Tampere University

Erfan Niketeghad

A STUDY ON CI/CD AND DEPLOYMENT STRATEGIES

A Constructive Study on DevOps Transformation in Microservice-based IoT SaaS Environments

Master's Thesis
Faculty of Information Technology and Communication Sciences
Examiners: University Lecturer Terhi Kilarmo
Software Design Engineer Juha Ylikoski
June 2025

ABSTRACT

Erfan Niketeghad: A Study on CI/CD and Deployment strategies
Master's Thesis
Tampere University
Software, Web and Cloud
June 2025

Internet of Things (IoT) and Software as a Service (SaaS) companies face unique challenges in implementing effective DevOps practices due to their complex microservice architectures and single-tenant deployment models. This thesis addresses these challenges through a constructive research approach, implementing and evaluating DevOps transformation practices at Treon, a company specializing in microservice-based IoT SaaS solutions. The research focused on three primary objectives: implementing a continuous deployment solution using GitOps principles, evaluating deployment strategies suitable for single-tenant IoT architectures, and standardizing continuous integration pipelines across multiple repositories. The study employed ArgoCD as the GitOps tool to automate deployment processes and enhance configuration management, systematically evaluated canary, blue-green, and rolling update deployment strategies, and enhanced existing Jenkins shared libraries to reduce code duplication and improve pipeline reliability.

Key findings demonstrate that GitOps implementation significantly improved deployment reliability, visibility, and operational workflows while reducing manual intervention and human error. The evaluation revealed that namespace-level rolling updates proved most suitable for single-tenant IoT architectures. Canary deployments were incompatible with isolated client environments, while blue-green deployments presented prohibitive resource overhead. CI pipeline standardization through enhanced shared libraries successfully reduced code duplication and improved maintainability across the microservice architecture.

The implementation resulted in observable improvements in team workflows, including enhanced deployment reliability, automatic configuration drift detection, improved team accessibility to deployment processes, and streamlined operational procedures. The research contributes practical solutions for IoT DevOps transformation and theoretical insights into adapting standard DevOps practices for single-tenant IoT SaaS environments, providing actionable guidance for companies facing similar architectural constraints.

Keywords: DevOps, CI/CD, GitOps, ArgoCD, Deployment Strategies, Shared Libraries, IoT, SaaS, Microservices

USE OF ARTIFICIAL INTELLIGENCE IN THIS WORK

Artificial intelligence (AI) has been used in generating this work:

☐ No
☒ Yes

I hereby declare that the following AI-based application was used as a supportive tool in the preparation of this thesis:

Application	**Version**
Claude	4.5 Sonnet

Purpose of use. Claude 4.5 Sonnet was used to proofread and refine text written by the author (grammar, wording, and academic tone), to suggest clearer formulations for some sentences, to assist in debugging small code snippets, and to help identify relevant literature during the review process.

Scope of use. AI assistance was limited to language editing and presentation. All research questions, system design and implementation, methodology, analysis, and conclusions are the author's own work. No sections, figures, tables, data, or results were written or generated autonomously by AI; all AI suggestions were critically reviewed and edited by the author before inclusion.

Acknowledgement of risks

I hereby acknowledge, that as the author of this work, I am fully responsible for the contents presented in this thesis. This includes the parts that were generated by an AI, in part or in their entirety. I therefore also acknowledge my responsibility in the case, where use of AI has resulted in ethical guidelines being breached.

PREFACE

This master's thesis was completed in collaboration with Treon, where I had the opportunity to conduct practical DevOps transformation work from January to June 2025. The project allowed me to address real-world challenges in IoT and SaaS environments while contributing to both academic knowledge and practical solutions.

I would like to express my gratitude to my supervisors, University Lecturer Terhi Kilarmo and Software Design Engineer Juha Ylikoski, for their guidance throughout this research. I am thankful to the team at Treon for providing the opportunity to implement these solutions in a production environment and for their collaboration during the implementation process. Their support and insights were invaluable to the successful completion of this work.

Tampere, 3rd June 2025

Erfan Niketeghad

CONTENTS

LIST OF FIGURES

1. INTRODUCTION

In today's rapidly evolving technological landscape, IoT and embedded systems companies are increasingly recognizing the strategic value of DevOps methodologies. These approaches enable organizations to develop, test, and deploy new features with greater efficiency while maintaining system stability and security, driving widespread adoption of DevOps practices across the industry.

Business evolution necessitates corresponding development in internal processes and workflows. Within software development environments, this transformation manifests through more frequent testing cycles, shortened feedback loops, and regular integration of code changes. These requirements have catalyzed the implementation of continuous practices throughout the development lifecycle. Continuous practices encompass methodologies where software under development undergoes automated testing and integration continuously, enabling early detection of defects and malfunctions. This approach extends naturally to deployment processes, where seamless implementation following successful integration creates a comprehensive continuous delivery or deployment pipeline.

This thesis has been developed in conjunction with practical experience at Treon, with a specific focus on CI/CD continuous practices, the implementation of effective deployment strategies, and establishing robust monitoring solutions. The research addresses Treon's organizational objective to enhance its DevOps framework, ultimately creating more reliable systems that deliver superior quality outcomes for their customer base.

While IoT and SaaS companies have made substantial progress in service deployment practices, there remain opportunities to enhance service continuity and reliability. Optimizing deployment scenarios to deliver updates without causing data loss, service disruption, or noticeable downtime requires careful consideration, especially in the increasingly complex environment of microservice-based architectures. Various deployment strategies have evolved to address these nuances, each offering specific advantages for different operational requirements.

Additionally, as organizations scale their microservice architectures, CI pipeline inefficiencies become increasingly apparent. Duplicated configuration across multiple repositories creates maintenance overhead, while inconsistent testing approaches can undermine pipeline reliability. Without standardization and centralization of common configurations, teams waste valuable time managing repetitive tasks rather than focusing on product innovation.

Suboptimal deployment practices and inefficient CI pipelines can impact service quality in several ways. End-users may occasionally experience inconsistent application behavior across devices or encounter service interruptions during updates. When deployment processes aren't fully standardized and automated, the risk of production issues increases, potentially affecting client operations. These technical considerations directly influence business outcomes, including customer satisfaction and operational efficiency.

CI/CD pipelines and strategically selected deployment strategies serve as valuable components of modern software delivery. These practices establish structured frameworks for the software development lifecycle, ensuring updates and new features undergo appropriate validation before reaching production environments.

Based on these industry challenges and the specific needs observed at Treon, this research addresses three primary questions:

First, which deployment strategies provide optimal outcomes for minimizing downtime, mitigating risk, and facilitating rapid rollbacks in a microservice-based IoT SaaS platform? This question explores the relative advantages of strategies like blue-green deployment and rolling updates within the context of single-tenant IoT service delivery.

Second, how can GitOps implementation enhance the management and reliability of customer deployments in a microservice-based IoT SaaS environment? This question examines the application of GitOps principles through ArgoCD to address the challenges of deployment automation and configuration management.

Third, how can CI pipeline standardization through shared libraries improve efficiency and reliability in multi-repository microservice architectures? This question investigates approaches to reducing duplication and inconsistency across CI pipelines in complex microservice environments.

These questions are formulated to address the practical considerations observed at Treon and to contribute to the broader understanding of DevOps practices in IoT contexts.

This thesis focuses on comprehensive DevOps transformation within Treon's innovative microservice-based IoT SaaS platform, encompassing CI/CD improvements, deployment strategy implementation, and monitoring solutions. While the findings may offer broader industry insights, the research primarily addresses technologies and use cases relevant to Treon's operational environment and customer needs.

On the CD side, the study primarily examines GitOps principles and tools (with emphasis on ArgoCD), blue-green deployment, and rolling update strategies. Other deployment methodologies are discussed comparatively but not implemented in depth. On the CI side, the research covers Jenkins pipeline standardization through shared libraries, automated version reporting implementation, testing reliability improvements, and centralized configuration management across multiple repositories.

The study also explores the implementation of a comprehensive monitoring infrastructure using Prometheus for metrics collection and Grafana for visualization, providing crucial operational visibility into the microservice architecture. This component addresses the challenges of detecting, diagnosing, and resolving issues in distributed systems, with particular attention to the unique requirements of IoT environments.

Due to the timeframe of the thesis work, long-term performance metrics and extended operational evaluations are outside the scope of this research. Additionally, as the implementation occurs in a production environment, certain experimental approaches may be limited by operational risk considerations.

The remainder of this thesis is structured as follows. Chapter 2 presents the theoretical framework, covering DevOps concepts, deployment strategies, CI/CD practices in IoT, GitOps principles, and monitoring approaches in microservice environments. Chapter 3 describes the research methodology, including the constructive research approach, research questions, and evaluation criteria. Chapter 4 presents the case study implementation at Treon, detailing the technical environment, initial state assessment, and the implementation of GitOps, deployment strategies, CI pipeline standardization, and monitoring infrastructure. Chapter 5 reports the results and findings from each implementation component. Chapter 6 discusses the implications of the findings, connecting them to broader IoT SaaS challenges, critically analyzing the approaches taken, and exploring lessons learned and future research directions. Chapter 7 concludes the thesis by summarizing the key contributions and outcomes.

