Practical Experience in DevOps Implementation

Liliia Bodnar[1], Mykola Bodnar[2], Kateryna Shulakova[3,4], Oksana Vasylenko[4], Roman Tsarov[3], and Eduard Siemens[4]

[1]South Ukrainian National Pedagogical University, Staroportofrankyvska Str. 26, 65020 Odesa, Ukraine
[2] LLC B&B Solutions, Dukivska Str. 5, 65000 Odesa, Ukraine
[3]State University of Intelligent Technologies and Telecommunications, Kuznechna Str. 1, 65023 Odesa, Ukraine
[4]Anhalt University of Applied Sciences, Bernburger Str. 57, 06366 Köthen, Germany
bodnarl79@pdpu.edu.ua, katejojo29@gmail.com, oksana.vasylenko@hs-anhalt.de, rcarev@gmail.com, eduard.siemens@hs-anhalt.de

Keywords: DevOps, CI/CD, Commercial Experience, Educational Resources, IT Professionals.

Abstract: This scientific article serves as a comprehensive exploration of the amalgamated technical expertise derived from hands-on involvement in commercial projects. Its primary objective is to harmonize the training of upcoming DevOps engineers with the dynamic demands of the real-world market. The focus of our investigation lies in the intricate stages of continuous integration (CI) processes, meticulously shaped by the practical experiences of seasoned DevOps engineers. Delving into these stages, we unveil invaluable insights that extend their applicability to benefit both IT instructors and active DevOps practitioners. The elucidation of these stages is firmly grounded in the pragmatic utilization of a diverse array of DevOps practices. By drawing from the multifaceted experiences encountered in real-world scenarios, our article presents a nuanced understanding of the challenges and triumphs that permeate the field of DevOps. This nuanced approach not only enhances the theoretical knowledge imparted to future engineers but also provides a reservoir of practical wisdom that can be readily applied in professional settings. That is, our undertaking encapsulates the core objective of bridging the disjunction between theoretical knowledge and practical application within the domain of DevOps.

1 INTRODUCTION

In recent years, DevOps has emerged as one of the most popular methodologies that bridges the gap between software development and operations. The DevOps methodology involves the collaboration of development teams and operational specialists to create software with the goal of swiftly deploying IT solutions to the market. Research [1] has determined that the primary objective of DevOps is efficient implementation of new features and reduction of time frames from development to production. These findings underscore the impact of this aspect on improving software development outcomes. However, DevOps is more than just a methodology; it is a culture of collaboration and automation that contributes to reduced burnout and enhanced productivity [2]. Research involving over 36,000 professionals worldwide regarding DevOps practices [3] supports successful software delivery and operational efficiency. Automation, in particular, helps ensure code quality and reduces the number of defects [4]. Additionally, a systematic literature review [5, 6] emphasizes the importance of integrating security into the DevOps process to ensure secure development and effective product delivery. This highlights how modern enterprises in the digital transformation era face new challenges, compelling IT teams to adapt to emerging technologies and embrace customer-centric approaches. Consequently, traditional software development models like the Waterfall Model and Spiral Model have given way to agile practices such as Kanban and Scrum [7]. DevOps has become the most relevant approach in software development, leading to an increased demand for individual IT solutions. Rapid software deployment is no longer just a competitive advantage; it’s a survival requirement in today’s market. The demand for DevOps specialists has steadily grown over the past few years, and this trend is expected to continue. Alongside demand, the responsibilities of DevOps engineers have expanded. Today, they are not only responsible for process automation and infrastructure configuration but also collaborate with developers, testers, and system administrators. DevOps engineers are highly sought after in the job market, yet existing educational programs don't always provide them with the necessary up-to-date knowledge and skills.

In this study, we explore key technical competencies and practical experience that make DevOps engineers marketable. Our research aims to provide recommendations for preparing DevOps professionals who meet modern requirements. By analyzing current trends and drawing from the practical experience of DevOps engineers, we identify the essential components that contribute to a successful CI process. Understanding these stages and their nuances empowers aspiring DevOps specialists to navigate the complexities of CI effectively.

