UDC: 004.9 https://doi.org/10.15407/jai2025.02.063

V. Falkevych[1], A. Lisniak[2]
[1,2]Zaporizhzhya National University, Ukraine
66, University st., Zaporizhzhia, 69600
[1]vitaliifalkevich@gmail.com
[2]kmm@znu.edu.ua
[1]https://orcid.org/0000-0002-1114-7206
[2]https://orcid.org/0000-0001-9669-7858

OPTIMIZATION OF INFRASTRUCTURE DEPLOYMENT FOR MULTI-FRONTENDS IN MONOREPO

Abstract. Context: this study examines the optimization of CI/CD pipelines for the subsequent deployment of containerized applications in order to enhance the efficiency of multi-frontend projects within monorepo environments.

Objective: the primary objective is to develop and evaluate the effectiveness of the proposed infrastructure deployment approach by implementing specialized pipeline configurations that support parallel task execution and caching mechanisms.

Method: an empirical research methodology was adopted to analyze the impact of CI/CD pipeline optimization techniques on build performance and resource utilization. The study includes a comparative experiment between sequential and parallelized pipeline executions, demonstrating a measurable reduction in total build timewhen parallel processing and caching are employed effectively. These results highlight the practical benefits of task decomposition and concurrency in complex, component-driven architectures. Additionally, the study explores automation strategies for managing the CI/CD lifecycle, including artifact storage, cleanup policies, and deployment orchestration.

Results: this research contributes to the field of software engineering by providing a validated methodology for CI/CD optimization in large-scale, monorepo-based multi-frontend systems. The findings offer actionable insights for developers and DevOps practitioners seeking to modernize their deployment processes and can be extended to broader software delivery pipelines to improve maintainability and operational efficiency.

Conclusions: the results obtained underscore the importance of aligning architectural decisions with CI/CD design. In particular, the use of parallelization and modular builds not only enhances performance but also promotes better separation of concerns and system modularity. These findings encourage further exploration of optimization strategies that integrate architectural and infrastructural improvements in tandem.

Keywords: CI/CD, Pipelines, Frontend, Monorepo, Development Infrastructure, GIT Flow.

Introduction

The design and optimization of infrastructure for modern web systems constitute a critical research area in software engineering, particularly in the context of scalable, multi-frontend architectures.

Although numerous ready-made solutions exist, bespoke software projects require tailored and optimized approaches to infrastructure deployment. This research examines the architectural and methodological foundations for developing infrastructure in multi-frontend environments, with a focus on monorepo-based development methodologies. In this context, a multi-frontend project is defined as a collection of analogous frontend applications that share or customize common packages to achieve brand-specific adaptations. The primary advantage of such an approach is its adaptability, allowing both functional and stylistic customization while ensuring code reusability and modularity. Shared features can be leveraged across multiple brands while maintaining the flexibility to adjust styling and branding independently. This modularity ensures that distinct features can be activated or deactivated within each project as required, enhancing scalability and maintainability.

Infrastructure design for multi-frontend projects encompasses the entire lifecycle of CI/CD processes, from build orchestration to deployment and monitoring. A key challenge in this domain is the efficient configuration of multi-frontend build pipelines capable of assembling shared libraries, constructing brand-specific builds, and generating Docker images for deployment. The deployment process, in turn, demands robust automation mechanisms that ensure stability, scalability, and monitoring efficiency.

An essential aspect of this research is the implementation of data caching between pipeline steps, enabling the reuse of intermediate build results, such as dependency installations and compiled assets, which signifi-

cantly reduces build times and resource consumption. Furthermore, effective artifact management plays a crucial role in optimizing CI/CD workflows. By preserving essential build outputs, such as compiled packages, and making them available for subsequent pipeline stages, the overall efficiency of the deployment process can be enhanced. Proper retention policies further optimize storage usage, while automated cleanup mechanisms prevent storage bloat, ensuring optimal resource utilization in shared environments.

Additionally, optimizing the usage of CI/CD runners is a critical factor in minimizing pipeline execution latency. This involves strategies such as reusing runners for consecutive commits within the same branch, skipping redundant builds for unchanged components, and prioritizing lightweight tasks to reduce computational bottlenecks. These measures collectively improve resource efficiency and accelerate the deployment cycle.

By integrating these practices, the proposed methodology enhances the efficiency, modularity, and scalability of modern web system architectures, allowing development teams to focus on innovation rather than repetitive infrastructure-related tasks. The research employs a combination of experimental evaluation, architectural design, and analytical modeling to validate the proposed approaches.