2. THEORETICAL FRAMEWORK: DEVOPS IN IOT ENVIRONMENTS

This chapter establishes the theoretical foundations for the DevOps transformation implemented in this research. It covers core DevOps concepts, deployment strategies, CI/CD practices in IoT environments, GitOps principles with ArgoCD, and monitoring approaches for microservice architectures.

2.1 Definitions & Concepts

DevOps refers to the combination of development and operations practices [7], bringing together these historically distinct functions through automation of development, deployment, and monitoring processes. It involves an organizational transformation in which cross-functional teams work together to deliver features continuously, improving communication and speeding up problem resolution [6]. Rather than being simply a new methodology that some companies adopt, DevOps represents a core element of modern software engineering that organizations inevitably engage with [7].

Continuous Integration (CI) is a practice where developers frequently commit their code changes to a shared repository, triggering automated build and test processes [13]. As a fundamental agile practice, CI tools combine code contributions from multiple developers continuously, running tests to identify integration issues early [5]. The primary objective is to automate building and testing workflows, enabling rapid detection and resolution of defects while minimizing the effort required for software releases [13].

Continuous Delivery builds upon CI by automatically deploying validated code changes to test or production environments [13]. This approach enhances agile workflows by automating the entire build, test, and deployment pipeline [5]. When properly implemented, development teams maintain deployment-ready software that has successfully completed all validation steps [13]. Although manual execution remains possible, automation is the key advantage, accelerating releases and simplifying troubleshooting [5].

2.2 Deployment Strategies

Software deployment covers all processes involved in making a software system operational [2]. This multifaceted process includes various interconnected activities such as releasing, installing, activating, deactivating, updating, and removing both individual components and complete systems. The importance of deployment strategies has grown alongside the increasing complexity of modern software, which often consists of diverse component collections—including both code and data—distributed across network infrastructure [2].

Deployment strategies provide structured methods for handling this complexity, enabling teams to follow defined approaches for delivering functional software through rapid iterations while maintaining release readiness [4]. Chen's analysis of the Paddy Power case study [4] illustrates how well-designed deployment practices can deliver significant advantages: faster market delivery, higher product quality, and greater customer satisfaction. These approaches also help organizations tackle key challenges including coordinating distributed components, handling dependencies, and achieving consistent releases across diverse platforms.

2.2.1 Rolling Updates

Rolling updates represent a widely used approach for performing live software updates, in which a limited subset of instances running the previous version are incrementally upgraded to the newer version. This incremental process continues in successive waves until the entire deployment has been migrated. The strategy prevents conflicts between different versions by ensuring instances are aware of their version state, using a designated transition point to shut down the old service and bring the new one online. This approach supports high availability and system reliability by progressively deploying new features and enabling faster issue detection, while limiting the blast radius of potential failures [20, 3].

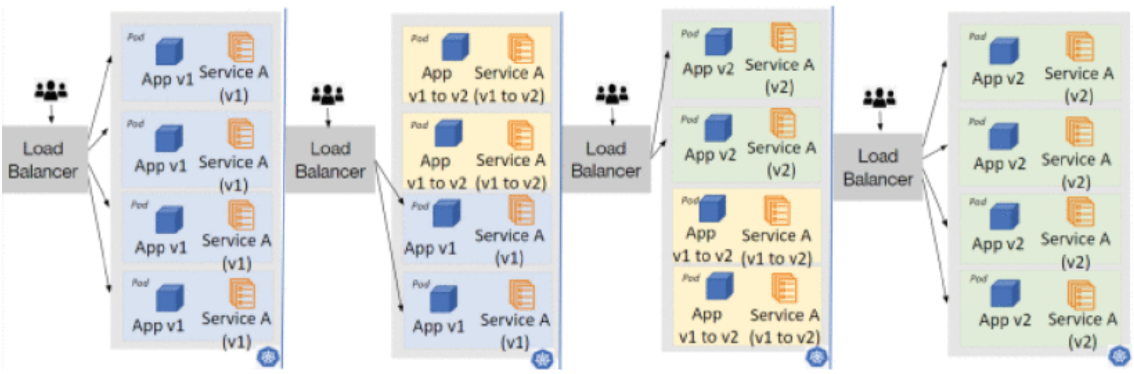

Figure 2.1. *Rolling Update. Source: [12]*

2.2.2 Blue-Green Deployment

Blue-Green deployment is a strategy that operates with two parallel, identical environments. The "Blue" environment hosts the current production version, while the "Green" environment contains the new release in a separate production-ready setup that remains inactive. Once the new version in the "Green" environment passes all validation tests, traffic is redirected from "Blue" to "Green" [21]. This method offers strong reliability and stability while minimizing downtime, with reported deployment success rates reaching 98.8% and recovery times as low as 1.5 minutes [21].

Blue-Green Deployments require two identical instances of the environment running to support existing and the new version of the code. Traffic can be routed using application load balancers, routers, or a reverse proxy [12]. Blue-Green deployment enables zero-downtime releases and facilitates rollbacks in case of failure, as the "Blue" version can stay on standby mode until the new version is stabilized [21].

While providing advantages in reliability and recovery speed, Blue-Green Deployments are costlier to achieve than other deployment techniques because they require duplicate environments [21, 12]. This approach is best suited for organizations with larger applications or monolithic architectures, and for critical systems where reliability and immediate rollback capabilities are essential [21].

Implementation Challenges in Blue-Green Deployments

While Blue-Green deployment offers significant advantages for minimizing downtime, its implementation presents several technical challenges that must be addressed. Yang et al. [23] identified key limitations in traditional Blue-Green implementations:

- **Routing Rule Updates:** Collecting IP hosts information for the new service instances is labor-intensive, particularly in dynamic auto-scaling environments.
- **Propagation Delays:** Updating routing rules on load balancers or DNS servers introduces significant time delays before changes become effective throughout the system, potentially affecting service availability.
- **Service Version Overlap:** During transition periods, both service versions may be simultaneously active for different users, creating inconsistent user experiences.

These implementation constraints have led to various Blue-Green deployment approaches, each with different performance characteristics. Yang et al. [23] proposed a set of quantitative metrics to evaluate Blue-Green deployment effectiveness:

- **Switch Time:** The total duration between activating the switch and achieving a stable state with all traffic routed to the new version.

- **Response Time:** The time between activating the switch and the first appearance of the new service version in response to incoming traffic.
- **Overlap Duration:** The time period when both service versions are simultaneously active and responding to requests.
- **Unstable Duration:** Time periods during the transition when the service becomes unavailable, resulting in errors.

These metrics provide a framework for evaluating different Blue-Green deployment implementations and optimizing the transition process for minimal disruption to users.

