

A MODERN AND THOROUGH APPROACH TO SOFTWARE DELIVERY WITH DEVOPS

Slobodan Čabarkapa¹, Djordje Mladenović², Emina Jeremić Marković³

Abstract: As we have already experienced, delivering software can be a very time-intensive and risky process, often plagued by integration issues and last-minute defects that can lead to dissatisfaction among users, business stakeholders, and development teams. Additionally, a lack of collaboration between teams can result in the implementation of problematic code. This paper offers a comprehensive overview of various software delivery practices designed to enhance the process's efficiency and effectiveness. These practices focus on continuous process improvement and the cultural changes needed to create a better working environment.

Key words: DevOps, Lean, Continuous Delivery, Software development practices.

1. INTRODUCTION

The traditional approach to software development involves a linear workflow that starts with concepts from marketing and business departments. These ideas are then passed to design and product management teams, who translate them into specifications and business requirements. Next, the development teams begin coding, which is later subjected to quality assurance testing to ensure it meets the original product requirements. Once approved, the IT Operations team deploys the code and sets up the necessary environments. However, security testing typically happens late in this process. Additionally, Development and IT Operations often have conflicting goals. Development focuses on introducing new features and change, while IT Operations aims to maintain stability and avoid change that could cause instability. These teams are often organized in separate silos, each focused on protecting their domain, leading to poor software quality, service issues, and a negative customer experience. This siloed approach fails to emphasize the common goal: the fast and continuous delivery of valuable software to customers.

The process typically involves at least five critical handovers between departments, where information is passed along and often distorted due to differing interpretations. From a time perspective, the siloed structure results in significantly longer lead times, as each department causes delays in the process. If we define Wait Time as the ratio of busy versus idle time, at 50% utilization, wait time is one unit, and at 90% utilization, it increases to nine units. With five handovers, the total wait time can reach 45 units [1]. As stated, “When people are trapped in this downward spiral for years, especially those downstream of development, they often feel stuck in a system that preordains failure and leaves them powerless to change the outcomes. This powerlessness often leads to burnout, marked by fatigue, cynicism, and feelings of hopelessness and despair.” [1] A change in working methods is necessary, and DevOps practices offer a promising solution.

High-performing IT organizations that adopt DevOps are outpacing their competitors. They deploy code 30 times more frequently, with a 60-times higher success rate, and achieve 200 times faster lead times for changes, alongside 168 times faster recovery from service interruptions [2]. These organizations not only experience higher growth but also report greater employee satisfaction and lower burnout rates. They can consistently provide stable, reliable, and secure services to their customers.

The DevOps approach enables organizations to establish a stable and efficient working model, where small, cross-functional teams can independently develop, test, and deploy code, delivering value to customers quickly, securely, and reliably. This approach maximizes productivity, fosters organizational learning, and builds a high-trust, collaborative culture that drives success in a competitive market.

DevOps practices complement Lean and Agile Software Development methods and can be categorized into three key areas [3]:

¹Academy of Technical and Art Applied Studies Belgrade, email: slobodan.cabarkapa@ict.edu.rs

²Academy of Technical and Art Applied Studies Belgrade, email: djordje.mladenovic@ict.edu.rs

³Center for Globalization Studies, email: emina.jeremic.markovic@cgs.org.rs

- Ensuring a smooth and fast flow of work from product design and development to operations, and ultimately to the customer.
- Enabling rapid feedback from operations to development allows for quick detection, recovery, and prevention of future issues.
- Cultivating a high-trust culture of continuous improvement that encourages initiative, experimentation, and organizational learning.

As defined by DevOps, the technology value stream is the process required to turn a business idea into a technology-enabled service that delivers value to customers [2]. The primary goal of Flow, Feedback, and Continuous Improvement is to ensure the fast and reliable delivery of valuable software to the customer.

This paper contributes to the understanding that DevOps represents a modern, holistic approach to software delivery that leads to high-quality products and customer satisfaction. More importantly, it emphasizes the need for a cultural and behavioral shift in software development practices—one that includes taking ownership of work, being accountable for products and services, optimizing workflows, and fostering teamwork in the face of challenges. When these practices are applied collectively, they lead to a cutting-edge software development process.