The object of this work is the infrastructure deployment process for multi-frontend projects, while the subject of this study is the optimization of CI/CD pipelines and containerized deployment strategies using reusable libraries.

The subject of this study is the optimization of CI/CD pipelines and containerized deployment strategies using reusable libraries.

The purpose of this study is to propose an efficient and scalable approach to infrastructure deployment by integrating reusable library builds, CI/CD pipelines.

1. Problem statement

Multi-frontend architecture enables the development and maintenance of multiple frontend applications within a shared infrastructure, promoting consistency, reusability, and centralized management. However, this approach introduces several challenges, particularly in library reuse, dependency management, and build optimization. Ensuring efficient development processes requires the creation of reusable packages that can be seamlessly integrated into various projects while maintaining adaptability for brand-specific customizations. Monorepo-based development simplifies dependency tracking and version control but also increases the risk of unintended cross-project dependencies and potential conflicts arising from shared library modifications.

A critical aspect of multi-frontend design is the clear separation between business logic and styling, allowing for the consistent implementation of shared features while preserving brand-specific flexibility. Achieving this modularity, however, presents significant technical complexities, necessitating a well-structured architecture that maintains both maintainability and extensibility. Furthermore, build optimization remains a pressing challenge, as shared libraries often introduce unused dependencies, inflating the final bundle size. Implementing tree-shaking and other optimization techniques is crucial to eliminating redundant code, improving application performance, and ensuring scalability.

To address these challenges, this study aims to:

– design pipeline configurations tailored for multi-frontend architectures within a monorepo environment to ensure an efficient and scalable development process;

– conduct an experiment and measure the efficiency of the proposed build method for projects in a multi-frontend architecture.

2. Review of the literature

The study presented in [1] explores an approach to automating software deployment pipelines through the integration of Continuous Integration, Continuous Security, and Continuous Deployment. The research demonstrates that this methodology contributes to improved code quality, mitigates security vulnerabilities, and reduces deployment time by leveraging tools such as Jenkins and Veracode. However, despite these benefits, the study does not address the optimization of CI/CD processes for multi-frontend projects

within a monorepo architecture, as its primary focus lies in broader CI/CD automation strategies rather than specific implementation scenarios.

Similarly, the study [2], which investigates the restructuring of CI/CD pipelines, provides valuable insights into the evolution of CI/CD practices within open-source projects. The study introduces a taxonomy comprising 34 restructuring actions that impact the maintainability, performance, and security of CI/CD pipelines. While this research highlights the growing adoption of Docker as a containerization technology, it does not examine the integration of reusable libraries within a monorepo structure, which represents a more specialized case within the broader context of CI/CD pipeline optimization.

The research presented in [3] examines the automated deployment of web applications within cloud infrastructures, specifically utilizing AWS CodePipeline. The findings indicate that AWS CodePipeline significantly enhances deployment efficiency and reduces time-to-market by streamlining process management. However, the study primarily focuses on AWS-specific solutions and does not address broader scalability concerns in CI/CD for multi-frontend projects.

Additionally, prior research on cloud deployment methodologies incorporating Docker and Kubernetes underscores the effectiveness of cloud-native CI/CD solutions. These studies outline practices for integrating containerization within CI/CD pipelines; however, they do not specifically aim to address the complexities associated with managing multi-frontend monorepos and the efficient reuse of shared libraries, as these aspects pertain to a particular subset of deployment challenges rather than the primary scope of the research.

A more recent study [4] explores the impact of CI/CD automation on developer productivity, demonstrating that automated pipelines can reduce deployment errors by 70% and unplanned work by 22%. The research draws on real-world case studies from the technology, finance, and e-commerce sectors, illustrating how CI/CD implementation accelerates release cycles and improves software quality. However, this study does not address the complexities of deploying multi-component frontend systems within monorepos or optimizing CI/CD pipelines for scalable frontend deployment.

The study [5] investigates CI/CD pipelines in cloud infrastructure deployment, focusing on the integration of orchestration tools like Terraform and Kubernetes. While this research highlights the efficiency gains from DevOps automation, it does not consider the optimization of CI/CD processes for frontend development or the reuse of shared libraries, making its findings less applicable to multi-frontend monorepos.

