Enhancing Software Reliability: The Role of Automated Continuous Integration and Continuous Delivery

Sandip J. Gami
Independent Researcher
Brambleton, VA, USA

Chandrasekhar Rao Katru
Independent Researcher
Indian Land, South Carolina

Kevin N. Shah
Independent Researcher

ABSTRACT
Software reliability is among the ultimate goals of the modern software engineering discipline. Continuous Integration and Continuous Deployment, both CI/CD, have dramatically transformed the development cycle by incorporating testing and automatic deployment. In this paper, let's discuss the effects encompassing the implementation of CI/CD pipelines with automation characteristics. Automation is defined across three areas with quantified practices, tools, and measures and underscored with research evidence and examples. This paper explains CI/CD as automation and shows that automated testing, deployment strategies, and monitoring contribute to stable software, minimize human involvement by lowering failure rates, and optimize speed and time. Best practice solutions for effective CI/CD adoption are discussed, and aspects of scalability, security, and organizational integration are covered.

Keywords
Continuous Integration, Continuous Delivery, Software Reliability, Automated Testing, DevOps, Monitoring Tools.

1. INTRODUCTION
Software reliability is a crucial aspect of modern software development, determining the probability that a program will function without failure for a specified duration. This is particularly vital in today's interconnected world, where system failures can lead to significant financial losses and operational disruptions. One of the key methods to enhance software reliability is through Automated Continuous Integration and Continuous Delivery (CI/CD) pipelines, which streamline the development workflow by automating code integration, testing, and deployment. CI/CD automation enhances software quality through rigorous testing at various stages, reduces human errors, and accelerates time-to-market by enabling frequent and incremental updates. Additionally, automated pipelines facilitate improved collaboration between development and operations teams, ensuring visibility, continuous monitoring, and real-time feedback for efficient issue resolution [1-4].

Improving software reliability is essential as it minimizes downtime, enhances user experience, and reduces long-term maintenance costs. Software failures or frequent disruptions can negatively impact business operations, brand reputation, and customer trust. Organizations can mitigate such risks by implementing robust automated testing, continuous monitoring, and proactive issue resolution strategies. Furthermore, reliability plays a pivotal role in supporting agile development and continuous delivery, ensuring that new software updates and enhancements do not introduce defects into production systems. Secure and well-tested software fosters business continuity and strengthens customer confidence in digital platforms. Industries like healthcare, finance, and e-commerce particularly rely on highly reliable software to ensure compliance with regulatory standards and data protection requirements [5,6].

Beyond operational efficiency, software reliability contributes to an organization's long-term growth by enhancing scalability, performance, and risk management. Reliable software systems can handle fluctuating workloads efficiently and ensure optimal performance under high traffic conditions. Additionally, organizations leveraging proactive security measures—such as vulnerability scanning and compliance checks—can detect and mitigate potential threats before software deployment. By maintaining a stable and reliable software environment, companies can accelerate time-to-market, allocate resources effectively, and continuously improve their software delivery lifecycle. Ultimately, reliability in software engineering is not just a technical necessity but also a strategic imperative for businesses aiming to remain competitive in the evolving digital landscape.

2. LITERATURE SURVEY

2.1 Evolution of CI/CD
Continuous Integration (CI) and Continuous Delivery (CD) processes are the descendants of Agile software development that focuses on integrated cooperation, density, and the swift delivery of the software. When Agile practices became more popular, there was a need to provide faster and more stable methods for deploying software more often. [7-11] CI/CD then came into the scene as the solution to meeting these needs. Some of the earliest and seminal codified influences for current CI/CD include the Continuous Delivery book. They focused on the fact that minimizing the issues related to their work is important, as that is done with the help of manual deployment and integration. This was aimed at bringing about the continuous delivery of the software without flushing the quality and stability of the code. CI/CD has moved on from basic automated build systems to fully automated systems covering code commit to production release, again mitigating all integration and deployment risks.