2. FLOW

The objective of DevOps Flow practices is to ensure a seamless transition of work from Product Design, through Development, and into Operations. Since software gains value only when it is deployed and utilized by customers, the focus is on speeding up the process and minimizing the time between a business idea and its actual deployment in production. To achieve this, it is crucial to make every aspect of the value stream transparent—whether it is product backlog tasks, software bugs, production incidents, or service requests. These elements can be managed through tools like Kanban or Scrum boards, enabling a pull-based work system. This approach also supports value stream analysis, which serves as a foundation for identifying areas of improvement. One common method for conducting this analysis is by creating a value stream map, illustrated in the figure below.

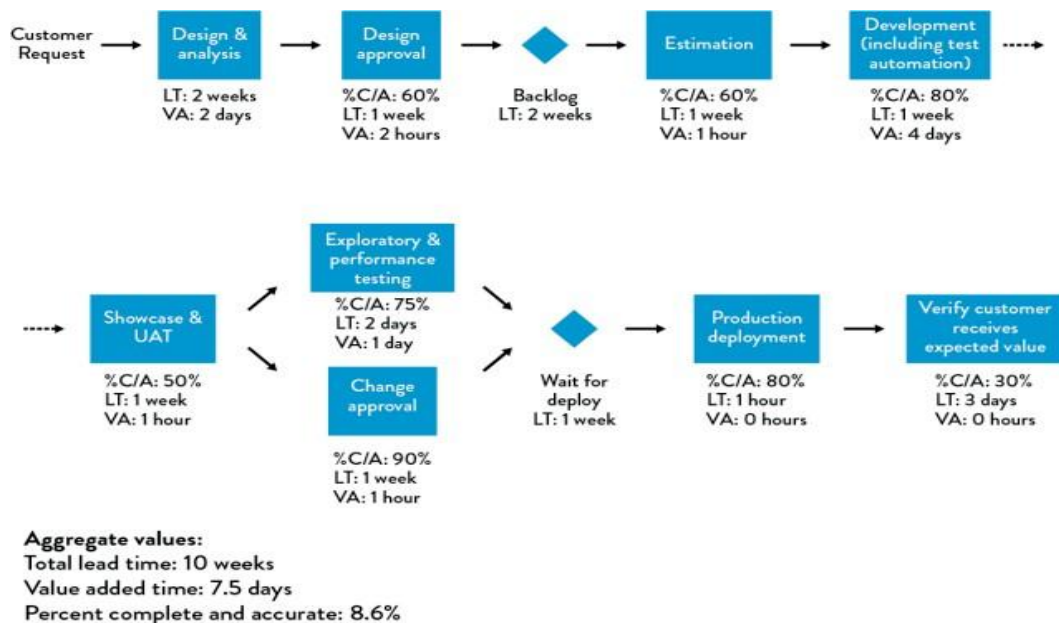


Figure 1 – Value Stream Map

Similar to manufacturing, and drawing on the Theory of Constraints, improving the flow of a process requires focusing on identifying and addressing the bottleneck. Enhancing the flow at any station before the bottleneck will only lead to an accumulation of work at the bottleneck itself (such as uncommitted code) while improving stations downstream of the bottleneck can result in idle workstations (such as a lack of code to test). This approach to improvement is essentially a continuous cycle, akin to the well-known Plan-Do-Check-Act (PDCA) cycle from Deming.

The factors that disrupt the productive flow in the technology value stream, often referred to as sources of waste, include defects, unfinished work, delays, unnecessary movement, task switching, non-value-added processes, excess features, non-standard or manual work, and reliance on "heroic" efforts [3]. The continuous improvement cycle aims to identify these sources of waste and inefficiency, making them visible, and systematically eliminating them.

To achieve this, we applied several techniques in our ongoing efforts to reduce waste and improve flow.

2.1. Reduce the Number of Handoffs

Our first step was to integrate the software development and quality assurance teams into a single, unified team. This integration went beyond just personnel—it also involved merging activities. Testing was incorporated early in the delivery iterations, with tasks like test case preparation starting from the outset. The introduction of automated testing significantly reduced wait times and helped foster a collaborative approach to testing, where the entire team was involved. This shift not only streamlined the flow of work but also enhanced flexibility across the process.