In the work [6] is discussed the advantages of micro-frontend architecture, emphasizing its benefits in modularity, scalability, and independent team workflows. The study argues that micro-frontends allow teams to develop and deploy different parts of an application independently, improving maintainability and release efficiency. However, despite these advantages, the study does not explore how micro-frontends can be effectively integrated into CI/CD pipelines or orchestrated within containerized environments, leaving a gap in the understanding of deployment automation for scalable frontend systems.

In summary, while existing studies provide valuable insights into CI/CD optimization, containerization strategies, and deployment automation, a comprehensive framework that holistically addresses the challenges of multi-frontend infrastructure deployment in monorepos remains underexplored. This highlights the need for further research into developing an efficient and scalable infrastructure deployment strategy that optimizes CI/CD pipelines and containerized deployments while ensuring the effective reuse of libraries.

3. Materials and methods

This study addresses to design pipeline configurations tailored for multi-frontend architectures within a monorepo environment to ensure an efficient and scalable development process.

The following tools and technologies were employed in the experimental framework of this research:

– GitLab & GitLab CI: GitLab serves as the primary platform for version control and continuous integration/continuous deployment (CI/CD). GitLab CI facilitates the automation of the entire CI/CD pipeline, including building, testing, and deployment processes;

– YAML: YAML files were used extensively for defining infrastructure and application configurations, including CI/CD pipeline stages in GitLab.

To optimize CI/CD pipelines, caching mechanisms were implemented within GitLab CI to reduce build time and optimize resource utilization. These mechanisms specifically target dependencies and build artifacts, ensuring faster rebuilds by caching intermediate build steps. Additionally, a novel approach to multi-frontend development was introduced by adopting a monorepo structure, which facilitates shared package management and modular library reuse. Unlike traditional sequential build processes, the proposed method enables parallel package builds, significantly reducing overall build times and improving efficiency. This structured approach ensures seamless dependency management across multiple frontend applications, leading to a more maintainable and scalable architecture.

Cache storage mechanisms were integrated into GitLab CI to securely store build outputs. This cache is utilized across multiple pipeline stages, enabling the reuse of previously generated outputs and minimizing redundant builds. Furthermore, a strategic cleanup mechanism was introduced to manage storage efficiently, ensuring that outdated build artifacts and logs are regularly purged to prevent excessive storage consumption and system bloat [7].

This study contributes to the field of informatics by proposing an optimized approach for multi-frontend CI/CD pipelines that enhances efficiency through structured monorepo management and parallelized build execution. By shifting from traditional sequential builds to parallelized package processing, the research provides a scalable methodology that can be applied to other complex frontend architectures. Additionally, the integration of caching and modular library strategies demonstrates a systematic way to optimize resource utilization while maintaining consistency across multiple frontend applications. These findings offer a practical framework that can be extended to various CI/CD environments, promoting best practices in pipeline optimization for large-scale software projects [8].

The development of multi-frontend pipeline configurations was facilitated by adopting a monorepo architecture, which houses multiple frontend applications, libraries, and shared components. This structure promotes code reuse and maintains consistency in dependency management, as all projects within the monorepo reference a unified set of libraries and components. To accommodate multiple frontend applications efficiently, GitLab CI pipelines were configured to handle build and deployment processes in parallel. This setup ensures that each frontend application can be developed and deployed independently while still leveraging shared libraries and components across the entire monorepo [9].

By implementing these methodologies, the research demonstrates the practical impact of CI/CD pipeline optimizations in a real-world multi-frontend development environment.

4. Experiments

To validate the proposed optimizations, a series of controlled CI/CD pipeline execution tests were conducted. The experiments aimed to compare the efficiency of pipeline execution before and after applying optimizations. The following conditions were maintained to ensure test consistency:

– each test was conducted within the same time frame to avoid variations due to external factors;

– the same package versions and conditions were used across all test executions;

– the same virtual machine instance was utilized for all tests to eliminate hardware-related discrepancies;

– all experiments were executed on free-tier GitLab runners to reflect real-world usage scenarios;

– caches and artifacts were cleared before each test run to ensure a clean environment and eliminate residual effects from previous executions.

5. Results

In a monorepo setup, shared libraries and packages are extensively utilized to optimize the build pipelines of all applications within the repository. The build process is structured to maximize efficiency while minimizing both time and cost (Figure 1).

The process begins with installing third-party dependencies required for subsequent stages of usage and building. To optimize this step, caching mechanisms are employed to retain previously installed dependencies unless library updates occur. This prevents redundant downloads and speeds up the build process.