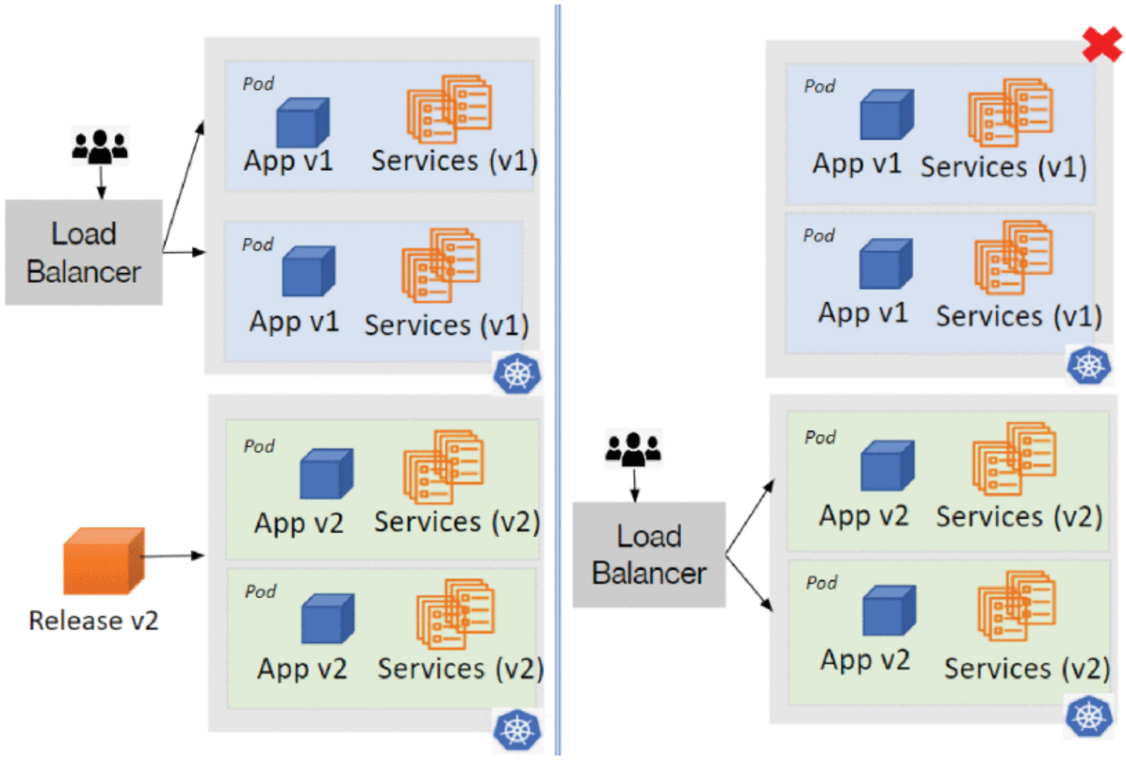

Figure 2.2. *Blue-Green Deployment. Source: [12]*

2.2.3 Canary Deployment

Canary deployment is a strategy where new changes are initially exposed to a limited group of users before being rolled out more broadly [12]. With this approach, software updates are progressively released to a small user segment—referred to as "canaries"—allowing teams to assess the deployment's effects before wider distribution [21].

The process works by designating either a specific percentage or selected users as the canary group, who receive the updated version first. Teams then monitor how the new release performs and behaves under real conditions. After confirming stability, the update can be extended to all users. Should problems emerge during this testing phase, reverting is straightforward since traffic can simply be shifted back to the previous stable release [12].

Canary deployments achieve faster time-to-market (9.7 hours compared to 12.3 hours for Blue-Green) and enable early detection of issues through staged rollout and real-time user feedback [21]. However, this approach comes with higher operational overhead, including extensive monitoring and coordination alongside multiple incremental rollouts [21].

This deployment strategy works best in large and complex applications where a sudden switch could have bigger ramifications, particularly for organizations with microservices architectures that need flexibility for small incremental changes [21, 12]. However, it may be less suitable for single-tenant environments where each client has their own namespace/instance, as canary deployment is designed for gradual user exposure across many users, which doesn't match the single-tenant model [20].

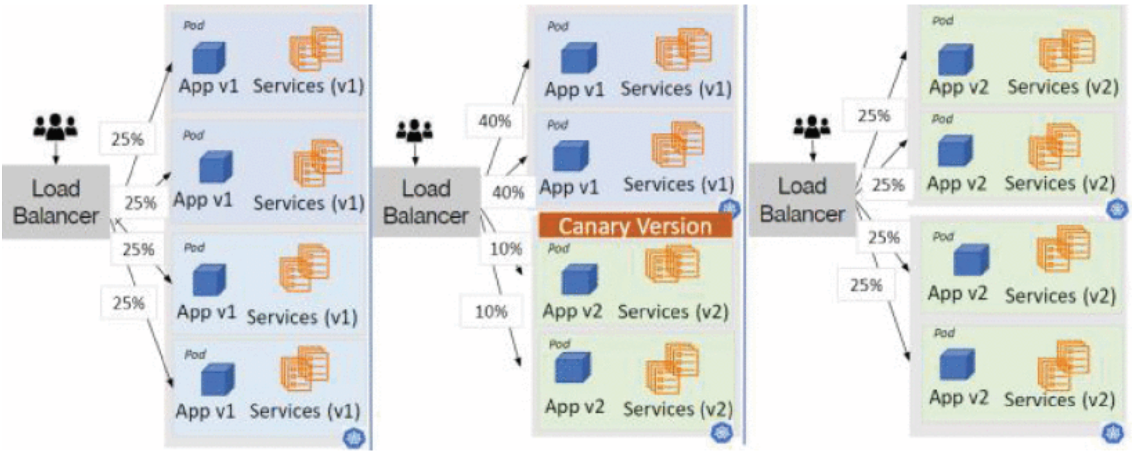

Figure 2.3. *Canary Deployment. Source: [12]*

Beyond the standard deployment strategies discussed above, research by Veith et al. [22] provides insights into how these approaches can be adapted for IoT environments with cloud-edge infrastructures. Their work demonstrates that effective deployment strategies for IoT must consider not only the application architecture but also the topology of the underlying infrastructure. For large-scale deployments spanning multiple edge sites and cloud resources, they found that strategies that incorporate infrastructure-awareness and region-based application partitioning can reduce end-to-end latency by up to 50% compared to cloud-only deployments [22]. These findings suggest that the choice of deployment strategy should be influenced by both application characteristics and infrastructure topology, which aligns with our evaluation approach for deployment strategies at Treon.

Recent empirical research by Rakshit and Banerjee [16] provides quantitative evaluations of how these deployment strategies perform in Kubernetes environments. Their comparative analysis offers valuable insights into the performance implications of each approach.

Comparative Performance of Deployment Strategies

When evaluating deployment strategies for production environments, performance metrics provide critical guidance. Rakshit and Banerjee's study [16] examined the following key metrics across Rolling Updates, Blue-Green Deployments, and Canary Releases:

- **Latency and Response Time:** Blue-Green Deployments demonstrated the most favorable latency characteristics during updates, with controlled transitions minimizing performance degradation. Rolling Updates showed higher latency during the update process, while Canary Releases exhibited moderate spikes during testing phases.
- **Throughput and Error Rates:** Blue-Green Deployments maintained the most stable throughput during transitions, while both Rolling Updates and Canary Releases experienced some throughput reduction during the deployment process. Regarding error rates, Blue-Green Deployments and Canary Releases showed advantages over Rolling Updates, particularly in isolating potential issues.
- **Application Availability:** Blue-Green Deployments provided the highest availability metrics due to their complete environment switching approach, while Rolling Updates showed slight availability dips during the update process.

Resource Considerations [16]

Resource utilization is a critical factor when selecting deployment strategies, particularly in resource-constrained environments. The empirical findings revealed:

- **CPU and Memory Utilization:** Blue-Green Deployments demonstrated more moderate resource usage patterns during transitions compared to Rolling Updates, which showed higher CPU utilization due to simultaneous operations. However, the duplicate environment requirement of Blue-Green Deployments results in overall higher resource needs throughout the deployment lifecycle.
- **Resource Efficiency:** Rolling Updates and Canary Releases achieved better resource efficiency metrics compared to Blue-Green Deployments, which require maintaining duplicate environments during transitions. This efficiency difference becomes particularly significant in large-scale deployments with numerous services.

Operational Metrics [16]

Beyond performance and resource considerations, operational factors significantly impact deployment strategy selection:

- **Recovery Time:** Rolling Updates provided the fastest rollback capabilities with quick and incremental reversion options. Blue-Green Deployments required longer recovery times due to the need to revert entire environments, while Canary Release recovery times varied based on the canary group size.

- **User Impact:** Blue-Green Deployments minimized negative user experiences during transitions through controlled switching. Canary Releases limited impact to a subset of users, while Rolling Updates potentially affected all users but with incremental exposure to changes.
- **Deployment Frequency:** Canary Releases supported the highest deployment frequency due to their targeted nature, while Blue-Green Deployments showed lower frequency capabilities due to their operational complexity.

2.3 CI/CD in IoT

Continuous Integration and Continuous Deployment (CI/CD) practices have become essential in the IoT domain, enabling automated building, testing, and deployment of IoT applications across the distributed computing continuum from mist to cloud layers [15]. CI/CD for IoT provides a systematic approach to manage the deployment and updates of applications spanning multiple computing environments with different capabilities and constraints.

While traditional CI/CD pipelines focus on application delivery to centralized cloud resources, IoT environments introduce unique challenges due to their distributed nature and resource heterogeneity. Veith et al. [22] emphasize that managing deployments across cloud-edge infrastructures requires considering the stringent latency requirements of IoT applications, particularly when data sources are distributed across large geographical areas. Their research demonstrates that naive application placement approaches can result in high end-to-end latency, making it difficult to achieve near real-time processing solely with cloud resources. This challenge is particularly relevant for microservice-based IoT SaaS environments like the one at Treon, where components may need to be strategically distributed across edge and cloud resources to optimize performance.

In IoT environments, CI/CD facilitates streamlined software delivery across heterogeneous infrastructure, allowing developers to maintain consistent functionality while adapting to the specific requirements of each deployment layer [15]. The process automates the validation and deployment of applications, ensuring that changes can be delivered reliably across the entire IoT computing spectrum, from resource-constrained edge devices to powerful cloud servers.

Modern IoT CI/CD solutions incorporate security directly into the pipeline. The GATEKEEPER platform implements OWASP-driven CI/CD tools that monitor service building and deployment while performing regular security checks throughout the integration process [8]. These tools enable organizations to maintain secure IoT deployments by detecting vulnerabilities early in the development lifecycle.