The second step focused on strengthening collaboration between Development and IT Operations. Depending on the situation, we employed two approaches: functional teams and integrated feature teams (also referred to as market-oriented teams). In the functional team setup, the IT Operations team was tasked with enabling full self-service capabilities for software development teams. This allowed development teams to complete tasks independently, without having to request assistance from IT Operations. Key activities in this context included provisioning environments and deployment tasks, which were critical handover moments that often caused delays and defects. For other tasks, each development team had a dedicated point of contact in the IT Operations team, responsible for understanding the development team's needs and ensuring the self-service platform met those requirements. In the feature team model, IT Operations became an integral part of the development team, working alongside developers on shared tasks. This approach fostered a natural exchange of knowledge, with developers taking on deployment tasks and system engineers gaining a deeper understanding of application functionality.

The third step concentrated on enhancing collaboration with product management. The objective was to create a shared understanding of each team's role and to improve the quality of requirements and specifications. We applied the technique of Specification by Example [4], which involved collaborative specification (e.g., through user story mapping sessions and refinement meetings), using examples to clarify requirements, and maintaining living documentation. This approach ensured that we were not only building the right features but also building them correctly.

2.2. Reduce Batch Sizes

Flow can be improved by reducing work intervals, minimizing batch sizes, building quality from the outset, and preventing defects from reaching downstream work centers. Batch size plays a crucial role in enhancing flow, as smaller batches lead to reduced lead times and improved quality.

In the software development value stream, batch size refers to the amount of undelivered code. Large batches often result in integration challenges, which directly extend lead times and reduce quality. Lower quality, in turn, causes significant disruptions for downstream work centers like Quality Assurance, IT Operations, and Security. On the other hand, reducing batch sizes leads to shorter lead times, quicker detection of errors (while they are still small), and faster recovery times. Modern software delivery practices tend to prioritize optimizing Mean Time to Restore (MTTR) over Mean Time Between Failures (MTBF). This approach emphasizes the ability to recover quickly from

failures, which are inevitable, rather than solely focusing on minimizing defects. Reducing batch sizes is achieved by shortening delivery iterations and frequently integrating code.

2.3. Limit Work in Progress

Work in progress (WIP) is one of the most significant sources of waste in software development. Unfinished tasks act as inventory that, while not directly providing value, incur costs. When not properly managed, WIP tends to accumulate, leading to multitasking and frequent task switching, which severely disrupts the flow of work. To improve efficiency, WIP needs to be limited to ensure that tasks are completed. Limiting WIP also makes it easier to identify and address issues that are preventing work from being finished. For example, if limiting WIP leads to a situation where there is no active work, it may be tempting to start a new task. However, a more effective approach would be to investigate the cause of delays in the upstream work center and resolve those issues, rather than simply adding more tasks to the queue.

2.4. Continuously Identify and Eliminate Waste.

As we have already stated above, the focus should always be on identifying and addressing a single constraint at a time. Once the constraint is identified, continuous improvement efforts should be directed solely toward it. Improving areas outside of the current constraint is wasteful at that moment. Common sources of constraints in software development include unclear requirements, disruptions in environment provisioning, deployment challenges, delays in test execution, and rigid or outdated architecture. Additionally, it's important to continually eliminate waste from the process, starting with minimizing rework by preventing defects from flowing downstream to subsequent work centers.

In all the cases mentioned, it became clear that while organizational structure plays a role, culture is the key driver of success. Culture defines how people act and react, especially during challenging times. Ownership of the product and a mindset focused on continuous improvement are critical success factors for any team. Teams that embraced ownership, worked cohesively, continuously improved, supported each other, and acted beyond individual competencies or preferences, were ultimately the ones that successfully delivered high-quality results.

2.5. Continuous Delivery

Technical practices of Flow, implemented through the Continuous Delivery pipeline [5], enable an integrated and automated flow of work from the developers' keyboards to the production environment. These practices establish a repeatable and reliable process for software development teams to continually check in code changes to the version control system, perform automated tests, and deploy to production.

