically through the analysis of Azure DevOps pipeline logs, revealed that certain processes contain redundant steps, particularly unnecessary repeated invocations of specific commands. Although individual steps within the pipeline may appear negligible in isolation, their cumulative impact can significantly affect overall efficiency. These repetitive steps potentially contributed to extended build times, thereby hindering the overall effectiveness of the process. Accordingly, this study focuses on analyzing the impact of optimizing repetitive steps in CI/CD processes. [6,7]The central research question examines whether simplifying and optimizing duplicated operations can lead to a significant reduction in build time. This objective aligns with the broader goal of improving CI/CD pipeline efficiency, as the elimination of unnecessary steps can result in faster continuous integration processes. With this understanding of the problem and the potential

benefits of optimization, the research objective is defined as analyzing the impact of optimizing repetitive steps in a continuous integration pipeline, using Azure DevOps as a case study, in order to reduce overall build time. The study aims to analyze the effectiveness of eliminating redundant steps in optimizing continuous integration execution time. By combining quantitative analysis with a case study approach, this research examines the potential for significant reductions in build time and provides valuable insights for optimizing CI/CD processes in similar scenarios. By achieving this objective, the study aims to demonstrate a clear relationship between the elimination of duplicated operations and improved build time efficiency in real-world CI/CD process scenarios. The findings provide valuable insights into potential optimization strategies and their effectiveness for similar processes across different frameworks [8,9].

Research Methodology and Its Relevance: This study adopts a mixed-methods approach that combines quantitative analysis with a specific case study in order to comprehensively examine the impact of optimizing repetitive steps on CI/CD execution time. This approach offers several advantages: Quantitative Analysis, Measuring the impact on build time: By running simulations on virtual machines, quantitative analysis enables the execution of controlled experiments to isolate the impact of the proposed changes—specifically the reduction of redundant restore and build operations—on build time. This approach allows for the measurement of actual time differences before and after optimization, thereby providing a clear metric for evaluating efficiency [10,11].

Data-driven decision making: Quantitative analysis enables the application of data analysis techniques to derive statistically meaningful conclusions regarding the effectiveness of the proposed optimization strategy. This strengthens the research findings by providing objective evidence of build time reduction.

Real-world validation: While simulations conducted in a virtual machine environment provide valuable insights, the case study component adds significant real-world applicability. By applying the optimization strategy to real CI/CD pipelines within the Azure DevOps all-in-one

platform, it becomes possible to observe its impact on actual continuous integration execution time in a production environment. This enhances the generalizability of the findings by demonstrating effectiveness in a practical, real-world scenario.

Addressing potential variations: Real-world pipelines may involve additional complexities compared to virtual machine environments. The case study approach allows for the identification of any unforeseen factors that may influence build time beyond the targeted optimizations. This provides a more comprehensive understanding of how optimization strategies may interact with existing continuous integration configurations.

Integration of research methods, The combination of quantitative research and a case study approach provides a more comprehensive perspective: Virtual machine experiments provide a controlled environment to isolate the impact of optimization on build time, The case study validates the effectiveness of these findings in a real-world context, while accounting for potential complexities, Together, these methods provide a robust understanding of the potential for reducing build time through the optimization of repetitive steps.

Discussion of Alternative Approaches: Although alternative methodologies could have been employed, a purely quantitative approach (e.g., analyzing only existing pipeline logs) may lack the level of control required to isolate the impact of the proposed optimization strategy. Conversely, a purely qualitative approach (e.g., interviews with software developers and DevOps engineers) may not provide the same level of measurable data regarding build time reduction. The mixed-methods approach establishes a balanced framework by enabling both controlled measurement and real-world application. This combination provides a more comprehensive and reliable understanding of the research objective [11].

3. Results and Discussion

Data collection was performed through simulations in a virtual machine environment.

To measure the impact of optimizing redundant steps, experiments were conducted in a virtual machine environment. The experiment involved the creation of two scripts designed to simulate the company's continuous integration process. For automation within the Azure DevOps environment, a Microsoft .NET project was selected.

This choice was motivated by several factors:

Seamless integration with Azure DevOps: Since the Azure DevOps platform had already been analyzed, including its strengths and limitations, it was considered an appropriate choice for the case study. As an all-in-one platform, Azure DevOps simplifies the implementation of the experimental setup by providing an integrated and user-friendly environment with extensive built-in functionality.