2 ANALYZE THE STATE OF THE PROBLEM

As a result of our analysis, we've identified key components that make DevOps engineers highly sought after in the job market (Figure 1), considering both recent trends [8] and practical insights [9-11]:

- Deep understanding of Software Development Life Cycle (SDLC). A DevOps professional must have a clear understanding of all SDLC stages, from planning and design to testing and deployment. This understanding enables effective collaboration with development, QA, and operations teams to ensure a smooth and efficient software development and release process.
- Proficiency in Infrastructure as Code (IaC) principles. DevOps engineers should be able to describe and manage infrastructure using code, leveraging tools like Ansible, Puppet, Chef, or Terraform. IaC allows for automated infrastructure creation and configuration, making it more efficient, reliable, and scalable.
- Thorough configuration management knowledge. DevOps experts need to grasp configuration management principles and utilize tools such as CMDB, Puppet, Chef, or Ansible for centralized configuration management of infrastructure and software. This ensures consistent and coordinated configuration across all systems, reducing the risk of errors and improving reliability.
- Confident mastery of Continuous Integration and Continuous Delivery/Deployment (CI/CD). DevOps professionals should be adept at setting up CI/CD processes, which automate software build, testing, and release. CI/CD significantly reduces software release time and enhances its quality.

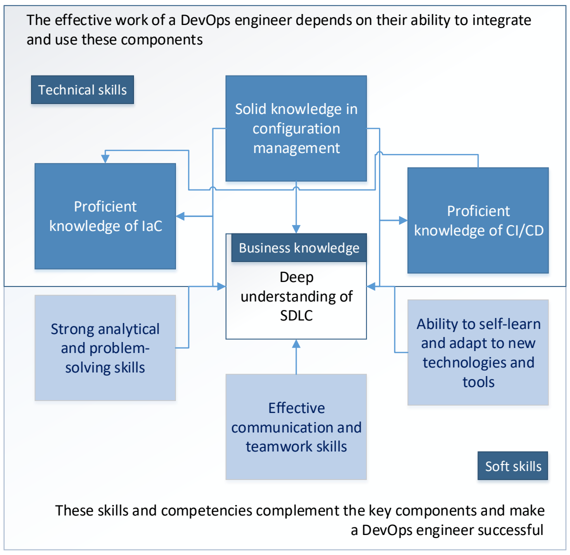
The effective work of a DevOps engineer depends on their ability to integrate and use these components
Technical skills
Solid knowledge in configuration management
Proficient knowledge of IaC
Proficient knowledge of CI/CD
Business knowledge
Deep understanding of SDLC
Strong analytical and problem-solving skills
Ability to self-learn and adapt to new technologies and tools
Effective communication and teamwork skills
Soft skills
These skills and competencies complement the key components and make a DevOps engineer successful

Figure 1: Components of a successful DevOps engineer.

In addition to these key components, a successful DevOps engineer should also possess [12]:

- Strong analytical and problem-solving skills.
- Effective communication and teamwork abilities.
- Adaptability to learn and embrace new technologies and tools.

Mastering these knowledge areas, skills, and competencies makes a DevOps professional a valuable asset for any organization striving to successfully implement DevOps practices.

Currently, information about DevOps specialists is scattered across the internet, and there is no single guide that describes best practices and offers solutions for effectively combining them to achieve optimal results. In light of this, we have decided to create recommendations that allow us to consolidate our experience and catalogs, showcasing a systematic approach to problem-solving. From Figure 1, it is evident that describing all components is a rather labor-intensive process. Therefore, within the scope of this article, we have developed an approach for one of the CI/CD (Continuous Integration/Continuous Delivery/Continuous Deployment) components: continuous deployment. CI/CD represents the practice of automatically delivering and deploying applications to the production environment after successfully passing all CI stages (Figure 2). One of its distinguishing features is that continuous delivery deploys code continuously using manual triggers, while continuous deployment automates deployment without human intervention. While continuous delivery and deployment processes do not require special approaches, continuous integration involves developing a more detailed strategy.