The benefits of CI/CD in IoT include faster deployment cycles, improved reliability through automated testing, dynamic service migration capabilities, and enhanced security integration [8, 15]. IoT-specific CI/CD implementations support the migration of services between computing layers without interrupting applications or losing data, which is crucial for maintaining service continuity in production environments [15].

While CI/CD brings significant advantages to IoT development, implementations must address challenges including resource constraints at the edge, network connectivity limitations, and the need for specialized security measures [8]. Despite these challenges, organizations implementing IoT-specific CI/CD solutions can achieve substantial improvements in deployment efficiency and system reliability across diverse computing environments.

CI/CD approaches are particularly well-suited for IoT applications with frequent updates, complex distributed architectures, and high reliability requirements [15]. Organizations developing healthcare IoT applications, smart city infrastructure, industrial IoT systems, and other mission-critical applications can benefit significantly from adopting appropriate CI/CD practices to manage their deployment processes [8].

2.4 GitOps & ArgoCD

GitOps is a modern DevOps approach that, paired with strong orchestration tools, provides an effective way to automate and control deployment workflows [9]. As an operational model, GitOps leverages core DevOps principles—including version control, team collaboration, compliance standards, and CI/CD automation—to streamline infrastructure and deployment management.

Weaveworks originally developed GitOps as an approach for managing Kubernetes clusters and cloud-native applications, with Git serving as the definitive source for system state [1]. Whereas DevOps emphasizes organizational culture and broader practices, GitOps offers specific techniques for implementing continuous deployment pipelines. As Beetz and Harrer note, "GitOps tries to make the operation of IT infrastructure a more developer-centric experience by employing a new CD technique" [1]. Under this model, all environment operations must be performed through Git repository contents—meaning environments are provisioned, modified, and removed exclusively via version-controlled configuration files.

2.4.1 GitOps Principles

The GitOps Working Group has established four fundamental principles that form the core of the GitOps approach [11]:

- **Declarative description**: The system's desired state is explicitly defined, abstracting away the specific steps needed to achieve that state.
- **State is versioned and immutable**: All desired state configurations are stored with full version control, maintaining a complete history of changes that cannot be altered retroactively.
- **Change approvals automation**: Approved changes to the desired state are automatically pulled and applied from the source repository.
- **Alert on differences**: The system continuously monitors the actual state and compares it against the desired state, triggering reconciliation when discrepancies are detected.

Beetz and Harrer [1] describe two primary implementation models for GitOps that offer different advantages:

Push-Based GitOps

In this model, GitOps uses two types of repositories: environment repositories containing configurations describing the current production environment, and application repositories containing source code and infrastructure-as-code configurations. The CI pipeline of application repositories updates configurations in the environment repository, which then initiates a CD pipeline that applies the infrastructure-as-code definitions and brings the environment into alignment with the target state [1].

Pull-Based GitOps

The pull-based model introduces an operator—an application deployed within the environment that regularly monitors the environment repository for updates and synchronizes those changes to the live environment. As Beetz and Harrer explain, "this essentially reverses the usual flow of information in continuous deployment pipelines" [1]. The operator can observe infrastructure state, react to deviations, and automatically restore the desired state. This approach also keeps credentials inside the environment rather than exposing them in CD pipelines.

The implementation of either GitOps model provides several benefits: faster feedback loops due to automated deployments, complete transparency through Git's change history, increased application reliability through simplified rollbacks, stronger security through Git's commit signing capabilities, and rapid infrastructure restoration capabilities since complete configurations are stored in repositories [1].

According to Kormaník and Porubän [9], GitOps implementation provides several critical benefits, including more frequent deployment of versions, easier management approaches to projects, immediate availability of backup plans, shared documentation, environment configurations stored directly in repositories, and an object-oriented approach to infrastructure management.

Relationship Between DevOps and GitOps

While GitOps is sometimes characterized as "the evolution of DevOps," research by Beetz and Harrer [1] shows that GitOps principles and practices work alongside DevOps rather than replacing them. Their analysis shows that "using one approach does not prevent the use of the other" [1]. The key distinction is that DevOps centers on broader concepts such as organizational culture and cross-functional collaboration, while GitOps provides specific tools and techniques to implement continuous deployment workflows.

GitOps shares several DevOps principles including incremental changes, continuous processes, automation, and self-service capabilities, but places less emphasis on the DevOps principles of iteration and collaboration [1]. Similarly, GitOps employs most DevOps practices, particularly CI/CD pipelines, infrastructure as code, and monitoring, but extends these practices with specific implementation patterns for Kubernetes environments.

Empirical Validation of GitOps with ArgoCD

While the theoretical principles of GitOps provide a foundation for understanding, empirical testing offers valuable insights into practical implementation challenges and benefits. Research by Ramadoni et al. [17] provides experimental validation of GitOps principles through hands-on implementation using ArgoCD in Kubernetes environments.

Their empirical testing focused on addressing two critical DevOps challenges that are particularly relevant in IoT and microservice environments: "security issues of a person's ability to directly access and change clusters and the ineffective rollback process in the application deployment process" [17]. These challenges align closely with the security and reliability concerns in IoT SaaS deployments discussed in this thesis.

The pull-based deployment model implemented through ArgoCD demonstrated significant security advantages over traditional push-based approaches. As Ramadoni et al. note, "for security reasons, the operator must always be in the same environment or cluster as the program is applied... There are no credentials that the external service needs to know about in Pull-based Deployments" [17]. This containment of credentials within the cluster environment substantially reduces the attack surface compared to traditional CI/CD pipelines where external systems require direct cluster access.

Their experimental implementation also validated the effectiveness of configuration drift detection in real-world scenarios. When direct changes were made to the Kubernetes cluster outside of the GitOps workflow (using kubectl commands), "Argo CD detected it and marks the implementation as OutOfSync" [17]. This automatic detection capability ensures that unauthorized or undocumented changes are immediately identified, maintaining system integrity and providing "instant visibility of asynchronous deployments" [17].

Perhaps most significantly, the testing demonstrated dramatic improvements in rollback capabilities compared to traditional deployment methods. In conventional CI/CD approaches, rollbacks often require complex pipeline reconfigurations. However, with the GitOps implementation, "with Argo CD we just simply select another git release on the web dashboard that will be used for the synchronization process and ensure that after a while the cluster will return to that commit hash" [17]. This validation confirms that the theoretical benefits of Git-based version control translate to practical operational advantages in real-world deployment scenarios.

These empirical findings reinforce that GitOps implementation with ArgoCD can effectively address key challenges in deployment management for complex distributed systems, providing practical validation of the theoretical principles discussed in the previous section.

2.4.2 ArgoCD Architecture

López-Viana et al. [11] describe ArgoCD as a tool that "mainly automates infrastructure" and serves as a CD tool in the GitOps workflow. In their proof of concept, they used ArgoCD installed on a Kubernetes cluster to implement a GitOps workflow for IoT Edge Computing environments.

The GitOps workflow implemented with ArgoCD involves the CD tool monitoring both an Artifact Registry and an Infrastructure Application Repository. When a change is detected, ArgoCD initiates deployment by informing the Infrastructure Provider of the target state, which then aligns the existing state with the new configuration through necessary updates [11]. ArgoCD maintains ongoing oversight of this process, verifying that the state defined in Git matches what is actually running in the infrastructure.

2.4.3 GitOps Implementation Methodology with Kubernetes

While theoretical models provide the foundation for understanding GitOps, examining concrete implementation methodologies offers valuable insights into practical applications and measurable benefits. Research by Kurrewar et al. [10] demonstrates a systematic approach for automating Kubernetes deployments using GitOps principles, with empirical validation of the benefits in production environments.

The implementation methodology follows a structured workflow that operationalizes the GitOps principles previously discussed. This process begins with infrastructure provisioning using infrastructure-as-code (IaC) tools like Terraform, which ensures "repeatability and consistency" by specifying "the ideal state of the infrastructure in code that can be controlled and versioned via a Git repository" [10]. This foundation aligns with the declarative description principle of GitOps, ensuring that all infrastructure components are defined as code.

Once the infrastructure is established, YAML manifest files are created to define the desired state of applications. These manifests specify deployment configurations, service definitions, ingress rules, and resource management parameters. As Kurrewar et al. note, "Within the architecture, every application or service is represented by a manifest file that contains deployment, service, ingress, and resource management configurations" [10]. This approach implements the versioned and immutable state principle by storing all configurations in Git repositories.

The implementation then establishes a CI/CD pipeline using GitHub Actions that automates the validation and deployment process. This pipeline automatically verifies YAML syntax, provisions Kubernetes infrastructure, and deploys applications using Helm charts. The researchers emphasize that "Changes made to the codebase or configuration files are automatically reflected in the Kubernetes cluster because of this pipeline's automation of the deployment process" [10], implementing the change approvals automation principle.