2.2 Core Principles of CI/CD
CI/CD hymenia-times several fundamental concepts that aim to facilitate the enhancement of building software delivery pipelines. Among those is the one rooted in the culture of continuous integration of small, manageable chunks of code into the repository, perhaps multiple times a day. This simplifies the integration process and makes detecting problems possible before they get out of hand. Another principle is automated testing, where each change is run through a series of tests (unit, integration, regression, etc.), with results reported immediately to developers. Such automation helps reduce reliance on human observation and speeds up the process of defect identification. Finally, immutable infrastructure is particularly useful for CI/CD

pipeline deployments. This means that during the deployment, rather than updating only parts of the system or container at a time, the whole system is updated at once to match the configuration of another environment, for example, to avoid configuration drift and potential deployment failure.

2.3 Empirical Studies

Scientific research on CI/CD practices helps to understand its effects on the quality of created software and the organization's performance. For example, in Study A, authors discovered that organizations employed CI/CD experienced a 40% decline in production failures. CI/SD also reduces this because it increases visibility and has automated testing built into the continuous integration and continuous delivery pipelines. This is supported by study B, which indicates that another CI/CD element, automated testing coverage, boosted overall systems reliability by 35%. These studies show customer value by helping to enhance software quality, release software faster to the market, and minimize instances of making a product in production. In these studies, there have been strong signs of improvement, especially when mentioning that both implementation and choice of tools should be proper.

2.4 Tools and Technologies

Its deployment is primarily contingent on the adequate choice of tools used in the CI/CD process. Some tools considered standard for CI/CD have evolved over time, and here they are! Jenkins is one of the leading CI/CD tools, being an automation server of open-source code, providing flexibility in the construction, testing, and deployment of software due to its extensive plugin base. Another tool is GitHub Actions, which is tightly tied to version control and enables developers to perform workflows from the repositories used on GitHub. GitLab CI/CD is an integrated product that comes with a very solid pipeline stack, allowing for complete, extensive, first-party offerings right from Version Control to Continuous Deployment. CircleCI – is a cloud-based CI/CD service that is convenient in terms of pipeline handling and is designed to work with cloud-based teams that require the most scalability. These are some of the tools that, together with the mentioned ones, give you the basics to start a full CI/CD pipeline solution since they have features that allow for choosing according to the environment and needs of a team.

2.5 Gaps in Literature

As observed with prior literature, the advantages of CI/CD have been extensively discussed in the past, and there is a dearth of knowledge on the factors that companies encounter while extending their CI/CD pipeline. For instance, the research manifests that the vast majority of the work is on Small to Medium Scale projects, and there is less understanding of the issues one would encounter when applying CI/CD for Larger Enterprise Systems. Moreover, there is still a gap in how to adapt legacy systems to CI/CD initiatives. Most organizations continue to use outdated technologies and structures for DevOps that do not correlate with CI/CD tools, which, in many cases, makes the change to automated processes painstakingly slow. It is also evident that there is little research on how security and compliance are managed within CI/CD pipelines in firms, especially those in heavily regulated industries. By recognizing these issues, there will be a greater appreciation for the concerns surrounding CI/CD and its recognition, easing any implementation process for organizations studying this methodology.

3. METHODOLOGY

3.1 Research design

The study used both qualitative and quantitative research methodologies in order to produce a holistic assessment of the effects that the automated CI/CD implementation has on software reliability. This work relied on qualitative and quantitative measures, such as defective rates, deployment frequencies, and system downtimes, to partition proven reliability enhancements. This data has, therefore, been obtained from CI/CD pipelines in real-world organizations where pre and post-implementation comparisons can be made. [12-16] To support this quantitative analysis, qualitative case-study research was performed via interviews with DevOps teams to elucidate their experiences concerning automation. The fact that both approaches were employed in the study meant that CI/CD's contribution to improving software reliability in the manufacturing company was well established from the perspective of practical concern and the understanding of the environment into which the innovation was integrated.

3.2 Framework for CI/CD Implementation

CI/CD, in general, refers to a processes-oriented approach in the implementation of a system that is aimed at the integration and testing of code and its delivery and monitoring. To further elaborate, the phases of CI/CD have been illustrated in a pictorial form in Figure 1, depicting how the code passes through the development to the deployment phase while checking the reliability at every phase can be viewed. The key stages of this framework include:

Code Integration: This stage requires developers to make frequent commits of new code versions to the common repository. In other words, revision control systems like Git or GitHub enable us to trace changes, work with the branches, and more. Retrieval and merging often reduce integration problems because they often catch the problems before they become big.