Simplified automation for .NET development: Azure DevOps provides extensive support for .NET-related tasks, including building, testing, and deploying .NET applications. This capability simplifies automation workflows for .NET projects within the Azure ecosystem and enables efficient configuration of CI/CD pipelines.

Microsoft support and resources: Selecting a tool developed by Microsoft ensures access to comprehensive and well-documented official resources, including detailed technical documentation and long-term platform support.

Rich CI/CD functionality: Azure DevOps delivers a feature-rich CI/CD solution that supports comprehensive pipeline configuration, execution, and monitoring.

Script 1 (simulation of the existing version): This script represents the commands typically used in a .NET CI pipeline, including dotnet restore, dotnet build, and dotnet test.

```
set log=f:\log_before.txt
cd ../project_path
echo -------  >> %log%
echo %time% >> %log%
dotnet restore project_name
```

```
echo %time% >> %log%
dotnet build project_name
echo %time% >> %log%
dotnet test project_name
echo %time% >> %log%
```

At first glance, this script appears straightforward; however, an examination of Azure DevOps pipeline logs revealed that certain commands are executed multiple times in a redundant manner.

Script 2 (optimized version): This script incorporates the proposed optimizations. To eliminate redundant operations—specifically repeated restore and build processes—the --no-restore and --no-build parameters are applied in the .NET commands.

```
set log=f:\log_after.txt
cd ../project_path
echo -------  >> %log%
echo %time% >> %log%
dotnet restore project_name
echo %time% >> %log%
dotnet build project_name --no-restore
echo %time% >> %log%
dotnet test project_name --no-build
echo %time% >> %log%
```

Although the implementation of optimization strategies varies across different frameworks, the underlying principle remains the same. To illustrate this approach, Java and Python programming environments are considered as representative examples.

A standard Maven build script typically includes the stages of cleaning, building, testing, and packaging. While comprehensive, this process can be time-consuming when repetitive tasks are executed unnecessarily. As an optimization mechanism, Maven profiles can be utilized to define alternative build configurations. For this purpose, a profile named fast-build is created in the pom.xml file (figure 2).

Figure 2. Maven Profile for Build Process Optimization

```
<profile>
  <id>fast-build</id>
  <build>
    <skipTests>true</skipTests> <plugins>
      <plugin>
        <artifactId>maven-compiler-plugin</artifactId>
        <configuration>
          <skip>true</skip> </configuration>
      </plugin>
    </plugins>
  </build>
</profile>
```

When executing the command mvn clean install -P fast-build, the -P parameter specifies the optimized profile, which skips test execution and may potentially bypass the compilation stage if the required artifacts are already available.

In Python, when using pip, the standard build process may include dependency installation, test execution, and the creation of a deployable package. Unlike the .NET and Java examples, built-in optimization mechanisms are not available in the same form; however, optimization can be achieved through the use of virtual environments and dependency caching:

Create a Python virtual environment: python -m venv venv.

Activate the virtual environment: venv/bin/activate on Linux/macOS systems, or venv\Scripts\activate on Windows systems

Install dependencies using caching: pip install --cache-dir ~/cache/pip -r requirements.txt, using a dedicated cache directory to store downloaded packages.

Subsequent build processes can leverage cached dependencies, thereby reducing installation time.

It should be noted that these examples are illustrative. Specific optimization techniques and tools vary depending on the framework and project requirements. Therefore, consulting the official documentation of the selected framework is essential to identify and apply best practices for efficient build and integration processes.

Both scripts developed in this study were executed on a virtual machine configured with specifications comparable to the company's CI environment. To ensure consistency and minimize the influence of external factors, each script was executed five times. During each execution, the

start and end times of each build stage (restore, build, and test) were recorded using a timestamping mechanism, specifically the echo %time% command.

Data Analysis: Build Time Improvement

The collected data were analyzed to assess the impact of optimization. To calculate the average build time, the data obtained from each script (existing and optimized) were transferred to an Excel file for further processing and comparison.

Comparison of total process execution time: Based on the data collected from each script, diagrams were constructed and compared. These diagrams illustrate the execution time of each pipeline stage before and after the applied optimizations.

Build time improvement: The percentage improvement in build time achieved as a result of script optimization is calculated using the following formula:

Build time improvement percentage = (Total execution time of the existing script – Total execution time of the optimized script) / Total execution time of the existing script * 100

This formula calculates the difference between the total build execution times of the existing and optimized scripts. The resulting percentage represents the improvement in build time achieved through script optimization.