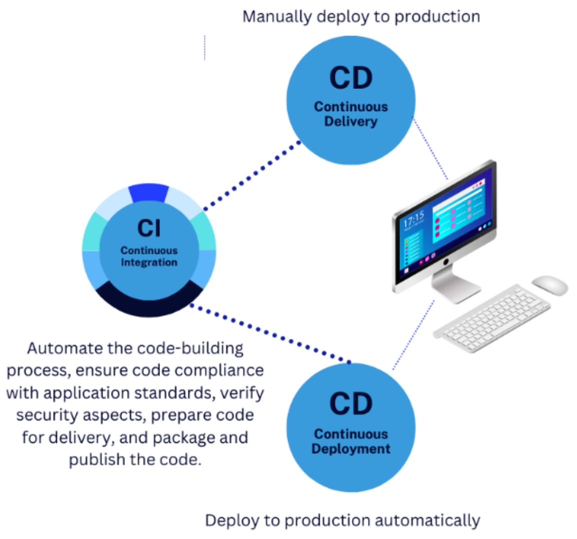
Manually deploy to production
CD
Continuous Delivery
CI
Continuous Integration
Automate the code-building process, ensure code compliance with application standards, verify security aspects, prepare code for delivery, and package and publish the code.
CD
Continuous Deployment
Deploy to production automatically

Figure 2: CI/CD basic workflow.

The core idea of CI is to automate the code build process, verify code compliance with application standards, assess security factors, prepare code for delivery, package it, and publish it. Imagine assembling a car. Each developer creates individual components, such as the engine or doors. CI acts as the conveyor belt that brings these components together. It automatically "transfers" them from one stage to another as soon as they are ready. This approach allows for early error detection and correction during development, ensuring swift and reliable delivery of new features. By reducing the time between developing new functionality and putting it in users' hands, CI also minimizes the risk of deployment issues.

Analyzing works [13-14], we can identify key aspects of CI:

- Automated Builds. Developers commit their code changes frequently, triggering automated builds. These builds compile the code, run code quality tests, and generate deployable artifacts.
- Unit Testing. CI systems execute unit tests to validate individual components of the codebase. Early detection of defects helps maintain code quality.
- Quality Assurance Testing. CI ensures that code changes integrate seamlessly with existing code.
- Artifact Generation. The CI process produces deployable artifacts (e.g., binaries, containers) ready for deployment.
- Feedback Loop. Developers receive immediate feedback about their code changes, allowing them to address issues promptly.

3 DEVELOPMENT OF A SYSTEMATIC METHOD OF SOLUTION

Despite the fact that the CI workflow shares commonalities with Agile methodologies, it has its specific elements. While there isn't a universal algorithm for CI processes, we can identify a general structure for the workflow. Figure 3 illustrates the six stages involved:

1) Source code build stage. This initial step involves compiling and building the source code.
2) Self-test stage. Automated tests are executed to validate the functionality of the code. These tests cover unit testing, code quality testing, and other quality checks. It ensures that the code adheres to coding standards and conventions.
3) Artifact preparation stage. Here, the code is packaged into deployable artifacts. These artifacts can be binaries, containers, or other forms suitable for deployment.
4) Quality Assurance (QA) stage. QA processes, including manual testing and user acceptance testing, ensure that the artifacts meet the desired quality standards.
5) Security checking stage. Security scans and vulnerability assessments are performed to identify and address any security risks.
6) Artifact publishing stage. Finally, the validated and secure artifacts are published to the appropriate environment (e.g., staging or production).

CI is a cornerstone of successful DevOps practices, enabling faster development cycles, improved collaboration, and higher-quality software delivery.

Let's analyze each stage of CI and formulate a systematic approach for each stage.