Automated Build: Once code is committed there is a process that compiles the code into executable binaries: 'build'. Continuous delivery automation, as in Jenkins or CircleCI, confirms that the code and its integration are genuine and creates test and deployment assets.

Automated Testing: A key mileage that assures that the new codes will not contain new defects or that the new changes will not negatively affect the existing functionalities. From the perspective of unit, integration, and regression testing, tests are run in particular, whereby Selenium or Junit comes in. It is the successful passing of the application to the further stages that shows that it is sufficiently stable.

Deployment to Staging: This is the environment tested using the code taken to the production environment, which is a copy of this one. It also provides an environment for extra testing, such as performance and security testing, on aspects that can be done on the live system without influencing it.

Continuous Delivery or Deployment: when all tests in staging are passed, all staging code goes to the production automatically or semi-automatedly. Continuous delivery always builds the code to the production level, but in continuous deployment, it is mostly automated.

Monitoring and Feedback: Monitoring after the deployment using application metrics like Prometheus and Grafana makes coordination and checking if the system is running well possible. Performance data metrics, error logs, and user

behavior are collected through feedback loops to use in future rounds as well as pipeline improvements.

3.3 Data Collection Techniques

Surveys with Developers: The interviews were conducted with developers and DevOps engineers currently practicing CI/CD practices. More specifically, the survey was concerned with their successful practices of CI/CD pipelines, problems met, and potential gains in software reliability. Questions also asked about how far automation has been taken, which kinds of testing are being done, and how frequently releases are being conducted. Although these responses offered a primarily quantitative assessment of automation's effects, they gave a qualitative understanding of how automation affects the productivity of lines of work and software quality.

Log Data from CI/CD Tools: Log data were comprehensively gathered from the CI/CD systems from Jenkins, GitLab, and CircleCI. This information entailed build success rates, test coverage, failure profile, and the time taken to deploy solutions. Such logs enabled the author to have raw quantitative measures of the reliability increase and give them information about bottlenecks or repeated errors in the pipeline. The logs were also used for reference when determining the level of performance before and after the implementation of CI/ CD.

Feedback from End-Users on Reliability Improvements: Surveys were conducted on the end users of the software deployed in the market to assess the effect of CI/CD on the reliability of the developed product. Questionnaires and focus groups investigated users' impressions of the application's stability, speed and readiness after installation. External data captured from the end-users was crucial as it helped establish how specific enhancements done in and through CI/CD made a difference on the users' end. To this end, the research took a broad functional angle to guarantee that both the technical and the user-based impacts were captured.

3.4 Tools Used in Experimentation

Pipeline Setup: Jenkins, GitLab: When it came to creating pipelines for continuous integration and continuous delivery, two main tools that were chosen for their versatility and usage in many projects are Jenkins and GitLab. Jenkins, an open-source automation server, provided solutions for building and deploying versatile, sophisticated pipelines made from various plugins. Most of the time, it was handy for managing build, test, and deployment activities successfully. GitLab, in particular, offered versioning and CI/CD pipeline orchestration right on the platform. It offered a simple setup, understanding, and use of configuration syntax, including other attributes such as code reviews, making it ideal for dealing with fewer teams and projects.

Monitoring Tools: Prometheus, Grafana: Prometheus and Grafana were used to monitor or watch the CI/CD processes in a real-time fashion. Prometheus was used as a monitoring system, and it takes and stores information depending on the current data, such as build time and error rate, and uses resources at every stage of the CI/CD pipeline. It defined how to detect signs which presuppose reliability threats in advance. Prometheus was complemented well by Grafana, where the lower-level metrics could be turned into powerful dashboards that made it simple for teams to regularly monitor their applications or observe trends, detect potential pain points, or assess the overall status of their new deployments. Together, they made it a point that monitoring was a core to preserving the pipeline and the system.