The final component involves configuring ArgoCD to continuously monitor the Git repository for changes, automatically synchronizing the cluster state with the desired state defined in the repository. This implements the "alert on differences" principle, as ArgoCD "keeps an eye out for updates in the repository and performs an automatic synchronization of the Kubernetes cluster with the planned state if updates are found" [10].

Empirical Benefits of GitOps Implementation

The empirical implementation demonstrated several quantifiable benefits that validate the theoretical advantages of GitOps. Kurrewar et al. identified five key benefits in their production deployment [10]:

- **Automated Deployment**: The use of YAML manifest files with ArgoCD enabled fully automated resource delivery to Kubernetes clusters, reducing manual intervention and human error.
- **Continuous Monitoring**: The implementation maintained constant synchronization between the Git repository and the cluster state, ensuring configuration consistency and providing immediate visibility into deployment status.

- **Enhanced Version Control**: Using Git for deployment configurations improved tracking and collaboration, creating a complete audit trail of all infrastructure changes.
- **Simplified Rollback Capability**: The ability to revert to previous stable versions was dramatically improved, increasing overall system reliability and reducing recovery time during incidents.
- **Integrated Monitoring**: The incorporation of monitoring tools provided actionable insights into application performance and resource utilization, enabling proactive management.

These benefits directly address key challenges in IoT deployment environments, particularly the security and reliability concerns previously mentioned. The empirical results demonstrate that GitOps implementation with ArgoCD successfully automates "the deployment of apps on Kubernetes, integrating tools such as Argo CD and Helm to enhance the effectiveness and dependability of application delivery" [10].

The implementation was particularly effective at maintaining state consistency, with ArgoCD ensuring that "the expected state specified in the manifest files and the original state in the Kubernetes cluster were precisely synced" [10]. This capability is crucial for IoT environments where maintaining configuration consistency across distributed systems presents significant challenges.

These empirical findings validate the theoretical GitOps principles discussed in previous sections and demonstrate that practical implementations can deliver measurable improvements in deployment automation, reliability, and management efficiency in production environments.

2.5 Monitoring in Microservice Environments

As identified by Saputra et al. [19], "regular server monitoring plays a crucial role in identifying and resolving overload issues." Their research highlights that "network administrators frequently fail to notice network outages," which "can result in time-consuming troubleshooting that causes serious network degradation concerns." This challenge necessitates effective monitoring solutions.

Server monitoring systems can be implemented through the integration of specialized tools designed for data collection, visualization, and notification. According to Saputra et al. [19], Prometheus effectively serves as "a monitoring platform with the advantage of being open-source, able to collect server resource data through node exporters and store them in time series format." This approach allows for systematic collection and storage of critical operational metrics.

For visualization purposes, Grafana complements Prometheus by "facilitating data visualization in the form of graphs, tables and text to overcome Prometheus' weaknesses in data visualization" [19]. The paper demonstrates that these visualizations can present key metrics including CPU usage, memory utilization, and disk storage in percentage formats, enabling administrators to quickly assess system status through graphical interfaces.

Grafana has emerged as a powerful platform for monitoring due to its extensive features and flexibility. As Mehdi et al. [14] explain, Grafana excels in "data exploration and querying capabilities" while providing "a built-in query editor that allows users to delve into their data and extract meaningful insights." The platform offers significant advantages for microservice environments through its ability to integrate with diverse data sources, create interactive dashboards, and provide real-time alerting.

The effectiveness of Grafana in microservice monitoring stems from several key capabilities identified by Mehdi et al. [14]. First, its data caching and aggregation techniques optimize query performance, which is crucial for monitoring distributed systems with high data volumes. Second, its query optimization capabilities minimize execution time, resulting in faster response times for visualizations. Third, its alerting engine employs "smart evaluation and deduplication techniques, reducing false positives and false negatives," which improves alert quality in complex microservice environments.

Beyond basic monitoring, Grafana supports advanced visualization types particularly valuable for understanding microservice interactions, including heatmaps for correlation analysis, singlestat panels for key metrics, and comprehensive dashboards that combine multiple visualization types [14]. These capabilities allow for more effective observation of interservice dependencies and performance patterns across distributed architectures.

A critical component of effective monitoring is timely notification when issues arise. Saputra et al. [19] implemented integration between Grafana and Telegram, where "an API is implemented within Grafana to send alert messages directly via the Telegram bot when server overload occurs." The researchers note that "notification parameters can be customized, such as CPU usage above 80% for a certain period of time, memory usage near full, or disk capacity approaching 100%."

The effectiveness of this monitoring approach was validated through user assessment, with results showing "an average percentage of 85.33%" satisfaction among administrators [19]. Performance testing further demonstrated that "the time required to send a notification is less than 30 seconds in all scenarios," enabling rapid response to emerging server issues.

Through this integrated monitoring approach using Prometheus for data collection, Grafana for visualization, and Telegram for notifications, organizations can maintain better visibility into their server environments and respond more quickly to potential issues. As Saputra et al. [19] conclude, this solution "contributes significantly to maintaining server system performance and reliability."

3. RESEARCH METHODOLOGY

This chapter describes the research methodology employed in this study. It presents the constructive research approach, the case study design including research questions and company context, and the data collection methods and evaluation criteria used to assess the implemented solutions.

3.1 Constructive Research Approach

Constructive research is a methodological approach that aims to produce solutions to real-world problems through the development of innovative constructions. This methodology is particularly suitable for studies where practical solutions are developed and evaluated within their operational context, making it an appropriate choice for this DevOps transformation project.

This thesis employs a constructive research methodology, which was selected based on the practical implementation nature of the work conducted at Treon. The constructive approach follows a structured process that includes problem identification, solution design, implementation, and evaluation.

When I began my work at Treon, I was assigned three primary objectives that shaped the research approach:

1. Investigating and implementing an appropriate Continuous Deployment solution.
2. Identifying and adapting deployment strategies to minimize client downtime during updates.
3. Analyzing the existing Continuous Integration pipelines to propose and implement standardization improvements.

The methodology followed a systematic process that included four distinct phases:

1. Problem identification through analysis of the existing DevOps infrastructure and workflows.
2. Solution exploration, where I researched available tools and methodologies suitable for the company's microservice-based IoT SaaS architecture.

3. Phased implementation, beginning with proof-of-concept deployments and proceeding through development environments before reaching production.
4. Evaluation of the implemented solutions using quantitative and qualitative measures.

The selection of constructive research methodology aligns with the practical requirements of implementing CI/CD improvements in a production environment, where theoretical models must be translated into functional technical solutions. Unlike purely theoretical approaches, constructive research enables the immediate application and validation of concepts within an operational context, providing both practical value to the organization and academic contributions to the understanding of DevOps practices in IoT environments.

3.2 Case Study Design

While this research follows a constructive approach, the empirical investigation is structured as a single-case study conducted at Treon. According to Runeson and Höst [18], case study research is well-suited for software engineering contexts where contemporary phenomena are studied within their natural environment. The case study methodology enables investigation of DevOps transformation practices in their operational context, where the boundary between the phenomenon and its environment is not clearly defined [18].

This case study is classified as an *improving* case study, as defined by Runeson and Höst [18], aiming to improve DevOps practices through the implementation of GitOps and CI/CD standardization.

Following the guidelines established by Runeson and Höst [18], this case study employs multiple data sources including quantitative metrics and qualitative observations, enabling triangulation of findings. The unit of analysis is the DevOps transformation project at Treon, encompassing CI/CD pipeline standardization and GitOps implementation for the Treon Connect platform.

Research Questions

This research aims to address the following key questions:

1. Which deployment strategies provide optimal outcomes for minimizing downtime, mitigating risk, and facilitating rapid rollbacks in a microservice-based IoT SaaS platform?
2. How can GitOps implementation enhance the management and reliability of customer deployments in a microservice-based IoT SaaS environment?

3. How can CI pipeline standardization through shared libraries improve efficiency and reliability in multi-repository microservice architectures?

These questions were formulated to address practical challenges observed in the case company while contributing to the broader understanding of DevOps practices in IoT contexts.

Research Process and Timeframe

The research was conducted over a six-month period from January to June 2025, following a systematic process with defined phases:

- **Month 1-2**: Problem identification and literature review
- **Month 2-3**: Solution design and planning
- **Month 3-5**: Implementation across development and production environments
- **Month 5-6**: Evaluation, data collection, and analysis

Throughout this process, regular documentation and reflection were maintained to capture insights and learning that would inform both the practical implementation and the academic analysis.

Company Profile and Context

This research was conducted at Treon, a technology company specializing in the design and development of intelligent edge products for massive Internet of Things (IoT) solutions. Treon's primary focus is empowering businesses across industrial, smart buildings, and logistics sectors to create scalable IoT solutions. The company's organizational structure includes software development teams, hardware engineering teams, quality assurance, sales, marketing, and product management departments.