Once dependencies are installed, the next stage involves building the necessary packages. Since packages in a monorepo are typically independent, they can be built in parallel, significantly reducing the overall build time. Each build produces artifacts that are cached per commit, ensuring they can be reused in subsequent stages without requiring a full rebuild.

Following the build process, testing is executed immediately for each package. Running tests in parallel ensures that potential issues are caught early while maintaining a continuous feedback loop for developers. This approach enhances stability and reliability, as faulty builds are detected before they impact the next stages.

After successful package builds and tests, applications within the monorepo are built. These applications rely on the previously built and tested packages but, similar to packages, are often independent of each other. As a result, the application build process can also be parallelized. The generated artifacts are cached per commit, improving efficiency in future runs and avoiding unnecessary recompilations.

The final stage involves assembling and packaging the applications, typically by creating Docker images for deployment. Depending on the architecture, a single Docker image may contain multiple applications, or separate images may be built for each. Caching at this stage further enhances deployment efficiency, ensuring that only modified components are rebuilt.

By leveraging caching strategies and parallel execution at both the package and application levels, the CI/CD pipeline significantly reduces build time and improves resource utilization. This optimization enables faster iteration cycles, minimizes bottlenecks, and ensures a streamlined deployment process, making monorepo-based development more scalable and efficient [10].

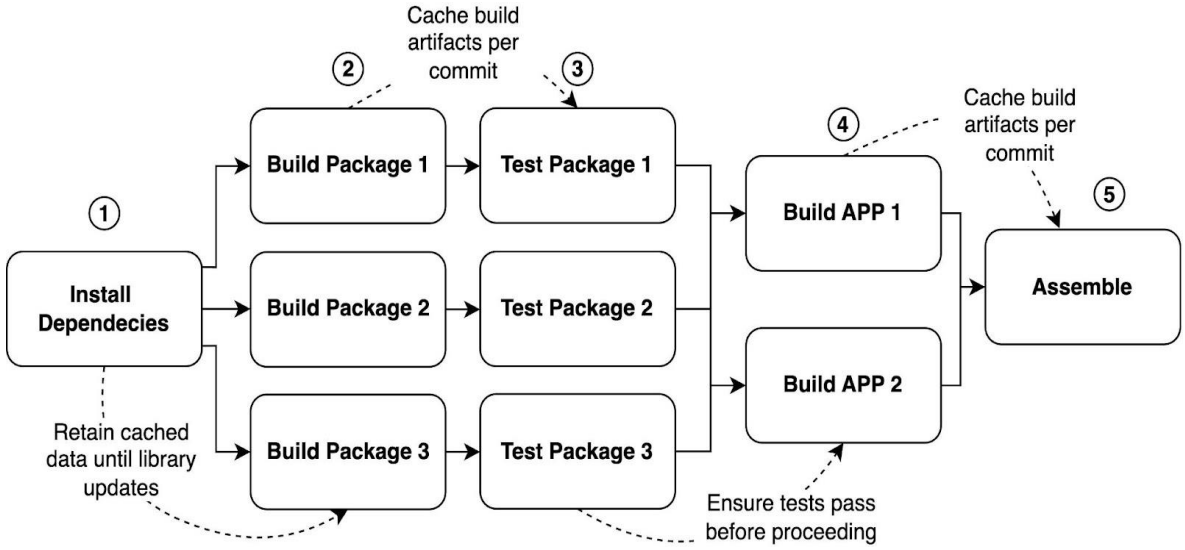

Fig. 1. CI Pipelines stages and jobs

A series of experiments were conducted to evaluate the impact of optimizing CI/CD pipelines by introducing parallel execution strategies. The pipeline consists of multiple stages, including dependency installation, package building, testing, application compilation, and Docker image creation. The initial configuration (sequential execution) was compared against an optimized version that leveraged parallelism in key stages. The results are summarized (Figure 2) with the representing the baseline sequential execution and the optimized parallel execution. In cases where multiple jobs were executed in parallel, the total stage duration was determined by the longest-running job.

The sequential pipeline required 1661 seconds (27 minutes and 41 seconds) to complete, whereas the optimized parallel pipeline completed in 1150 seconds (19 minutes and 10 seconds), representing a 30.7% reduction in total execution time.

The time required for installing dependencies showed a marginal improvement, decreasing from 47 seconds to 45 seconds. This suggests that this stage has only one task, and the optimization did not bring significant benefits. The 2-second difference can be considered within the margin of error due to the runner's expected execution time for this stage.