These practices can be divided into five key areas:

- Version Control Practices
- Continuous Integration
- Test Automation
- Infrastructure Automation
- Automated Deployment

All source code changes-whether related to the application, tests, configurations, data, or infrastructure-are checked into a single Version Control System (VCS). These changes are made directly on the master branch (in a practice known as trunk-based development) or through short-lived feature branches. Binaries are stored in an artifact repository but can also be recreated from the source code if necessary. Techniques like feature toggles allow code integration without impacting existing functionalities.

Each check-in potentially triggers a Continuous Integration (CI) build. The integration is automatically verified through a build process and by executing unit and integration tests to catch component and integration errors as quickly as possible. This reduces the number of integration issues

and enables faster development. If the build is broken, all ongoing work halts until the issue is resolved and the build is made releasable again.

Infrastructure Automation enables the automated, on-demand provisioning and configuration of hosting environments. Using modern "Infrastructure as Code" tools such as Puppet or Chef, along with public or private cloud and container technologies, we can rapidly create environments from the code, run tests, and decommission the environments once tests are complete. This approach also allows us to scale up the number of testing environments we need, reducing resource dependencies. Software development teams can provision production-like environments early on and continuously ensure that the application runs reliably. By using Infrastructure as Code, all environment changes are driven by the code, minimizing the chance of manual errors. Faulty environments can be quickly decommissioned and replaced, as it is easier to rebuild than to repair.

Automated Deployment covers the "last mile" of the deployment pipeline. This process can be fully or semi-automated. Full automation is typically applied in testing environments, while in production environments, additional control is often maintained, such as manual exploratory or smoke testing before automated deployment. The deployment process must be automated and self-service, enabling any team member to deploy an application to any environment reliably, without fear of making mistakes. It is essential to distinguish between deployments (installing a specific version of software into a given environment) and releases (exposing new functionality to customers). Deployments should be easy, repeatable, and stress-free, while releases require a well-thought-out strategy for exposing new features to customers. There are two main release strategies: environment-based and application-based.

In the environment-based release strategy, different versions of an application exist in different environments. The release occurs when the desired environment is made available to all users or a select group (e.g., blue-green releases or canary releases). In the application-based strategy, the release is embedded within the application and configuration code itself, and the new functionality is made available through the appropriate code changes, often using techniques like dark launching or feature toggles.

However, tooling alone is not the solution for Continuous Delivery. While tools are essential for creating an integrated and automated workflow, success in Continuous Delivery relies on a culture of ownership and collaboration. Development, Test, and IT Operations teams must work together as one cohesive delivery team from the very beginning. This unified approach, supported by the right tools and culture, enables the fast and smooth flow of work from business ideas to successful products used by customers.

Based on our experience, software and system architecture choices very much determine our ability to set continuous delivery in practice. Systems should be architected for testability, deployment ability, and monitoring ability, and architecture itself should be able to evolve based on the needs of the system. Modular, loosely coupled, well-encapsulated architectures, with well-defined interfaces exposed through APIs, have low architectural entropy and enable a much higher degree of adaptability and agility in general. Legacy, monolithic systems require much more effort to reach a satisfactory level of automation, but patterns like strangler application patterns or branching by abstraction enable incrementally evolving a whole system, step by step leading to an entirely new one. These patterns allow the evolutionary design of the application architecture while allowing everybody to work on the same source code.

3. FEEDBACK

The goal of DevOps practices of Feedback is to enable fast and constant feedback loops, to create a more reliable, secure, and resilient system. Our continuous care should be to detect problems while they are still small and easy to fix, to find them before they are visible to our customers, and to prevent major system outages. In the end, we need to learn from problems and embed these learnings into future work.

We can do this only by creating, amplifying, and shortening feedback loops so that we see the problems as they occur. Failures are inevitable in any complex system, so we must design a safe system of work, where work is done without fear, with confidence that errors will be detected quickly, long before serious consequences.

Identifying problems as they occur enables us to validate or invalidate possible causes quickly and to be faster and therefore cheaper in finding and fixing the issues.

3.1. Solving Problems as They Occur

In the case when the issues occur, whether these are production issues or broken builds during continuous integration, we should team up to solve the problem quickly. Like in manufacturing, with the Andon cord of the Toyota Production System, the production line should be stopped until the problem is solved. Upon solution, we should share learnings throughout the organization to further improve the processes. This approach, compared to fixing when we have time, prevents the problem, as a source of waste, going to a downstream work center, where the cost and effort to repair it would be significantly higher. Moreover, it prevents the accumulation of technical debt.