In the presented diagram (Figure 5), a comparison of the total process execution times before and after optimization is shown. As can be observed visually, the execution time of the original script is consistently higher than that of the optimized script at each stage of the pipeline. The change is particularly evident in the testing time comparison diagram (Figure 5), which demonstrates a significant difference in the execution time of the dotnet test stage between the two scripts.

Following script optimization, the total process execution time improved by 26.78%, as shown in Table 1. Based on this observation, it can be inferred that eliminating redundant processes can significantly enhance the efficiency of the continuous integration workflow. In this particular case, the strategic use of the --no-restore and --no-build parameters enabled a streamlined build process.

Figure 3. Number of Runs and Continuous Integration Simulation Time.

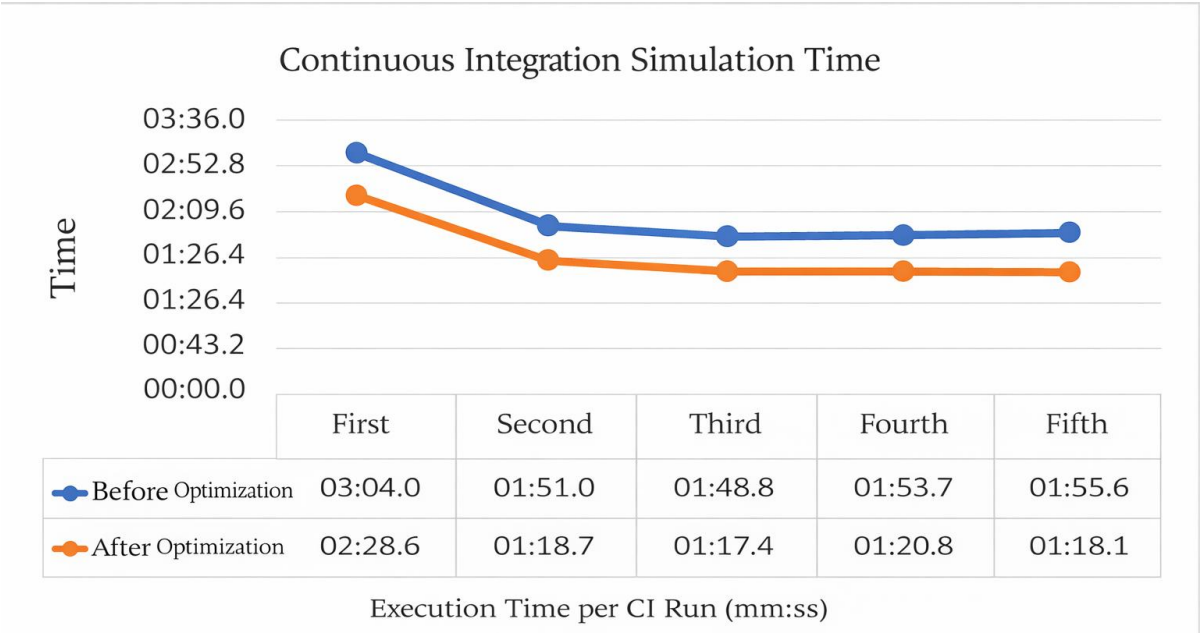


Figure 4. Number of Runs and Restore Time.