Testing Frameworks: Selenium, JUnit: Selenium and JUnit were especially useful in the realization of automated testing within the CI/CD. People knew about selenium for its capacity to support end-to-end testing of web apps, which guaranteed that all the significant elements aimed at users were practical across browsers and platforms. JUnit, a framework for unit testing in Java, allowed for testing parts of code that consist of multiple lines, which could ensnare errors at the early stages. By including these frameworks in the pipeline, test coverage was provided that mitigated the introduction of defects during deployment. Together, these tools enhance the testing phase of an application, which is vital for increasing software reliability.

4. RESULTS AND DISCUSSION

4.1 Metrics of Software Reliability

This was evident from the results of pre- and post-implementation comparisons that measured several aspects of software reliability in a system that employed CI/CD pipelines. They are the Defect Density, Deployment Frequency, and Mean Time to Recovery (MTTR), all of which indicate nimbleness, quality of software and even the effect of the software on other aspects of a business.

Defect Density: This led to a reduction of the defect density parameter; the number of defects per thousand lines of code (KLOC) reduced from 4.2 to 2.1, thus resulting in a 50% improvement. This has been made possible through the application of automated testing within the CI/CD system, which reduces development errors. Unit, integration, and regression tests run automatically to assert that code changes are thoroughly tested before being introduced into the production system. These reductions in defects per unit of code have a direct positive relationship with software dependability and customer satisfaction.

Deployment Frequency: A key improvement was made in the deployment frequency, which has increased monthly to weekly, translating to a hundred percent improvement. Continuous integration (CI) / continuous delivery (CD) integrates build, testing, and deployment and allows those frequent updates to be released more readily with no reduction in quality. These features enable organizations to roll out their solutions quickly to address user feedback, fix bugs swiftly, and implement new features. This ability is also a way of increasing reliability since it eradicates large monolithic releases, which are often risky.

Mean Time to Recovery (MTTR): The mean time to recovery was reduced by 75%, from 8 hours to 2 hours. This metric highlights the reliability of systems that underwent CI/CD improvements. CI/CD pipeline monitoring and alerting ensure the quick detection of problems while testing and optimizing the rollback process, and hotfix deployment reduces outages. The decrease in the average MTTR shows that CI/CD pipelines prepare the teams to mitigate incidents while keeping service availability at optimal levels for the end-user.

Table 1: Improvement in Metrics Post-CI/CD

Metric	Pre-CI/CD	Post-CI/CD	Improvement (%)
Defect Density (per KLOC)	4.2	2.1	50%
Development Frequency	Monthly	Weekly	100%

Mean time to Recovery	8 hrs	2 hrs	75%

4.2 Impact of Automation

Automated Testing: Selenium for Web applications and JUnit for Units testing have been crucial when it comes to minimizing the errors left by human beings during development processes. These tools help reduce repetitive and time-consuming routine testing, such as regression testing and deployment validation, that might be easily overlooked. The ability to rapidly execute predefined test cases automatically across the checkpoint of interest with any code change safeguards the code from passing defects to the production environment. It has thus directly led to a 45% reduction in man-influenced mistakes and a dramatic reduction of the defect density witnessed in the production releases. Moreover, automation reduces the time needed to test and offer feedback on the piece of software, and developers can solve problems more quickly.

Monitoring: Prometheus and other similar real-time monitoring tools have changed how teams work with system reliability. These tools gather a large number of characteristics, like response time, error rate, and utilization of resources, that enable users to gauge the application's well-being and the CI/CD process's efficiency. In a propagative sense, monitoring allows teams to foresee probable failures before they injure client audiences, given that it identifies patterns and outliers early on. This way, it reduces response time when problems occur, hence allowing for a dependable and efficient system. Furthermore, the tuning data collected can be displayed as graphical models on a dashboard, such as a Grafana one, to provide easier visualization of the system's performance to different teams for optimizing aspects such as velocity or different resources.