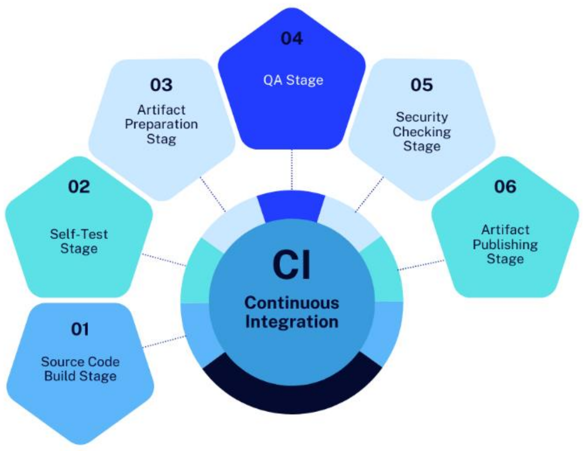

Figure 3: The general structure of the workflow CI.

3.1 Source Code Build Stage

In software development pipelines, the build stage is crucial for transforming source code into executable artifacts. While developers often handle this stage, a proficient DevOps Engineer should also understand the process thoroughly. After all, the DevOps Engineer will be responsible for configuring and maintaining this critical step in the pipeline.

The primary challenge lies in the fact that each programming language has its own specific build tools. For instance:

- .NET: Uses the dotnet builder.
- Java: Relies on the Maven build system.
- React (JavaScript): Requires its own set of build tools.

However, there are broader solutions that can simplify the process by taking a more universal approach. Examples include Gradle and NPM. These tools cater to specific domains: NPM is excellent for building web user interface (UI) components based on JavaScript frameworks like Angular and React, while Gradle serves as a versatile solution for Java-related applications (including Kotlin, Groovy, and Android). Unfortunately, neither Gradle nor NPM is suitable for building .NET applications.

So, how does a DevOps Engineer navigate this challenge? Let's explore some strategies:

A) Developer Responsibility Approach:
1) Some DevOps teams consider the build stage purely a developer's responsibility. They allow developers to configure the build process locally, assuming it will work seamlessly in a multiuser environment.
2) However, this approach has limitations. Developers often tailor their local setups to specific use cases, which may not translate well to a shared pipeline. This can lead to instability and friction between developers and DevOps engineers.

B) Universal Builders Approach:
1) A more pragmatic solution involves using universal builders. DevOps Engineers create general build configurations based on these tools.
2) Developers then describe their specific build processes within these universal solutions. For example:
    - The DevOps Engineer ensures the correct execution of the builder (e.g., Gradle or NPM).
    - Developers focus on defining the build steps and dependencies within the chosen tool.
3) This compromise streamlines the DevOps part, making it more predictable, easier to manage, and stable.
4) It also clarifies the division of responsibilities: DevOps handles the orchestration, while developers focus on the specifics of their application builds.

In summary, a collaborative approach that leverages universal builders strikes a balance between stability, flexibility, and clear responsibilities in the build process. DevOps Engineers play a crucial role in ensuring smooth execution, while developers contribute their domain-specific knowledge to the configuration.

3.2 Self-Test Stage

In software development pipelines, the build stage plays a critical role. It serves as the last checkpoint where developers can review their code before it proceeds to subsequent stages. These subsequent stages are beyond their control, as they fall outside the developers' direct responsibility.

The build stage is an ideal place to execute unit tests, lint checks, and other validations. These tests are typically short and efficient, minimizing the time spent in this phase. However, the main challenge lies in the human factor. Writing comprehensive unit tests requires disciplined processes and a strong commitment, which developers may not always prioritize. Common reasons include a lack of clear code standards and insufficient requirements for adhering to those standards.

While this leniency might be acceptable in startup environments, it becomes unacceptable for stable projects. In mature projects, where the codebase is well-established, any deviation from coding standards can lead to unpredictable outcomes.