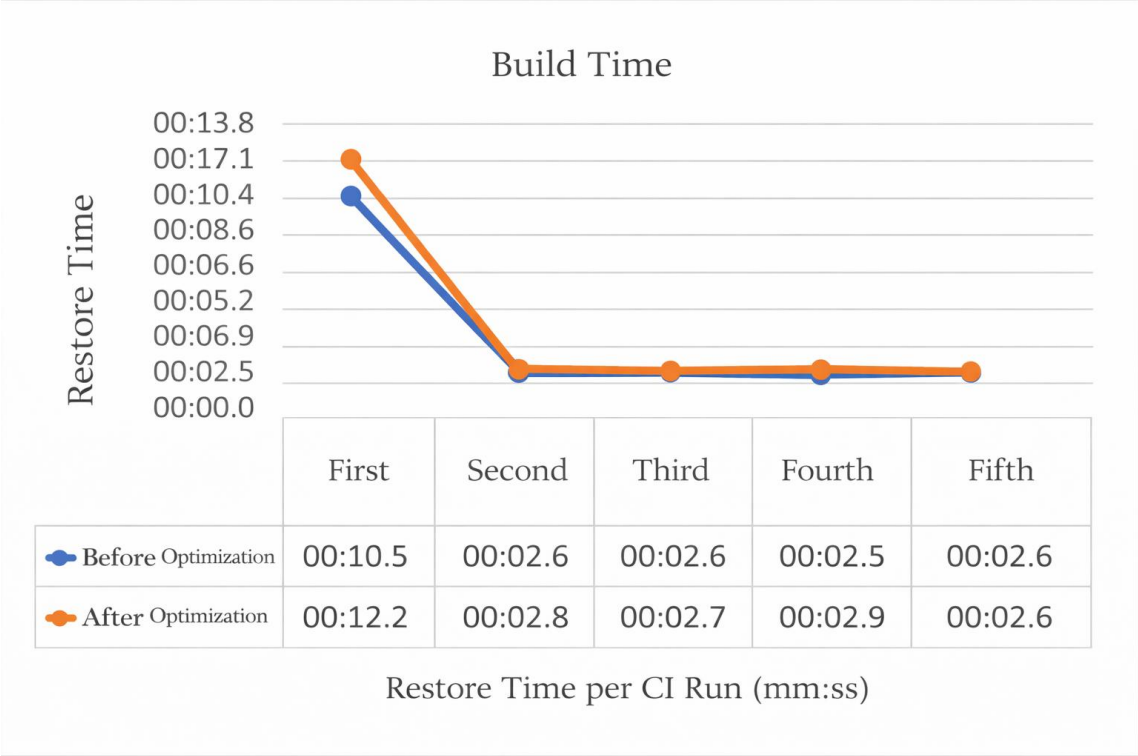


Figure 5. Number of Runs and Restore Time.

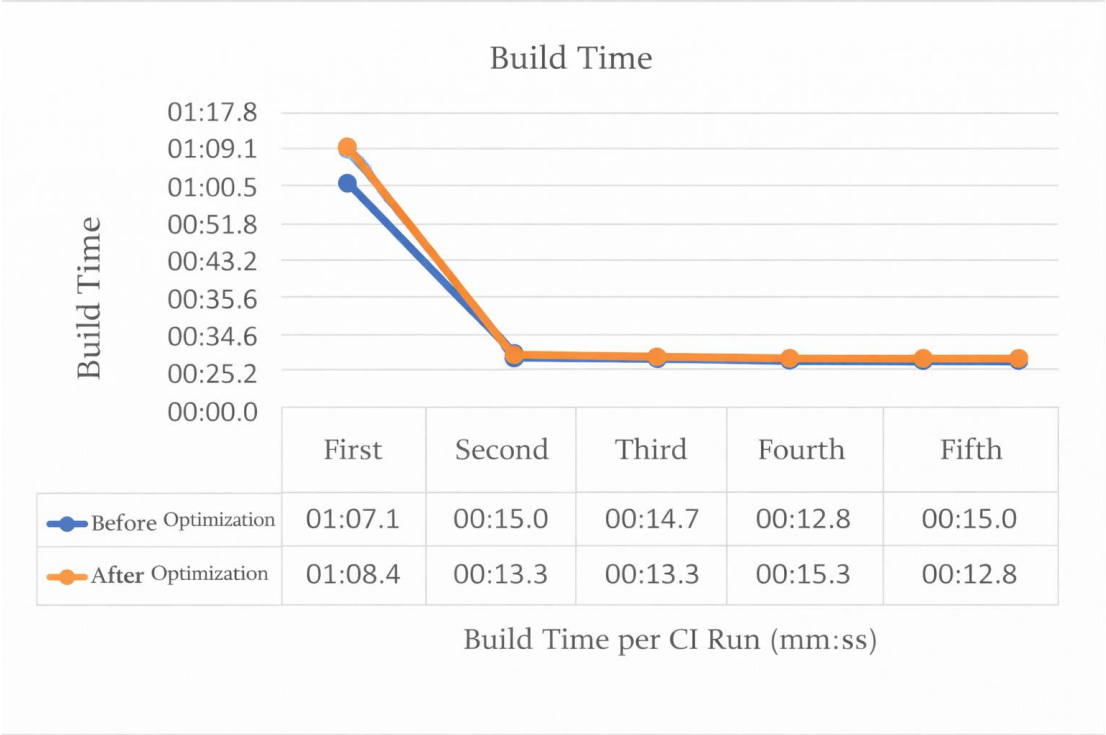
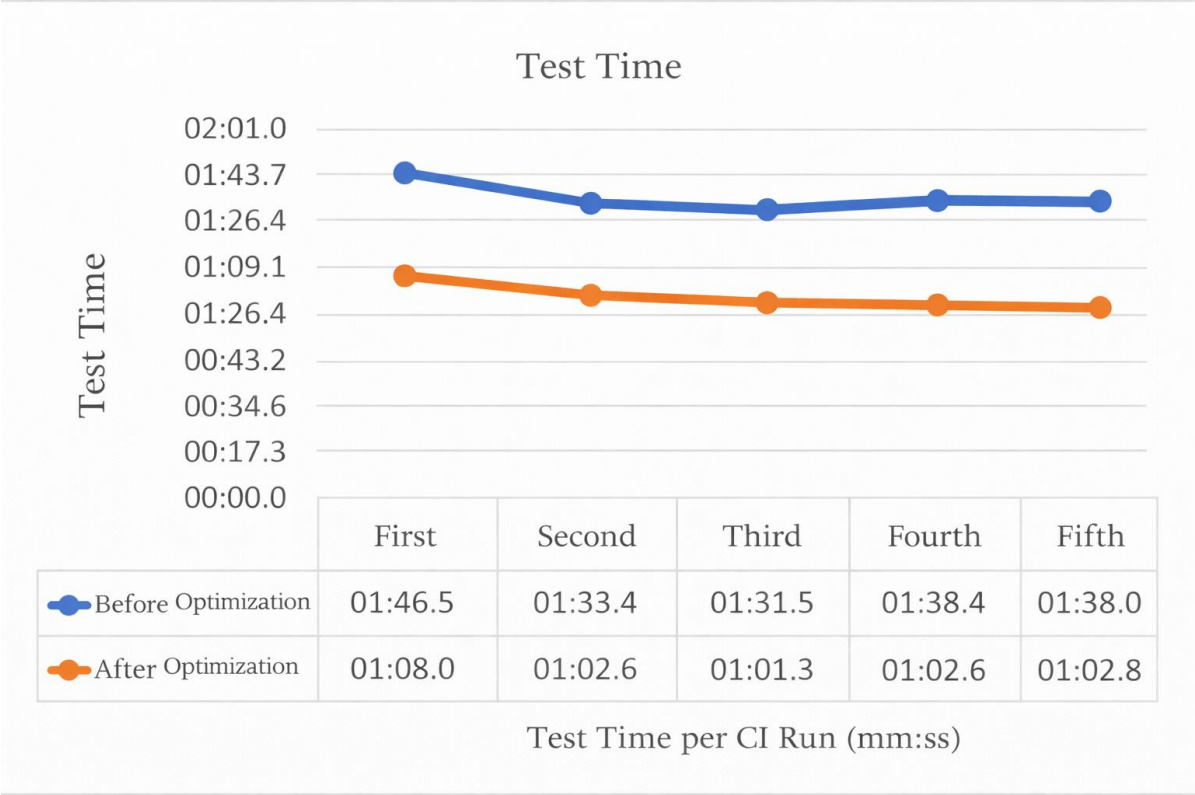