4.3 Challenges Encountered

Integration with Legacy Systems: One of the hardest areas of implementing CI/CD pipelines is working with outdated applications that cannot incorporate many CI/CD automation tools and flows. These systems have probably been developed by employing older solutions or don't possess the APIs and modular constructions necessary for successful integration. Consequently, the attempt to introduce automated workflows creates the first problem for organizations which have had to maintain legacy systems for quite some time. Hence, it requires a lot of hassles, like writing extra scripts or having middleware layers to implement such systems, thus raising cost and time. Modernizing these systems is frequently necessary but rarely simple and can be costly.

Skill Gaps in Using CI/CD Tools: The second most reported problem is that the development and operation teams are not skilful enough in using CI/CD tools. Implementing CI/CD requires quite a shift in the mindset, possibly resulting in the team facing new concepts in automation frameworks, pipeline configuration, and monitoring tools. This high learning results in problems of slowness, wrong settings, and lack of readiness to change on the part of the systems. Organizations need to provide proper training and follow-ups through real-life practice related to tools like Jenkins, GitLab, and Prometheus to cope with this gap.

Tool Configuration Issues: Tool usage, which took an average of 42 minutes, was also problematic, with tool configuration cited as an issue by 15% of participants. CI/CD tools may contain complex configurations which involve cooperation with version control, testing, and deployment systems. Setting these tools up in a way that meets organizational work processes might be relatively herculean and very cumbersome when handling intricate use cases. The possibility of synchronizing dependencies wrong, permissions wrong, and environments wrong are also possible pitfalls for teams that have a negative effect on pipeline effectiveness. It just has to constantly optimize, organize, and bring in proper standardized procedures to fix these configuration problems.

Other Challenges: A significantly smaller but still significant portion of responses are grouped under the "Other" category, which includes issues such as organizational change management resistance, budget limitations, and the management of multiple clouds. As these challenges are normally organization-specific, they may need custom interventions. For instance, breaking employees' resistance by encouraging them to adopt automation and agility will work as a solution. However, strategic planning and priority setting for financial or infrastructural constraints shall also do a lot.

Table 2: Challenges in CI/CD Adoption

Challenge	**Percentage**
Integration with Legacy Systems	40%
Skill Gaps in using CI/CD Tools	35%
Tool Configuration issues	15%
Other Challenges	10%

4.4 Discussion of Findings

The introduction of CI/CD pipeline practices appears to bring significant advantages in improving software quality, completing the deployment process, and enhancing the performance of the development team. However, organizations must first overcome some initial hitches that accompany the full optimization of automation applications and continuous integration.

Faster Issue Resolution: Another emergent finding in this research is the ability to resolve issues more quickly in environments that have CI/CD in place. The end-to-end and integrated continuous testing and monitoring feedback provided by the implementations were important in early defect identification and system anomalies. Selenium and JUnit testing, for example, make it possible for a check to be done as soon as any code is committed; hence, problems cannot reach the production level. It effectively reduces the time that may be taken to diagnose a problem after a product has hit the market. Also, automated testing and integration let the developers rapidly check the system's reliability when something has changed, repair the problem as soon as possible, and thus reduce the amount of time that the system is out of service. Several teams mentioned that due to the integration of CI/CD tools, immediate feedback in the form of test results, build logs and messages notify the teams about the root cause of the problems faster. This led to faster and quicker deployments and response time, which has reduced the sheer manual involved in formally tracing defects, which usually slows down the release cycle in conventional development settings.

Proactive Issue Identification: Monitoring integrated with the CI/CD pipeline also helped to enhance issue resolution because teams can address the likelihood of slowing down

during the pipeline. Prometheus and Grafana offered an understanding of system conditions in real time and showed such problems as high fail rate, performance decrease, or limited resources. By having this real-time monitoring, teams could pick problems or abnormalities while on it and not wait for the team to happen in the production phase, which the investigating team would have to follow. For example, automated Continuous Delivery encourages whether a build fails consistently in one stage of the pipeline and helps the development team to fix the issue accordingly. In the same way, measures of resource usage acted like alarms that indicated service failures are likely to happen due to resource shortage, for example, server memory or CPU. It gave capability directly into the CI/CD pipeline and helped teams keep an application steady and reliable without waiting for end-users to report issues.