Stage	Sequential jobs (sec)	Parallel jobs (sec)
(1) Install Dependencies	47s	45s
(2) Build 3 packages	614s	232s / **278s** / 224s
(3) Testing 3 packages	283s	104s / **246s** / 115s
(4) Build 2 applications	564s	291s / **428s**
(5) Build Docker image	153s	153s
Total waiting time	**1661s (27m 41s)**	**1150s (19m 10s)**

Fig. 2. Comparing CI/CD pipelines in sequential and parallel modes

The initial sequential build of the three packages took 614 seconds, while parallel execution distributed the load across three parallel tasks (232s, 278s, and 224s). The total time for this stage was determined by the slowest task (278 seconds), resulting in an improvement of 54.7%. The 278-second time was longer than the others due to the runner's waiting time to execute the task, but this time was not eliminated to reflect the task's execution under real conditions.

The sequential execution of testing required 283 seconds. In the parallel approach, three separate test jobs ran simultaneously (104s, 246s, and 115s), with the longest job completing in 246 seconds, reducing the time spent in this stage by 13.1%. The 246-second time can be explained by the runner's waiting time to prepare for the task, similar to the previous case.

The sequential execution for two applications required 564 seconds, while the parallel execution resulted in job durations of 291s and 428s, with the longest job determining the total stage time (428 seconds). This optimization led to a 24.1% reduction in execution time.

The Docker image build process remained unchanged at 153 seconds, indicating that this stage was not parallelized or optimized further.

The experimental results demonstrate that implementing parallel execution in CI/CD pipelines significantly reduces total build time. The optimized approach reduced the overall execution time by approximately 511 seconds (30.7%), with the most substan-

tial improvements observed in the package building and application compilation stages, when tasks were in parallel.

6. Discussion

The integration of caching mechanisms throughout the CI/CD pipeline significantly reduces redundant computational tasks, thereby improving build and deployment efficiency. This approach is particularly effective in large-scale systems, where minimizing build times directly contributes to overall system responsiveness and productivity. The ability to execute independent jobs concurrently not only accelerates processing but also optimizes the utilization of computational resources. These advantages align with previous research findings, which highlight the effectiveness of caching and parallelization in CI/CD workflows for reducing latency and operational overhead [11].

The experimental results confirm the efficiency of parallel execution strategies, demonstrating an overall 30.7% reduction in total pipeline execution time. The most substantial improvements were observed in the package building (54.7% improvement) and application compilation (24.1% improvement) stages, where parallelization allowed tasks to be distributed efficiently across multiple jobs. Notably, the testing stage exhibited a more modest 13.1% reduction, suggesting that further refinements in test execution strategies could yield additional performance gains. These findings reinforce the practical benefits of parallelism, particularly in monorepo-based multi-frontend architectures where independent components can be processed concurrently.

However, despite its advantages, the proposed methodology is not without limitations. The reliance on efficient caching mechanisms necessitates robust cache invalidation strategies to prevent inconsistencies in deployments. Additionally, the effectiveness of parallelized job execution is contingent upon the granularity of task decomposition; improper segmentation may lead to synchronization issues or resource bottlenecks. This is particularly relevant in stages where dependencies between tasks are not entirely independent, as observed in the testing phase where parallel execution yielded only a moderate improvement. These limitations underscore the necessity for further empirical research into the impact of caching granularity and job segmentation strategies on CI/CD performance across varying system architectures [12].

From a practical standpoint, the adoption of this methodology is particularly relevant for projects involving complex multi-frontend architectures. By employing monorepos and automated CI/CD pipelines, development teams can achieve streamlined workflows, faster release cycles, and improved scalability. The implications extend beyond individual software projects, as these principles can be adapted to broader software engineering practices to enhance deployment reliability and system maintainability. Future research should focus on refining optimization techniques, investigating alternative caching policies, and exploring hybrid deployment strategies that balance automation with manual oversight to further enhance system robustness [13].

Conclusions

This research has addressed the challenges of infrastructure deployment in multi-frontend architectures by proposing an efficient and scalable approach that integrates reusable library builds, CI/CD pipelines. The scientific problem of optimizing deployment workflows in multi-frontend systems has been effectively tackled through the introduction of a structured monorepo-based development strategy, enabling better management of shared libraries and reducing code duplication. This represents an improvement over existing methodologies, enhancing consistency across projects and facilitating more efficient collaboration within development teams.