Figure 6. Number of Runs and Test Time.



Case Study Results

This section analyzes the impact of implementing the proposed optimization on the company's real continuous integration pipeline deployed in Azure DevOps.

Reduction in Build Time:

Following the analysis of simulations conducted in a virtual machine environment and the evaluation of the obtained results, the proposed optimization strategy was implemented in the CI pipeline after consultation and agreement with the company. By strategically incorporating the --no-restore and --no-build parameters, the objective was to eliminate redundant restore and build processes in order to achieve a faster pipeline in a real production environment.

According to the workflow presented in the literature review, the development process follows a structured sequence. The process begins with a planning phase, during which team members meet to discuss and define the tasks to be completed. Following this discussion, a two-week sprint is initiated. It should be noted that the company employs a hybrid model, using GitHub for version control and Azure DevOps for managing continuous integration and deployment processes.[1]

Table 1. Comparison of Execution Times of Existing and Optimized Scripts.

Run	Unoptimized Script Execution Time	Optimized Script Execution Time
Run 1	03:04.0	02:28.6
Run 2	01:51.0	01:18.7
Run 3	01:48.8	01:17.4
Run 4	01:53.7	01:20.8
Run 5	01:55.6	01:18.1
Average	02:06.6	01:32.7
	02:06.6	01:32.7
Percentage of Execution Time		26.77807085

Once a developer completes a task, the source code undergoes isolated build/compilation and testing processes. According to records obtained from Azure analytics, the average execution time of the continuous integration process over the past 30 days is approximately 10 minutes (Figure 7). Given the size of the project comprising more than 100 libraries and applications, Based on statistical data, the CI process is expected to take an average of 10 to 20 minutes. It is observed that 42.9% of this time is consumed by the build stage, while the remaining time is allocated to source code retrieval, integration, unit testing, and dependency caching (Figure 8).