Enhanced Agility and Reliability: The most obvious benefit of particular CI/CD pipelines is flexibility and dependability in software development and online delivery. Continuous integration/delivery makes it possible to release a new update more frequently as they combine smaller and more frequent updates, leading to faster development of features, new code releases, resolution of bugs and performance optimization. The increased frequency of deployment also aids organizations in their ability to deliver fast responses to customer reactions to market trends and enhance overall user satisfaction based on the time factor between releases. Reliability increases as every deployment is fully tested, assessed, and verified. One can say that due to CI/CD continuity, there is no way a lower-quality code can get to the production environment. Also, the routine catastrophe of the system to a previous version in case of failure boosts the system's reliability. These capabilities are important to facilitate timely delivery of high-quality systems and enhancements as frequently as desired and when organizations need them.

Investment in Training, Tools, and Integration Strategies: Despite the added value that CI/CD offered, there were costs to be incurred to achieve these outcomes. First, all the development and operation teams should undergo training on how to use CI/CD tools before implementation. A lot of time is spent in training because understanding automation tools, which include Jenkins, GitLab and monitoring tools like Prometheus, can take some time to master. Training has to be implemented as a priority in organizations to compensate for the skills lacking, thereby affecting the application. Besides, choosing and configuring tool chains most resistant to destabilization is also critical. Jenkins for automation, selenium for testing, and Grafana for monitoring are tools that should fit into the existing infrastructure. Some of the challenges that may be incurred include the difficulty of implementing and managing these tools due to the high configurations of these gadgets alongside the existing older systems and efforts to work under strict resource constraints. Hence, there is a need for proper strategy and planning in order to have a correct integration and to be in a position to achieve the intended objectives. Lastly, integration strategies are important. Legacy systems, especially, are not compatible with modern CI/CD and, therefore, require significant amounts of time and money to be spent on refactoring code and its integration into CI/CD pipelines. Such integration may also warrant the settlement of compatibility problems and the generation of new working procedures to fit the automation aim.

5. CONCLUSION

CI/CD pipelines have become essential in modern software development, particularly in ensuring software reliability. These pipelines minimize human intervention in code integration, testing, and deployment, making the software delivery cycle more consistent, efficient, and error-free. By enabling frequent code commits that undergo automated testing, CI/CD practices help detect and address issues early, preventing major complications later in the development process. Automation tools such as Jenkins, GitLab, and CircleCI play a vital role in streamlining this process. Jenkins, as an open-source automation server, integrates with various version control systems to ensure that code is properly tested before deployment. Similarly, GitLab's built-in CI/CD capabilities enhance the development pipeline by automating everything from code construction to launch. This automation frees developers from routine troubleshooting, allowing them to focus on innovation while maintaining an optimal balance between software delivery speed and quality. As software systems grow more complex, CI/CD implementation reduces the burden on teams by simplifying testing, deployment, and updates. Additionally, continuous monitoring and feedback mechanisms within CI/CD pipelines help identify bottlenecks or performance issues early, ensuring proactive resolution before they escalate into significant challenges.

To fully leverage the benefits of automated CI/CD pipelines, organizations should adopt key strategies for optimal implementation. First, it is essential to invest in training for teams working with CI/CD tools such as Jenkins, GitLab, and CircleCI. These tools, while powerful, require a deep understanding to be used effectively. Proper training ensures that CI/CD processes run efficiently with minimal errors, ultimately shortening the software delivery cycle. Second, integrating robust monitoring solutions like Prometheus and Grafana into CI/CD pipelines enhances real-time visibility into pipeline activities, allowing teams to swiftly detect and resolve issues such as bottlenecks or build failures. Additionally, incorporating automation testing across the CI/CD lifecycle—spanning unit, integration, and regression testing—helps identify bugs early, reducing risks before deployment. Continuous testing and monitoring not only enhance software quality but also minimize manual effort, ensuring a more dependable and streamlined software delivery pipeline.

6. REFERENCES

[1] Humble, J., & Farley, D. (2010). Continuous delivery: reliable software releases through build, test, and deployment automation. Pearson Education.