Another challenge in this stage pertains to reporting and handling failures. CI pipelines often run in the background, lacking the UI where developers can directly view test results. Additionally, when multiple pipeline runs occur concurrently, the previous results may get overwritten by the current run. To address this, we rely on middleware tools such as SonarQube or SonarCloud. These tools integrate seamlessly with CI platforms and serve as repositories for test reports.

However, merely collecting reports isn't sufficient. We also need mechanisms to halt the pipeline when tests fail. This functionality is typically implemented within the chosen middleware.

Here are two trade-offs to consider for this step:

1) Base Code Formation. If the base code is still evolving, and the development team lacks strict code standards, you might choose to omit this stage initially. However, it's essential to add it as the team matures and establishes clearer guidelines.
2) Middleware Integration. For projects with well-defined standards, integrate middleware tools like SonarQube or SonarCloud into your CI process. These tools not only store reports but also provide stop triggers to prevent pipeline progression upon test failures.

In summary, striking a balance between flexibility and stability ensures that the DevOps role remains effective in managing the build process. Developers contribute their expertise within the universal framework, while DevOps Engineers orchestrate the overall pipeline execution.

3.3 Artifact Preparation Stage

The stage is optional, but in line with the modern trend of creating platforms based on microservices, the importance of stages is growing. These stages serve as containers for microservices, where each microservice can reside. The use of Docker containers for this purpose is particularly advantageous. By preparing a standardized environment for each service within a container, you can ensure that the application will function consistently from scratch.

Additionally, there are significant benefits for DevOps engineers. Working with Docker containers, rather than independent applications, allows for the unification and automation of the delivery and runtime processes. However, a key challenge arises during the CI/CD pipeline: how to package an application into a Docker image. When using a virtual machine (VM) as a build server, installing the Docker daemon and running the docker build command with a prepared Dockerfile is straightforward. But what if your CI/CD pipeline relies on Docker containers as build servers, as is the case with GitHub Actions, GitLab CI/CD, or Jenkins with Kubernetes plugins? In such scenarios, you must address several issues, including providing the Docker daemon inside a Docker container with access to the Linux socket.

To tackle this problem, the Google team developed an application called 'Kaniko.' Kaniko can create a Docker image based on a Dockerfile without requiring the Docker daemon. Instead, it treats the Docker image as a simple archive with a specific internal structure. Kaniko parses the Dockerfile based on the defined run points, creates the archive package (Docker image), and pushes it to any Docker registry. Kaniko strikes an ideal compromise for this stage of the process.

3.4 QA Stage

The stage is ideal for running short-lived quality assurance tests. Specifically, we focus on tests like smoke tests, rather than larger ones such as performance or integration tests.

However, the main challenge with using a stage is that some teams treat the CI tool as the primary environment for running all tests conducted by QA engineers. This approach is fundamentally flawed because CI tools serve multiple purposes. While one of their functions is to execute scripts triggered by events, this functionality is primarily suited for running automatic test scripts.

Unfortunately, CI tools cannot fully provide the range of QA functionality required. For instance, they may lack proper mechanisms for collecting and viewing test reports or adjusting the runtime environment to consistently run the same set of tests automatically and manually.

To strike a balance, consider splitting your test suite into two groups:

1) Short-lived tests. These can be seamlessly integrated into CI pipelines.
2) Independent tests. These should be executed using dedicated QA tools like TestKube or similar alternatives.

When integrating QA tools with CI/CD pipelines, choose wisely. For instance:

- CI integration makes sense for controlling and automatically running short-lived, CI-compatible tests.

- CD integration is more suitable for other types of tests, as it accommodates specific triggers that may only be available during CD pipelines.

Selecting the right integration ensures correct automation and flexibility.

3.5 Security Checking Stage

The stage is optional, but in recent times, its importance has been growing. First and foremost, security considerations need to be defined. The field of security testing is vast, with multiple levels, each having its own penetration goals and scenarios. While it's not feasible to run the entire spectrum of security tests during the continuous integration (CI) pipeline due to various constraints (such as context or the inability to accurately assess application status), certain types of tests can indeed be executed within this pipeline.