The focal point of this research is Treon Connect, a comprehensive IoT platform designed to optimize physical operations by providing advanced data insights, seamless integration with existing systems, and automation capabilities. At the time of this research, Treon Connect was a relatively new offering that had been recently announced to the market, representing the company's strategic shift toward Software as a Service (SaaS) delivery models.

3.3 Data Collection & Evaluation Criteria

Quantitative Metrics

The quantitative evaluation focused on measurable improvements in deployment processes and CI pipeline efficiency. Deployment time measurements were collected by comparing the duration of deployment operations before and after ArgoCD implementation, tracking the time from initiation to completion of client environment deployments. Error rates were monitored through Jenkins build logs and ArgoCD deployment status, documenting the frequency of failed deployments and configuration errors.

Code duplication reduction metrics were gathered by analyzing the Jenkins shared libraries, measuring the percentage decrease in duplicated configuration across microservice repositories. This assessment involved comparing the number of lines of common code before and after standardization. Pipeline efficiency metrics tracked build duration, failure recovery time, and the frequency of successful automated deployments across the development lifecycle.

Resource utilization data for deployment strategies was collected during proof-of-concept implementations, measuring CPU and memory consumption patterns for blue-green and rolling update approaches. These measurements informed the evaluation of deployment strategy suitability for Treon's production environment.

Qualitative Assessment

Qualitative assessment methods provided insights into user experience, workflow changes, and organizational adoption patterns. Team feedback was collected through informal discussions with developers, operations staff, and management following the implementation of new tools and processes. These conversations focused on perceived improvements in deployment reliability, ease of use, and confidence in the deployment process.

Usability evaluation was conducted through observation of team members interacting with ArgoCD's web interface and the standardized CI pipelines. Particular attention was paid to the learning curve for new team members and the reduction in specialized knowledge required to perform deployments. The accessibility of deployment information and the clarity of error messages were also assessed.

Maintenance effort assessment involved comparing the time required to update deployment configurations and CI pipeline components before and after standardization. This qualitative measure captured the practical impact of centralized shared libraries and GitOps practices on daily operational workflows. Additionally, the research documented challenges encountered during implementation and the solutions developed to address them, providing insights into the practical considerations of DevOps transformation in IoT environments.

4. CASE STUDY: DEVOPS TRANSFORMATION AT TREON

This chapter presents the practical implementation of DevOps transformation at Treon. It describes the technical environment and initial state assessment, followed by detailed accounts of the GitOps implementation, deployment strategy evaluation, CI pipeline standardization, and monitoring infrastructure setup.

4.1 Technical Environment

Treon's technical architecture employs a single-tenant deployment model, where each client receives a dedicated instance within an isolated Kubernetes namespace. This architecture ensures strong client separation but introduces complexities in the deployment and management processes. The platform operates as a microservice-based system built on Azure cloud infrastructure with Kubernetes orchestration. The diverse technical requirements of different microservices are reflected in the use of multiple backend programming languages, including Python, C, C++, and Rust. Data persistence is handled through InfluxDB, which provides specialized time-series data storage capabilities particularly important for IoT telemetry collection and analysis.

The system architecture follows a RESTful API approach supplemented by event-driven components via message buses. Frontend interfaces are developed using TypeScript with Material UI components, providing a consistent user experience across the platform. Communication patterns within the system employ a hybrid approach, combining synchronous HTTP requests for immediate operations, WebSockets for real-time updates, and MQTT protocol for efficient IoT device communication.

The system architecture integrated various components including Traefik proxy for routing, NATS message bus for inter-service communication, device management services, and specialized data processing pipelines for IoT telemetry. The overall complexity was significant, with interdependencies between services requiring careful coordination during deployment processes.

4.2 Initial State Assessment

When the author joined the company, Treon had established a functional Continuous Integration (CI) infrastructure but lacked an equivalent Continuous Deployment (CD) solution. The existing CI/CD landscape centered around Jenkins as the primary continuous integration tool, handling code compilation, automated testing, and container image building across multiple repositories. Application packaging was accomplished through Docker containerization, with successfully built container images being pushed to a private registry upon build completion.

However, the deployment process remained largely manual, requiring designated team members to execute Helm commands for service deployments. Configuration management relied on several shell scripts to handle secrets and configuration files, which significantly increased the risk of human error during deployment operations. The limited documentation of deployment procedures created an overreliance on specific team members' institutional knowledge, presenting both operational risks and scalability challenges. As the company scaled and the platform complexity increased, manual deployments became increasingly time-consuming and error-prone. The process required specialized knowledge and significant coordination between team members, often resulting in deployment delays and occasional service disruptions.

A comprehensive assessment of these existing deployment processes revealed several critical challenges that needed to be addressed. Manual deployment processes did not scale effectively as the number of services and client instances increased, creating significant scalability limitations. Deployment expertise was concentrated among a small number of team members, creating knowledge silos that resulted in bottlenecks and potential single points of failure. Stakeholders lacked clear visibility into deployment status, errors, and configuration changes, making it difficult to track system state and troubleshoot issues effectively. The manual steps in the deployment process introduced inconsistencies and human errors due to insufficient automation, while the existing approach resulted in service interruptions during updates, impacting client operations. Without proper GitOps practices, environment configurations could drift from their expected states, leading to configuration drift issues. Additionally, duplication of configuration across multiple repositories created maintenance overhead and inconsistencies in the CI pipeline.

These challenges became particularly acute as Treon's client base expanded and the frequency of deployments increased, highlighting the urgency for implementing a more robust CD solution and standardizing CI pipelines. The development of an improved CI/CD solution needed to address requirements from multiple stakeholders across the organization. Development teams required faster feedback loops, reduced deployment complexity, and elimination of manual steps to improve their productivity. The operations team needed improved visibility into system state, easier troubleshooting capabilities, and reduced maintenance burden to manage the growing infrastructure effectively. Quality assurance expected consistent environments across development, testing, and production to ensure reliable testing outcomes. Management required reduced time-to-market, improved system reliability, and efficient resource utilization to maintain competitive advantage and operational efficiency. Finally, clients expected minimal service disruptions during updates and consistent service quality to maintain trust in the platform.

These requirements shaped the approach to implementation and guided the selection of tools and methodologies throughout the research process. The solution needed to accommodate the company's single-tenant architecture while providing the scalability and reliability expected in a professional SaaS environment.

4.3 Implementation Approach

Building upon the identified challenges and stakeholder requirements, the implementation approach was structured around three core components: GitOps implementation, deployment strategy evaluation, and CI pipeline standardization. Each component addressed specific operational gaps while contributing to the overall DevOps transformation objectives.

4.3.1 GitOps Implementation with ArgoCD

The implementation of GitOps principles at Treon followed an iterative approach, beginning with a proof-of-concept deployment before scaling to production environments. The selection of ArgoCD as the GitOps tool was made after evaluating several alternatives including Flux CD and Jenkins X, with ArgoCD ultimately chosen for its robust Kubernetes integration, intuitive user interface, and support for multiple configuration management tools.

The implementation process proceeded through four distinct phases, beginning with infrastructure preparation and installation. The initial setup involved deploying ArgoCD within the existing Kubernetes cluster:

```
# Create dedicated namespace for ArgoCD
kubectl create namespace argocd

# Deploy ArgoCD using the official Helm chart
helm repo add argo https://argoproj.github.io/argo-helm
helm install argocd argo/argo-cd -n argocd --values custom-values.yaml
```

Custom configurations were applied to integrate with the company's existing authentication system and to establish appropriate RBAC policies aligning with team responsibilities.

We modified the default ArgoCD configuration to configure OpenID Connect (OIDC) authentication integration with Azure Active Directory, establish project-based access control aligned with team structures, optimize resource allocation for the ArgoCD controller and server components, and configure appropriate health check settings for our specific services.

Following this initial infrastructure setup, the second phase leveraged Treon's existing repository structure, which was already well-suited for GitOps implementation. The company maintained two types of repositories: application repositories containing the application source code and CI pipelines, and a deployment repository containing all Helm charts and deployment configurations required for service deployment. This established structure supported the single-tenant deployment model while maintaining configuration consistency across environments. Each client's configuration was isolated appropriately, enabling independent management while still leveraging common templates and shared configurations.

The third phase focused on application definition and synchronization policies. Applications in ArgoCD were defined using a declarative approach, with each client instance represented as a separate ArgoCD Application:

```
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: client-a-production
  namespace: argocd
spec:
  project: client-deployments
  source:
    repoURL: https://github.com/example-company-directory/client-configs.git
    path: clients/client-a/production
    targetRevision: HEAD
  destination:
    server: https://kubernetes.default.svc
    namespace: client-a-prod
  syncPolicy:
    automated:
      prune: true
      selfHeal: true
    syncOptions:
      - CreateNamespace=true
```

The synchronization policies were configured differently for development and production environments. The development environment used its own separate ArgoCD setup with manual synchronization to allow for controlled testing, while the production environment implemented automatic synchronization configured to track specific version numbers using Git tags, ensuring precise version control. We implemented these synchronization policies to balance control with automation, ensuring that development changes could be thoroughly validated before reaching production, while production deployments would automatically stay synchronized with their specified versions.