Stopping all other work and especially preventing the start of new work before the problem is solved, ensures that new errors are not introduced into the system until current ones are resolved. The work center could potentially have the same problem in the next operation if we do not address it right away. The issues are often a result of specific circumstances in the complex systems, and, if not treated immediately, it is likely to be impossible to reconstruct particular events that were leading to them and therefore much more difficult to investigate the cause and find the solution. Good version control commenting practices and change logs prove to be efficient instruments for tracing the source of problems.

3.2. Team Ownership

We reinforce the feedback loop if Development teams participate in the support activities together with IT Operations and others. Support activities also include standby support during out of office hours. The risk of being woken up in the middle of the night makes everybody being much more diligent. In such cases, not only do developers better understand the hardships of downstream work centers, but also more quality gets built in. At many high-performing organizations with a functional way of working, development teams are responsible for running their services in production until the services are stable enough to be transitioned to IT Operations. With this approach, we push quality closer to the source, encourage teams to automate their processes, and above all, take ownership and responsibility for the products and services they are providing. The team should additionally use techniques of contextual inquiry, where they sit together with the end users to understand how they use the product and what are possible improvements from the usability perspective.

3.3. Peer Reviews and Change Coordination

The peer review process proved to be a useful practice of feedback that results in an increase in the quality of the code and enables knowledge sharing. Modern version control repositories enable peer review through a pull request process where each check-in gets reviewed by a predefined number of designated experts. This process requires discipline to keep review batch sizes small so that the review is easier to plan and perform. It also enables moving away from inspections and approvals which are typical for low-trust environments with command- and-control cultures. High-performing organizations rely more on peer reviews than on external approval of changes [3].

In multi-team environments, it is imperative to ensure change coordination and scheduling of changes. When the deployment batch size is more significant, change success rates go down, while the number of incidents and recovery time grow. Therefore, especially in a multi-team environment, it is essential to keep different teams working on the same tact, having them synchronized and managing the dependencies between them well. A possible approach is to have every team represented in a separate integration team that takes care of coordination, integration, and synchronization of work.

3.4. Centralized Telemetry

Technical practices of Feedback require creating means of centralized monitoring for gathering telemetry. We establish centralized monitoring with monitoring server(s), which collect different data (events, logs, and metrics) using agents running on the monitored objects (virtual machines, containers, physical equipment) or by entering data from different sources through an API. Modern automation techniques enable each machine or application can take care of parameters they want to be monitored on and make sure that either they are automatically registered or dynamically discovered by the monitoring server.

The gathered data we call telemetry and classify it into:

- Business telemetry (e.g., number of sales transactions, number of new users, number of logins)
- Application telemetry (application health/faults, transaction times, applications logs, external systems)
- Client telemetry (application errors and crashes on the client side)
- Environment telemetry (operating system, networking, database, storage, web traffic, CPU, memory)
- Deployment Pipeline telemetry (deployment frequency, deployment status, static code analysis metrics, code coverage metrics)

Telemetry helps us to pinpoint and solve problems faster, whether it is a defect in the application code, in the environment, data, or external. It enables a disciplined approach to problem-solving, where we can react based on the facts and not on the notion (e.g., restarting a server whenever we have a problem, without finding the underlying cause). Solving problems should also encompass adding new telemetry, which will enable easier discovery and prevention of such issues in the future. This process should be repeated for ever-weaker failure signals (incidents and near misses) to achieve proactive problem prevention.

The telemetry should be accessible and shared with the entire value stream. Retrieving telemetry information from the telemetry system should be self-service, through dashboards and APIs, and not by opening tickets for IT Operations. Adding business and application telemetry should be easy through code instruments and APIs. Furthermore, telemetry information should be put out on display in highly visible locations, through so-called information radiators. Having telemetry accessible and visible enables us to quickly notice the problem, structurally solve it based on the facts, and not only have a significantly better time to recover but also strengthen the relationship between Development and Operations by creating empathy and trust between upstream and downstream work centers.