Specifically, let's focus on vulnerability tests for applications. The goal of these tests is to identify security vulnerabilities in the code before it reaches the production stage. Scanning code for vulnerabilities is a challenging task that demands extensive knowledge not only about potential weaknesses in your application but also in the dependencies it relies upon.

The primary challenge with this stage lies in the lack of a universally understood approach. Engineers often seek the simplest solution—one that works out of the box without requiring deep dives into complex configurations. Unfortunately, such one-size-fits-all solutions don't exist.

A reasonable compromise is to explore the various CI plugins available in the market. By delving into these options, you can select the most suitable solution for your project, covering at least 50% of your security needs.

In different projects with varying requirements, we've encountered solutions like BlackDuck, which is intricate to understand and may not always provide stable results. On the other hand, X-Ray scanning, integrated into JFrog Artifactory, offers a simpler experience. JFrog takes care of X-Ray's functionality, although it might be less powerful in terms of where scanning results are visible.

Given the abundance of solutions in the global market, it's essential to find the one that best fits your project's specific context.

3.6 Artifact Publishing Stage

The stage is the finishing stage. The appearance and functionality of this stage are determined by the programming language used. Each language has its own build process and produces specific types of artifacts. An artifact is a compiled and packaged file ready for deployment within the application process. For instance, in Java, artifacts could be JAR files, application packages, WAR files, or even Docker image files.

However, a critical challenge arises: how to manage these artifacts effectively. Each type of artifact requires storage in a specific registry. While some artifacts (like JAR, WAR, and application files) can share a common repository (such as Maven or Gradle), others (like .NET artifacts) necessitate separate repositories (e.g., NuGet for .NET or npm for JavaScript).

To address this, consider using universal storage solutions like Sonatype Nexus Repository or JFrog Artifactory. These platforms offer a wide range of repository types, allowing you to create storage tailored to your project's needs. By integrating just one solution with your CI/CD tools, you simplify artifact management.

Beyond this primary compromise, there are additional tips to reduce repository complexity. For example:

- Microservices and Docker. If you're building a microservices platform based on Docker containers, you don't need to store individual application artifacts. Instead, focus on storing Docker images containing the application artifacts. This approach streamlines repositories within a single Docker registry.
- Helm Chart Repositories. Helm can utilize a Docker registry as a Helm chart repository. While this approach has limitations, it's worth considering if needed.

In summary, aim for a repository setup that aligns with your project's requirements while minimizing unnecessary complexity. However, native repositories offer features that can be highly beneficial for the CI/CD process. If you have the opportunity to use them, it's advisable to do so.

4 CONCLUSIONS

In our research, we delved into the key components that make DevOps engineers highly sought-after in the job market. Despite widespread adoption, there remains a scarcity of comprehensive resources that consolidate practical knowledge for aspiring DevOps professionals. Our study aims to bridge this gap by offering a holistic view of DevOps practices based on real-world commercial experience.

Our primary focus was on CI. We developed recommendations outlining six approaches for implementing CI. Additionally, we provided examples and approaches for applying these practices in real-world projects.

By considering these critical factors, IT companies can strengthen their position in the international market by hiring skilled DevOps engineers, thereby creating a dynamic environment that encourages growth, innovation, and sustainable global collaboration.

Our research contributes to the existing body of knowledge by providing a comprehensive overview of DevOps practices and their associated benefits.

These findings can be leveraged by:

- Universities: To develop and implement effective DevOps education programs.
- IT companies: To recruit and train skilled DevOps engineers.
- DevOps engineers: To enhance their skills and knowledge.

Future research directions include:

- Investigating other key components of DevOps, such as infrastructure as code (IaC) and containerization.
- Developing case studies on the successful implementation of DevOps practices in different industries.

We believe that this research will help to advance the field of DevOps and make a contribution to the IT industry.

ACKNOWLEDGMENTS

We acknowledge support by the German Research Foundation (Deutsche Forschungsgemeinschaft DFG) - and the Open Access Publishing Fund of Anhalt University of Applied Sciences.