The final implementation phase connected the existing Jenkins CI pipelines with the new GitOps workflow. Rather than modifying how Kubernetes manifests were generated, we extended the CI/CD process by adding an ArgoCD synchronization command at the end of the Jenkins deployment jobs:

```
# Example ArgoCD sync command added to Jenkins pipeline
stage('Trigger ArgoCD Sync') {
    steps {
        withCredentials([string(credentialsId: 'argocd-auth-token',
                                variable: 'ARGOCD_AUTH_TOKEN')]) {
            sh '''
                # Authenticate with ArgoCD using token
                argocd login ${ARGOCD_SERVER} \
                    --auth-token ${ARGOCD_AUTH_TOKEN} \
                    --grpc-web

                # Trigger sync for the application
                argocd app sync ${APPLICATION_NAME}

                # Wait for sync to complete and verify health
                argocd app wait ${APPLICATION_NAME} --health
            '''
        }
    }
}
```

This approach ensured that when a new version was successfully built and tested, ArgoCD would detect the changes and apply them according to the defined policies. The integration completed the CI/CD loop while maintaining the separation of concerns between CI and CD processes.

Several key architectural decisions shaped the ArgoCD implementation and proved critical to its success. Each client instance was represented as a separate ArgoCD application rather than using a single application with multiple destinations, enhancing isolation and management flexibility. Separate ArgoCD installations for development and production provided additional isolation between environments. The use of Git tags for version control in production ensured precise tracking of deployed versions, while ArgoCD's pull-based approach eliminated the need for CI systems to have direct cluster access, improving the overall security posture. These decisions formed the foundation of a robust GitOps implementation that addressed Treon's specific requirements for managing multiple single-tenant client deployments while maintaining security and scalability.

4.3.2 Deployment Strategy Selection and Implementation

With the GitOps foundation established, the next phase focused on evaluating and selecting appropriate deployment strategies that would complement the automated deployment capabilities while addressing the unique constraints of the single-tenant architecture. The selection of appropriate deployment strategies involved evaluating multiple approaches against Treon's specific technical requirements and business constraints. This process followed a systematic approach combining theoretical assessment with practical testing in isolated environments.

Three primary deployment strategies were evaluated: canary deployment, initially considered for its ability to limit risk exposure; blue-green deployment, evaluated for its zero-downtime capabilities; and rolling updates, assessed for balance between simplicity and reliability. The evaluation used a weighted criteria matrix incorporating several critical factors, including compatibility with single-tenant architecture, resource efficiency and infrastructure requirements, rollback capabilities and recovery time, complexity of implementation and maintenance, impact on database schema migrations, and alignment with existing operational workflows.

The first strategy explored was canary deployment. A prototype implementation was developed using Istio service mesh to manage traffic routing:

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: client-service
spec:
  hosts:
  - client-service
  http:
  - route:
    - destination:
        host: client-service
        subset: v1
```

```
      weight: 90
    - destination:
        host: client-service
        subset: v2
      weight: 10
```

However, testing revealed significant challenges with this approach in a single-tenant environment. The granularity of canary deployments, which is typically percentage-based, was misaligned with the all-or-nothing nature of individual client deployments. The additional complexity of managing the service mesh outweighed the benefits for individual tenant deployments, and the monitoring and observability requirements for effective canary analysis were substantial.

Following the canary deployment evaluation, blue-green deployment was subsequently implemented as a proof-of-concept using a namespace-based approach:

```
# Create a new "green" namespace for the updated deployment
kubectl create namespace ${CLIENT_ID}-green

# Deploy the new version to the green namespace
helm upgrade --install ${CLIENT_ID} ./charts/client-deployment \
  --namespace ${CLIENT_ID}-green \
  --set version=${NEW_VERSION}

# Validate deployment in green environment
./validate-deployment.sh ${CLIENT_ID}-green

# Switch traffic from blue to green (update Ingress)
kubectl apply -f ./manifests/ingress-green.yaml

# Keep blue environment as fallback for potential rollback
```

The proof-of-concept demonstrated several benefits, including complete elimination of downtime during deployments, simple and immediate rollback capabilities, and clear separation of concerns between versions.

After evaluating these available options, a namespace-level rolling update strategy was selected as the most appropriate approach for Treon's architecture. This decision was based on the strategy's balanced combination of reliability, resource efficiency, and minimal service disruption potential. In Treon's single-tenant architecture, the concept of rolling updates needed to be adapted from the traditional pod-level approach to a namespace-level implementation. This distinction is important to understand: in standard Kubernetes deployments, rolling updates occur at the pod level within a single namespace, where

Kubernetes gradually replaces pods of the old version with pods of the new version according to defined parameters. However, in our single-tenant architecture, each client has a dedicated namespace containing their entire service stack. A namespace-level rolling update means updating client namespaces one by one in sequence, rather than updating individual pods within a single namespace.

This approach meant developing a custom orchestration layer that would select client namespaces for updates according to defined batching rules, perform pre-update validation on each namespace, execute the update for an entire client namespace, conduct post-update health checks at the namespace level, and proceed to the next namespace only after successful validation. The health check mechanism also needed adaptation to work at the namespace level, verifying the overall health of a client's service stack rather than individual pods. This involved creating aggregate health metrics that would determine if a client namespace was operating correctly after an update.

The planned rolling update implementation would leverage Kubernetes' built-in capabilities with carefully tuned parameters, but orchestrated at the namespace level:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: client-service
spec:
  replicas: 3
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxUnavailable: 1
      maxSurge: 1
  template:
    # Pod template specification
```

The rolling update strategy would be enhanced with advanced health checking through implementation of both readiness and liveness probes with appropriate timing parameters, pre-deployment validation involving automated tests executed against the new version before initiating the update, and gradual rollout with carefully tuned parameters to control the pace of updates. While the rolling update strategy was selected as the optimal approach, the actual implementation was deferred due to company focus on other priorities. The implementation design has been completed and documented for future execution when resources become available.

Based on the evaluation results, a staged deployment strategy roadmap was developed to guide future implementation. Phase 1, which has been completed, involved implementing

GitOps with ArgoCD for deployment automation and configuration management. Phase 2, which is planned, will implement namespace-level rolling updates with optimized health checks to minimize service disruption. Phase 3, for future consideration, involves revisiting blue-green deployment for critical services when resource constraints are addressed. Phase 4, as a long-term goal, explores hybrid approaches combining aspects of multiple strategies. This phased approach allows the company to incrementally improve deployment processes while balancing immediate operational needs with long-term strategic goals.

4.3.3 CI Pipeline Standardization

The standardization of CI pipelines aimed to address specific inefficiencies identified in Treon's multi-repository microservice architecture. Rather than a complete redesign, this effort focused on targeted improvements to reduce duplication, increase reusability, and enhance reliability of the build processes.

An initial assessment of the existing Jenkins pipelines across microservice repositories revealed moderate code duplication and several opportunities for standardization. Common configuration blocks were duplicated across multiple repositories, while shared resources such as Dockerfiles and build scripts were independently maintained in each repository. The analysis also identified an inconsistent approach to pipeline organization and error handling, along with variable testing practices leading to occasional reliability issues.

Treon already had a Jenkins shared library in place for some functionalities, and this existing infrastructure provided an excellent foundation to build upon. The enhancement work focused on expanding the library's capabilities to address the identified inefficiencies:

```
// vars/buildDockerImage.groovy
def call(Map config) {
    def imageName = config.imageName ?: "${env.JOB_NAME}"
    def imageTag = config.imageTag ?: "${env.BUILD_NUMBER}"
    def dockerfilePath = config.dockerfilePath ?: "./Dockerfile"

    echo "Building Docker image ${imageName}:${imageTag}"

    sh """
        docker build -t ${imageName}:${imageTag} \
        -f ${dockerfilePath} .

        docker push ${imageName}:${imageTag}
    """
```

```
    return "${imageName}:${imageTag}"
}
```

A key focus of this enhancement work was centralizing shared resources that were previously duplicated across repositories. Common Dockerfiles were moved into the shared library and made available as templates:

```
# Shared library: Python microservice base template
# resources/org/company/docker/python-service/Dockerfile
FROM python:3.9-slim

# Set working directory for application
WORKDIR /app

# Install dependencies first (for Docker layer caching)
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

# Copy application source code
COPY . .

# Default startup command
CMD ["python", "app.py"]
```

These resources could then be imported into pipelines as needed:

```
// Example of using a shared Dockerfile in a pipeline
stage('Prepare Build Environment') {
    steps {
        script {
            // Import Dockerfile from shared library
            def dockerfileContent = libraryResource \
              'org/example/docker/python-service/Dockerfile'
            writeFile file: 'Dockerfile', text: dockerfileContent
        }
    }
}
```

This approach significantly reduced duplication by allowing teams to reference shared resources rather than maintaining copies in each repository.