3.5. Continuously Analyze and Improve Telemetry

Different graphical visualization tools can visually represent telemetry data. These tools, as well as different statistical analysis tools, can help us to analyze and use telemetry data for various purposes which range from trend analysis and predictions (e.g., the prediction for auto-scaling), outlier and anomaly detection, to smart, proactive alerting and escalation. Furthermore, these tools enable us to cross-reference different metrics to find correlations between business outcomes or application defects and application, environment, or deployment telemetry. The usage of simple statistical methods for data with normal distribution can prove to be quite useful for proactive alerting and automated actions in the case of outlier detection (detection of nodes that are different from others). More complex algorithms might be needed to analyze the data with non-normal distribution to achieve good anomaly detection or predictive analysis.

3.6. Experimenting

When we have our continuous delivery and telemetry system with relevant data in place, we become much more agile. Agility enables us to perform user research by experimenting in production and quickly validate or invalidate our business ideas. Without user research, there is a high chance (research [4] shows 2 out of 3) that our features deliver no value to our organization, while they make our code base more complex, and difficult to maintain and change. Furthermore, “the effort to build these features is made at the expense of delivering features that would deliver value” [4].

A/B testing or hypothesis-driven development techniques enable us to implement business ideas in a minimalistic form, quickly try them in production, compare the outcomes with expectations, and make informed decisions. We can do this without risk, in some cases even in a fully automated fashion, since the telemetry system and automated release techniques safeguard us from negative outcomes. A/B or split testing is a well-known marketing technique that enables the validation of an idea by comparing control and treatment specimens. When we release a new feature (treatment) to a targeted group of users (canary releasing), we can compare the telemetry data with the control specimen and decide on the validity of our business idea before further investment in its development. If the idea proves to be invalid, we can switch it off with feature flags. If the implementation is faulty, we can decide to fix it forward or roll it back.

4. CONTINUOUS LEARNING AND IMPROVEMENT

Finally, the DevOps practice of Continuous Learning and Improvement fosters a culture of ongoing learning and experimentation. It emphasizes the continuous creation of individual knowledge, which is then transformed into organizational knowledge, with the primary goal of improving the service we provide. A high-trust culture, one that eliminates fear and punishment for mistakes, is essential for fostering continuous improvement. Such a culture enables the development of safe, resilient work systems, leading to high-quality software products that are both reliable and secure.

4.1. Enable Organizational Learning

Safe systems of work mean that when incidents occur, we look for long-term structural solutions that prevent such incidents from happening again. We do this through a disciplined, systematic approach by avoiding any blaming of the ones potentially responsible for the problem. We should also avoid creating more processes and procedures that would “prevent” such problems from happening. Blaming and bureaucracy do not prevent nor solve the problems, but they do cause fear, which results in issues usually remaining hidden until real disasters happen. We must avoid it and instead, we should define failure as an opportunity for learning and improvement.

After the solution of the incident, we conduct blameless postmortem sessions, where we discuss what led to the incident and what measures, technical and behavioral, we can take to prevent it from happening again. We use this approach for serious to less serious issues, eventually getting to near misses and being able to prevent problems before they occur. We record these sessions into our knowledge database, promote this knowledge in the organization, and embed it as broader organizational learning. Knowledge databases should be easily searchable and accessible to everyone. Furthermore, team channels or chat rooms proved to be useful platforms for sharing knowledge for resolution of incidents and broader knowledge exchange, so logs from those sessions should also be accessible. All these local discoveries should be converted into global knowledge and global improvements.

In the end, “code is the ultimate truth” so we should keep knowledge in the version control repositories accessible, not only the software code, tests, and configuration but also living documentation [5], in the form of codified specifications expressed in Gherkin test [6] cases and automated tests. Institutionalize the Improvement of Daily Work “In the absence of improvements, processes do not stay the same due to chaos and entropy processes degrade over time.”

When we avoid fixing the problems structurally, continuously patching them with workarounds, our problems and technical debt accumulate. Eventually, technical debt makes any new work quite expensive or, even worse, prevents the organization from completing any further work. Reserving time to deal with technical debt, adding new automated tests to detect boundary conditions, adding new production telemetry, identifying categories of changes that require peer review, organizing Kaizen Blitzes or Hackathons, and conducting exercises with Game-Days [7] proved to be useful instruments to continuously improve the application and level of our service.