A key scientific contribution of this research is the advancement of CI/CD pipeline optimization techniques. The integration of caching mechanisms at all stages of the pipeline and the parallelization of pipeline jobs have been systematically implemented, leading to measurable reductions in build and deployment times. This optimization has allowed for improved resource utilization, re-

ducing operational costs and enhancing system scalability.

The scientific novelty of these results lies in the comprehensive integration of monorepo practices including CI/CD pipeline optimizations in a unified approach. Unlike prior research, which primarily focused on isolated aspects of deployment efficiency, this study provides a holistic methodology that ensures both speed and reliability. The findings demonstrate that parallelization and caching strategies, when systematically applied, significantly decrease build times and increase deployment stability.

From a practical perspective, the proposed methodology enables organizations to streamline their development workflows, reduce operational overhead, and achieve faster, more reliable deployments. The recommended framework can be adapted for various large-scale projects, ensuring consistent and efficient infrastructure management. Future research should explore further refinements in caching invalidation strategies, adaptive CI/CD pipeline configurations, and hybrid deployment models that balance automation with manual oversight. Additionally, investigating the impact of these optimizations on security and compliance within multi-frontend architectures will be essential for broader industry adoption.

References

1. Deepak, R.D. and Swarnalatha, P., 2019. Continuous Integration-Continuous Security-Continuous Deployment Pipeline Automation for Application Software (CI-CS-CD). International Journal of Computer Science and Software Engineering, 8(10), pp.247-253.

2. Zampetti, F., Geremia, S., Bavota, G. and Di Penta, M., 2021, September. CI/CD pipelines evolution and restructuring: A qualitative and quantitative study. In 2021 IEEE International Conference on Software Maintenance and Evolution (ICSME) (pp. 471-482). IEEE.

3. Singireddy, S.R., 2024, Analysis of continuous integration/continuous deployment (ci/cd) pipelines for automated cloud infrastructure management. International Journal of Core Engineering & Management, 7 (10), pp. 45-57.

4. Chittala, S., 2024. Enhancing developer productivity through automated ci/cd pipelines: a comprehensive analysis. International journal of computer engineering and technology (ijcet), 15(5), pp.882-891.

5. Slagsvold, A.F., 2023. Exploring CI/CD pipelines for cloud infrastructure deployment: Can one increase efficiency through amalgamation? Master's thesis. OsloMet – Oslo Metropolitan University, Department of Computer Science, Faculty of Technology, Art and Design.

6. Gashi, E., Hyseni, D., Shabani, I. and Çiço, B., 2024, June. The advantages of Micro-Frontend architecture for developing web application. In 2024 13th Mediterranean Conference on Embedded Computing (MECO) (pp. 1-5). IEEE.

7. Ghaleb, T.A., Abduljalil, O. and Hassan, S., 2024. CI/CD Configuration Practices in Open-Source Android Apps: An Empirical Study. arXiv preprint arXiv: 2411.06077.

8. Thatikonda, V.K., 2023. Beyond the buzz: A journey through CI/CD principles and best practices. European Journal of Theoretical and Applied Sciences, 1(5), pp. 334-340.

9. MUSTYALA, A., 2022. CI/CD Pipelines in Kubernetes: Accelerating Software Development and Deployment. EPH-International Journal of Science And Engineering, 8(3), pp. 1-11.

10. Jani, Y., 2023. Implementing continuous integration and continuous deployment (ci/cd) in modern software development. International Journal of Science and Research (IJSR), 12(6), pp. 2984-2987.

11. Reddy, S., Catharine, A. and Shanthamalar, J.J., 2024, May. Efficient Application Deployment: GitOps for Faster and Secure CI/CD Cycles. In 2024 International Conference on Advances in Modern Age Technologies for Health and Engineering Science (AMATHE) (pp. 1-7). IEEE.

12. Brousse, N., 2019, April. The issue of monorepo and polyrepo in large enterprises. In Companion proceedings of the 3rd international conference on the art, science, and engineering of programming (pp. 1-4).

13. Shabu, S.J., Kumar, S.P., Pranav, R. and Refonaa, S., 2023, April. Development of an E-Commerce System using MEAN Stack with NX Monorepo. In 2023 7th International Conference on Trends in Electronics and Informatics (ICOEI) (pp. 58-62). IEEE.

The article has been sent to the editors 29.04.25.
After processing 07.05.25.
Submitted for printing 30.06.25.