[2] Kim, G., Humble, J., Debois, P., Willis, J., & Forsgren, N. (2021). The DevOps handbook: How to create world-class agility, reliability, & security in technology organizations. It Revolution.

[3] Duvall, P. M., Matyas, S., & Glover, A. (2007). Continuous integration: improving software quality and reducing risk. Pearson Education.

[4] Zheng, X. S., Wang, M., Matos, G., & Zhang, S. (2011). Streamlining user experience design and development: roles, tasks and workflow of applying rich application technologies. In Human-Computer Interaction. Design and Development Approaches: 14th International Conference, HCI International 2011, Orlando, FL, USA, July 9-14, 2011, Proceedings, Part I 14 (pp. 142-151). Springer Berlin Heidelberg.

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

[6] Hu, H., Jiang, C. H., Cai, K. Y., Wong, W. E., & Mathur, A. P. (2013). Enhancing software reliability estimates using modified adaptive testing. Information and Software Technology, 55(2), 288-300.

[7] Ergun, Ö., Gui, L., Heier Stamm, J. L., Keskinocak, P., & Swann, J. (2014). Improving humanitarian operations through technology- enabled collaboration. Production and Operations Management, 23(6), 1002-1014.

[8] Zampetti, F., Geremia, S., Bavota, G., & 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.

[9] Houerbi, A., Siala, C., Tucker, A., Rzig, D. E., & Hassan, F. (2024). Empirical Analysis on CI/CD Pipeline Evolution in Machine Learning Projects. arXiv preprint arXiv:2403.12199.

[10] Stolberg, S. (2009, August). Enabling agile testing through continuous integration. In 2009 agile conference (pp. 369-374). IEEE.

[11] van Deen, W. K., Cho, E. S., Pustolski, K., Wixon, D., Lamb, S., Valente, T. W., & Menchine, M. (2019). Involving end-users in the design of an audit and feedback intervention in the emergency department setting–a mixed methods study. BMC health services research, 19, 1-13.

[12] Bowen, P. L., Heales, J., & Vongphakdi, M. T. (2002). Reliability factors in business software: volatility, requirements and end- users. Information Systems Journal, 12(3), 185-213.

[13] Garg, S., Pundir, P., Rathee, G., Gupta, P. K., Garg, S., & Ahlawat, S. (2021, December). On continuous integration/continuous delivery for automated deployment of machine learning models using mlops. In 2021 IEEE fourth international conference on artificial intelligence and knowledge engineering (AIKE) (pp. 25-28). IEEE.

[14] Pratama, M. R., & Kusumo, D. S. (2021, August). Implementation of continuous integration and continuous delivery (ci/cd) on automatic performance testing. In 2021 9th International Conference on Information and Communication Technology (ICoICT) (pp. 230-235). IEEE.

[15] Yan, F., Ruwase, O., He, Y., & Chilimbi, T. (2015, August). Performance modeling and scalability optimization of distributed deep learning systems. In Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (pp. 1355-1364).

[16] Rine, D. C., & Sonnemann, R. M. (1998). Investments in reusable software. A study of software reuse investment success factors. Journal of systems and software, 41(1), 17-32.

[17] Zahir Irani, P. E. (2000). The propagation of technology management taxonomies for evaluating investments in information systems. Journal of management information systems, 17(3), 161-177.

[18] Mowad, A. M., Fawareh, H., & Hassan, M. A. (2022, November). Effect of using continuous integration (ci) and continuous delivery (cd) deployment in devops to reduce the gap between developer and operation. In 2022 International Arab Conference on Information Technology (ACIT) (pp. 1-8). IEEE.

[19] Kempe, E., & Massey, A. (2021, September). Perspectives on regulatory compliance in software engineering. In 2021 IEEE 29th International Requirements Engineering Conference (RE) (pp. 46-57). IEEE.

[20] Mubarkoot, M., Altmann, J., Rasti-Barzoki, M., Egger, B., & Lee, H. (2023). Software compliance requirements, factors, and policies: A systematic literature review. Computers & Security, 124, 102985.

[21] Kempe, E., & Massey, A. (2021). Regulatory and security standard compliance throughout the software development lifecycle.