Figure 7. Average Pipeline Execution Time on Azure DevOps.

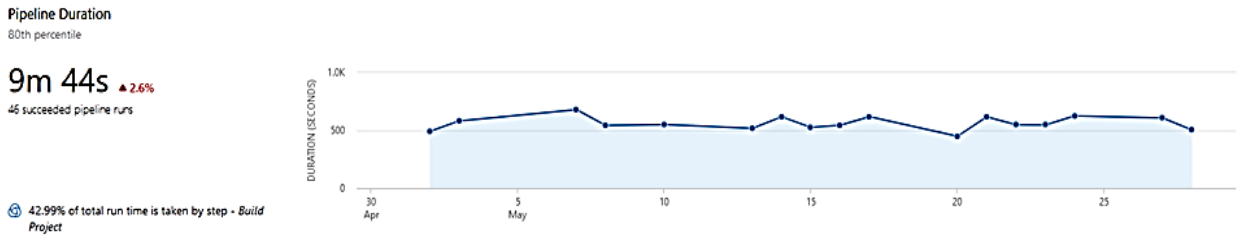
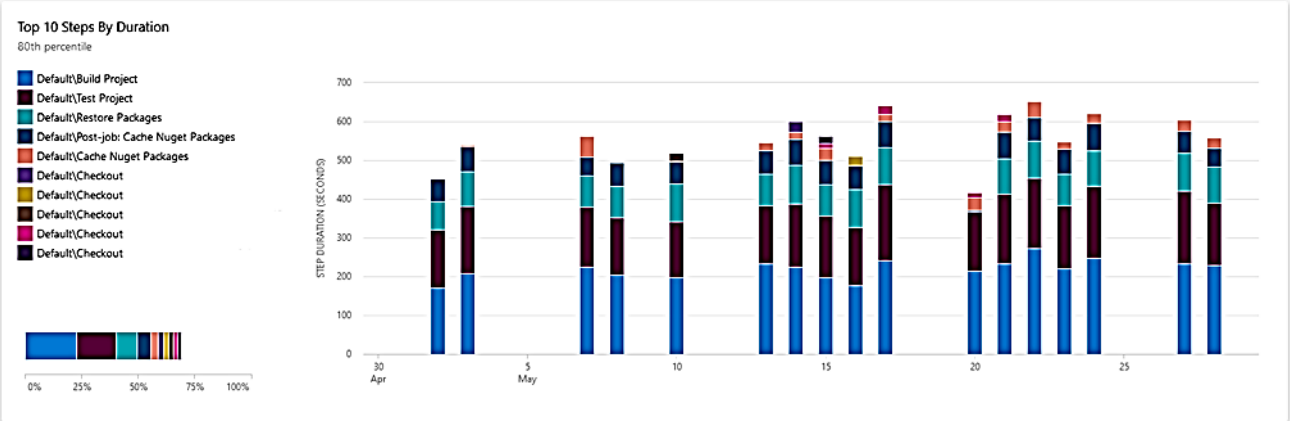
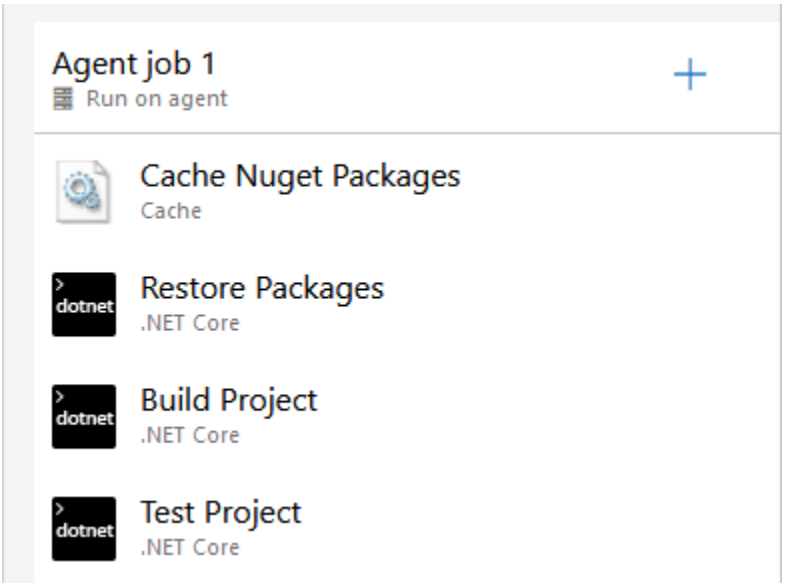


Figure 8. Pipeline execution times by step in Azure DevOps.