Based on our experience some 5 – 20% of the software development team time should continuously be used to pay the technical debt. This time should be wisely used to introduce improvements and innovation. We should always use a chance for “opportunistic refactoring” and

“always leave the code a little bit better than we found it”. While Kaizen Blitz exercises enable teams to self-organize and work for a predefined amount of time on fixing any problem they find essential, Game Day testing allows them to improve the resilience, reliability, and stability of their applications by introducing instability and failure into the system. On the other hand, Hackathons enable teams to work on their innovative ideas.

In the end, whatever we do, we should value the improvement of our daily work more than the daily work itself. It is the responsibility of leaders to create conditions in which their teams can thrive. They should promote the value of continuous learning, disciplined problem-solving, calculated risk-taking, continuous questioning, and experimentation over just being careful.

5. CONCLUSION

Based on our extensive experience, we firmly believe that the DevOps approach to software delivery is a modern and holistic methodology that drives the creation of high-quality software products and enhances customer satisfaction. These practices not only call for modernization, automation, and innovation in the technological aspects of the value stream but also require a fundamental shift in the culture and behavior surrounding software development. This shift involves fostering ownership of the work, accountability for our products and services, streamlining workflows, and promoting teamwork when facing challenges. When these practices are applied in tandem, they lead to a cutting-edge software development process, resulting in superior products, engaged employees, and delighted customers.

In conclusion, DevOps represents a modern and comprehensive approach to software delivery that integrates technology, culture, and process to create high-quality, reliable software. By fostering collaboration across traditionally siloed teams-development, operations, and quality assurance-DevOps breaks down barriers and promotes continuous communication, improvement, and shared responsibility. Through the use of automation, continuous integration, infrastructure as code, and streamlined workflows, DevOps accelerates the delivery of valuable software, reduces errors, and enhances system reliability.

However, the true success of DevOps lies not only in technological innovations but also in the cultural and behavioral shifts it drives. It requires teams to take ownership of their work, embrace accountability, and continuously seek improvement. The emphasis on collaboration, experimentation, and learning creates a high-trust environment where teams are empowered to solve problems and innovate. This culture of continuous improvement results in a more adaptive, resilient software delivery process that aligns closely with business goals and customer needs.

Ultimately, DevOps transforms the way software is developed, tested, and delivered-making it faster, more reliable, and capable of meeting the demands of a rapidly changing market. By adopting a DevOps mindset and applying its practices, organizations can achieve greater efficiency, higher quality products, and more satisfied customers, while fostering a workplace culture that encourages innovation, learning, and growth.

6. REFERENCES

- [1] Leite, L.; Rocha, C.; Kon, F.; Milošević, D.; Meirelles, P.: *A Survey of DevOps Concepts and Challenges*, ACM Computing Surveys (CSUR), Volume 52, Issue 6, Article No.: 127, Pages 1 – 35, 2019, <https://doi.org/10.1145/3359981>
- [2] Kim, G.; Debois, P.; Willis, J.; Humble, J.; Allspaw, J.: *The DevOps Handbook: How to Create World-Class Agility, Reliability, and Security in Technology Organizations*, IT Revolution Press, 2021.
- [3] Adzic, G.: *Specification by Example: How Successful Teams Deliver the Right Software*, Manning Publications, 2011.
- [4] Humble, J.; Farley, D.: *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*, Addison-Wesley Professional, 2010.

- [5] Gormley, A.; Gormley, R.: *Test Scenario Design Models: A Key to Agile Success*, Software Quality Professional, Volume 22, Issue 2, Pages 4-13, 2020.
- [6] Richardson, T.; Lagerberg, C: *Behavior-Driven Development: A Practical Guide for TDD and BDD*, Addison-Wesley, 2014.
- [7] Van Scoy, L.; Reading, J.; Hopkins, M.; Smith, B.; Dillon, J.; Green, M.; Levi, B.: *Community Game Day: Using an End-of-Life Conversation Game to Encourage Advance Care Planning*, Journal of Pain and Symptom Management 54(5), 2017, DOI:10.1016/j.jpainsymman.2017.07.034