REFERENCES

[1] L.E. Lwakatare, P. Kuvaja, and M. Oivo, "Relationship of DevOps to Agile, Lean and Continuous Deployment," in P. Abrahamsson et al. (eds), Product-Focused Software Process Improvement. PROFES 2016. Lecture Notes in Computer Science, vol 10027, pp 399-415, Springer, Cham, 2016, [Online]. Available: https://doi.org/10.1007/978-3-319-49094-6_27.

[2] M. Sánchez-Gordón and R. Colomo-Palacios, "Characterizing DevOps Culture: A Systematic Literature Review," in I. Stamelos et al. (eds), Software Process Improvement and Capability Determination. SPICE 2018. Communications in Computer and Information Science, vol. 918, pp. 3-15, Springer, Cham, 2018, [Online]. Available: https://doi.org/10.1007/978-3-030-00623-5_1.

[3] "2023 State of DevOps Report," [Online]. Available: https://cloud.google.com/devops/state-of-devops, [Accessed: 28 Dec 2023].

[4] D. Belcher, "Three Trends That Will Transform DevOps in 2023," 2023, [Online]. Available: https://devops.com/three-trends-that-will-transform-devops-in-2023, [Accessed: 28 Dec 2023].

[5] M. Gasparaite, K. Naudziunaite, and S. Ragaisis, "Systematic Literature Review of DevOps Models," in M. Shepperd et al. (eds), Quality of Information and Communications Technology. QUATIC 2020. Communications in Computer and Information Science, vol. 1266, pp. 184-198, Springer, Cham, 2020, [Online]. Available: https://doi.org/10.1007/978-3-030-58793-2_15.

[6] T. Leppanen, A. Honkaranta, and A. Costin, "Trends for the DevOps Security. A Systematic Literature Review," in B. Shishkov (eds), Business Modeling and Software Design. BMSD 2022. Lecture Notes in Business Information Processing, vol. 453, pp. 200–217, Springer, Cham, 2022, [Online]. Available: https://doi.org/10.1007/978-3-031-11510-3_12.

[7] M. Rütz, "DevOps: A Systematic Literature Review," Twenty-Seventh European Conference on Information Systems (ECIS2019), Stockholm-Uppsala, Sweden, 2019, pp. 1-16, [Online]. Available: https://www.researchgate.net/publication/335243102_DEVOPS_A_SYSTEMATIC_LITERATURE_REVIEW.

[8] M. Modi, "The Future of DevOps: 2024 and Beyond," 2023, [Online]. Available: https://www.knowledgehut.com/blog/devops/future-of-devops, [Accessed: 29 Dec 2023].

[9] "DevOps practice in Ukraine, 2024," [Online]. Available: https://bandbsolution.com.ua/en, [Accessed: 11 Jan 2024].

[10] "DevOps practice in Germany, 2024," [Online]. Available: https://www.fme.de/dienstleistungen/technology-services, [Accessed: 10 Jan 2024].

[11] "DevOps practice in USA, 2024," [Online]. Available: https://www.fme-us.com. [Accessed: 10 Jan 2024].

[12] G.B. Ghantous and A. Gill, "DevOps: Concepts, Practices, Tools, Benefits and Challenges," PACIS 2017 Proceedings, pp. 1-13, 2017, [Online]. Available: http://aisel.aisnet.org/pacis2017/96.

[13] L. Gryzun, V. Pikalova, and L. Bodnar, "Issues of Selecting an Instrument of Continuous Integration for Software Automated Testing," International Scientific and Practical Forum «Digital Reality» 2023 [«FDR2023»]: Cybersecurity and information technologies in the hybrid wars conditions, [Accessed: 2 Dec 2023].

[14] "DevOps Tools - GitLab CI/CD," [Online]. Available: https://digitize01.com/devops-tools-gitlab-ci-cd, [Accessed: 29 Dec 2023].