Based on the findings of our study, we decided to propose CI pipeline optimization to the company, with the specific goal of reducing CI process execution time. To analyze the process in greater detail, the complete workflow of the existing pipeline is considered, including source code retrieval, dependency caching, dependency restoration, project build, and test execution (Figure 9).

Figure 9. Continuous Integration Stages in Azure DevOps.



The changes were implemented in agreement with the company through a series of meetings and consultations. Following the introduction of these modifications, the first several pipeline runs resulted in a time savings of approximately 2–3 minutes (Figure 10). To fully assess the broader impact of the applied changes, it is necessary to observe the pipeline performance over the course of the following days.

Figure 10. Execution Time of Processes After Optimization

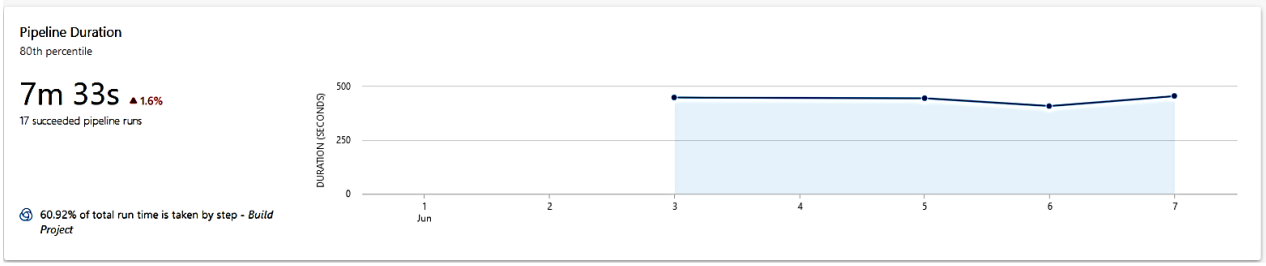
windows-2022	May 30 at 6:50 PM	<1s	6m 37s
windows-2022	May 30 at 6:39 PM	<1s	5m 37s
windows-2022	May 30 at 6:30 PM	<1s	5m 31s
windows-2022	May 30 at 6:22 PM	<1s	5m 57s

After a seven-day observation period, Azure analytics indicate that the average execution time decreased from 10 minutes to 7.33 minutes. This reduction suggests that further improvements may be achievable over time. To evaluate the effectiveness of the implemented changes on the CI process, the formula developed in this study is applied as follows:

$$\text{Improvement Percentage} = \frac{[584-453]}{584} * 100 = 22.6\%$$

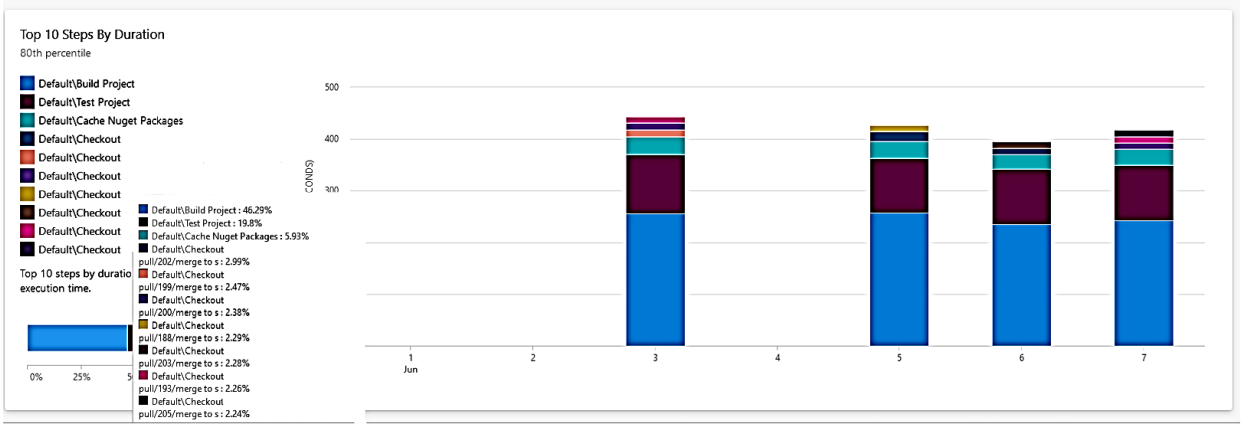
The findings of the case study demonstrate that the applied optimization strategy successfully reduced build time (Figure 11). Following the implementation of the changes, the average execution time of the CI process over the last seven days is presented in Figure 11.

Figure 11. Average CI Process Execution Time Over the Last 7 Days.



It is evident that the majority of the execution time is allocated to the build stage (Figure 12). Redundant restore processes were eliminated through the use of the --no-restore and --no-build parameters, as demonstrated in the optimized script example. [1,2]

Figure 12. Time Distribution of Executed Stages in the CI Process.



The experimental evaluation demonstrates a clear improvement in CI/CD pipeline performance following the implementation of optimization strategies. Comparative analysis of execution metrics indicates a consistent reduction in average build and integration times across

multiple test scenarios. Results obtained from virtual machine experiments show that eliminating redundant restore operations significantly reduces pipeline execution duration. In Azure DevOps pipelines, optimized configurations resulted in shorter build stages and more predictable execution behavior. These findings confirm that systematic pipeline optimization enhances overall CI/CD efficiency while maintaining functional correctness and test coverage.

The findings of this study highlight the critical impact of internal pipeline efficiency on DevOps performance. Redundant operations, often overlooked during pipeline design, introduce unnecessary delays and resource consumption. By addressing these inefficiencies, organizations can achieve measurable performance gains without additional infrastructure investment. The results further indicate that optimization strategies must be context-aware and framework-specific. Generic optimization approaches may yield limited benefits, whereas tailored configurations aligned with the underlying build frameworks and tools produce significant improvements. These observations reinforce the importance of continuous monitoring and iterative optimization in CI/CD environments.[3,4]

4. Conclusion

This study presents an empirical investigation into CI/CD pipeline performance optimization through the elimination of redundant operations. The results demonstrate that targeted optimization strategies can substantially reduce execution time and improve software delivery efficiency. The proposed approach offers practical guidance for DevOps teams seeking to enhance pipeline performance while preserving software quality and system reliability.[2] Future research may extend this work by exploring adaptive optimization techniques, chaos engineering experiments, and machine learning-based approaches for dynamic pipeline optimization.

REFERENCES

1. G. Kuchava, S. Gotsiridze, M. Mantskava, and N. Momtselidze, "Analyzing the performance of DevOps integral components continuous integration/continuous deployment (CI/CD) modules and investigating the functionality and possibilities for software improvement," in *Research Collaboration, Consolidation, Friendship*, pp. 53–54, 2024.
2. S. Gotsiridze, "Analysis of the efficiency of CI/CD modules as integral components of DevOps and opportunities for software improvement," M.S. thesis, Business and Technology University, 2024 (in Georgian).
3. N. Forsgren, J. Humble, P. Debois, J. Willis, and G. Kim, *The DevOps Handbook: How to Create World-Class Agility, Reliability, and Security in Technology Organizations*. IT Revolution Press, 2021.
4. Z. Kavtaradze, "Increasing software quality and deployment speed through CI/CD in DevOps," M.S. thesis, Business and Technology University, 2024 (in Georgian).
5. N. Demchenko, "Optimizing DevOps performance using chaos engineering," M.S. thesis, Business and Technology University, 2025 (in Georgian).
6. G. Kuchava, G. Dvalishvili, and I. Kartvelishvili, "Optimizing the quality and speed of software delivery in DevOps with CI/CD," Georgian Technical University, pp. 72–78, 2024 (in Georgian).
7. G. Kuchava, I. Kartvelishvili, and S. Vashalomidze, "Using chaos engineering to enhance the resilience of microservice architecture: An analysis of the 'Online Boutique' case," Georgian Technical University, pp. 469–473, 2025 (in Georgian).
8. L. Lwakatare, "DevOps in practice: A multiple case study of five companies," *Information and Software Technology*, 2019.
9. N. Forsgren and E. Freeman, *DevOps for Dummies*. For Dummies, 2019.
10. J. Hyman, *Microsoft Azure for Dummies*. For Dummies, 2023.
11. N. Forsgren, J. Humble, and G. Kim, *Accelerate: The Science of Lean Software and DevOps – Building and Scaling High Performing Technology Organizations*. IT Revolution, 2024.