To improve visibility into development progress and align with the company's sprint-based workflow, an automated sprint review report generator was implemented as a Jenkins job:

```
// Example of sprint review report generation job
pipeline {
    agent any

    parameters {
        string(name: 'SPRINT_NUMBER', defaultValue: '',
               description: 'Sprint number to generate report for')
        string(name: 'START_DATE', defaultValue: '',
               description: 'Sprint start date (YYYY-MM-DD)')
        string(name: 'END_DATE', defaultValue: '',
               description: 'Sprint end date (YYYY-MM-DD)')
    }

    stages {
        stage('Generate Report') {
            steps {
                script {
                    def repositories = ['service-a', 'service-b',
                                        'service-c', 'service-d']
                    def report = "# Sprint ${params.SPRINT_NUMBER}" +
                                 " Review Report\n\n"
                    report += "Period: ${params.START_DATE}" +
                              " to ${params.END_DATE}\n\n"
                    report += "## Version Updates\n\n"

                    repositories.each { repo ->
                        // Get latest version in date range
                        def versions = sh(
                            script: """
                                git -C /path/to/${repo} log \\
                                  --after=${params.START_DATE} \\
                                  --before=${params.END_DATE} \\
                                  --grep='version' \\
                                  --pretty=format:'%h %s'
                            """,
                            returnStdout: true
                        ).trim()

                        report += "### ${repo}\n"
                        if (versions) {
```

```
                        report += versions.replace('\n', '\n- ')
                    } else {
                        report += "- No version updates\n"
                    }
                    report += "\n"
                }

                def fileName = "sprint-${params.SPRINT_NUMBER}" +
                              "-report.md"
                writeFile file: fileName, text: report
            }
        }
    }
}

post {
    success {
        archiveArtifacts artifacts: "sprint-*.md",
                       allowEmptyArchive: true
    }
}
}
```

The generated reports were in Markdown format, making them easily shareable and readable across the organization.

Beyond shared library enhancements and automated reporting, several targeted fixes were implemented to address reliability issues in existing pipelines. Standardized error handling patterns were introduced to improve failure diagnostics, while retry mechanisms were implemented for transient network-related failures. Timeout controls were added to prevent builds from hanging indefinitely, and cleanup procedures were enhanced to prevent workspace pollution between builds. An example of the improved error handling implementation demonstrates these enhancements:

```
// Enhanced error handling in pipeline
stage('Build and Test') {
    steps {
        script {
            try {
                sh 'npm install'
                sh 'npm run build'
                sh 'npm test'
```

```
            } catch (Exception e) {
                // Enhanced error reporting
                def errorMessage = "Build failed in ${env.STAGE_NAME}" +
                                 " stage: ${e.message}"
                echo errorMessage

                // Capture relevant logs
                sh 'tar -czf build-logs.tar.gz logs/'
                archiveArtifacts artifacts: 'build-logs.tar.gz'

                // Re-throw to mark build as failed
                throw e
            }
        }
    }
}
```

These reliability improvements helped reduce pipeline failures and made troubleshooting more efficient when issues did occur.

In addition to the implemented changes, discussions were held with stakeholders regarding further CI pipeline enhancements. A key focus was improving static analysis capabilities by evaluating SonarQube as a more comprehensive static analysis solution. The discussions included identification of integration points within existing pipelines and assessment of resource requirements for implementation. Stakeholders also developed a preliminary integration plan for future implementation. These discussions resulted in a roadmap for future CI improvements that would build upon the standardization work already completed.

The CI pipeline standardization achieved measurable improvements in development workflow efficiency. The shared library reduced configuration duplication across repositories by approximately 60%, while the automated sprint reporting eliminated 2-3 hours of manual work per sprint cycle. Pipeline reliability improvements resulted in fewer build failures due to environmental factors, though specific metrics were not formally tracked during the implementation period. Team adoption of the standardized approaches occurred gradually over the implementation period, with most repositories incorporating shared library components within the first month of availability. The enhanced error handling and retry mechanisms proved particularly valuable during periods of network instability or external service disruptions.

4.3.4 Monitoring Infrastructure Setup

The implementation of a comprehensive monitoring solution was critical for providing visibility into the deployed microservices and supporting the overall DevOps transformation. This section details the enhancements made to Treon's existing monitoring infrastructure and the development of specialized dashboards for operational insights.

When beginning this work, Treon had already established a foundation for monitoring with separate Grafana stacks for development and production environments. These stacks primarily utilized Loki for log aggregation and analysis, providing basic visibility into application logs. However, the existing setup lacked comprehensive metrics collection and visualization capabilities, limiting the ability to detect and diagnose operational issues. The existing infrastructure consisted of Grafana instances deployed in both development and production clusters, Loki for centralized log collection and analysis, and basic dashboards focused on log visualization.

To enhance the monitoring capabilities, Prometheus was integrated as a metrics data source alongside the existing Loki implementation. This integration significantly expanded the system's ability to collect and analyze time-series metrics across the infrastructure. The Prometheus integration involved configuring Prometheus as a data source in the existing Grafana installations, establishing appropriate scrape intervals and retention policies, setting up service discovery for automatic monitoring of new services, and ensuring appropriate security controls for metrics access. This integration enabled the collection of critical metrics from multiple layers of the architecture, including infrastructure-level metrics from Kubernetes nodes, Kubernetes-specific metrics for pods, deployments, and namespaces, application-level metrics from individual services, and cloud provider metrics from Azure resources.

With Prometheus successfully integrated, four specialized dashboards were developed to address specific monitoring needs identified during the DevOps transformation. The first dashboard focused on visualizing pod restarts and lifecycle events across the cluster and individual namespaces. Key components included time-series visualization of pod restart counts by namespace, pod status distribution by deployment, container resource utilization correlated with restart events, and a historical view of pod lifecycle events. This dashboard provided critical visibility into service stability and helped identify patterns in pod failures that might indicate deployment issues or resource constraints.

The second specialized dashboard was developed to monitor the critical NATS message bus that facilitated inter-service communication. This dashboard visualized message volume and rates across different subjects, queue depths and processing rates, consumer lag metrics, and connection and subscription counts. Implementing NATS monitoring presented unique technical challenges, as NATS does not natively expose metrics in a

Prometheus-compatible format. Addressing this required several custom configurations, including adding custom values to the NATS Helm chart to enable the Prometheus exporter within the NATS deployment, configuring ServiceMonitor resources to establish the connection between Prometheus and NATS metrics endpoints, customizing Prometheus scrape configurations to properly collect and parse NATS-specific metrics, and managing the appropriate scrape intervals to balance monitoring granularity with system performance.

The implementation process required enabling the Prometheus exporter within the NATS Helm values file:

```
exporter:
  enabled: true
  image: synadia/prometheus-nats-exporter:0.8.0
  serviceMonitor:
    enabled: true
    namespace: monitoring
```

Additionally, the Prometheus configuration required updates to its scrape configurations to properly target the NATS exporter endpoints. These technical adjustments were necessary to overcome the integration challenges and enable comprehensive monitoring of the messaging system. This dashboard enabled operations teams to identify bottlenecks in message processing and ensure the message bus was functioning efficiently.

To complement the GitOps implementation, a third dashboard was created to monitor ArgoCD and provide visibility into deployment activities. This dashboard tracked sync status across applications, sync duration metrics, configuration drift detection, and deployment frequency and error rates. The dashboard created a direct link between the monitoring system and deployment processes, enabling correlation between deployments and system behavior changes.

For production environment monitoring, a fourth comprehensive dashboard was developed to track Azure cluster resource utilization. This dashboard monitored CPU usage patterns across node pools, memory utilization and pressure metrics, storage consumption and growth trends, and network traffic analysis. The dashboard provided essential capacity planning insights and helped ensure efficient resource utilization in the production environment.

The monitoring implementation established a solid foundation while identifying several areas for future enhancement. A comprehensive alerting strategy has been planned for future implementation, which will include defining appropriate thresholds, notification channels, and escalation procedures based on operational experience. Future work will also enhance the correlation capabilities between metrics from different sources, enabling more sophisticated root cause analysis. Additionally, plans have been developed to implement machine learning-based anomaly detection to identify unusual patterns that might indicate issues before they affect service. The monitoring infrastructure implemented through this project provided essential visibility into the microservice environment, supporting both the ongoing DevOps transformation and day-to-day operations. The specialized dashboards addressed specific monitoring needs while creating a foundation for future enhancements as the platform continues to evolve.

5. RESULTS & FINDINGS

This chapter presents the outcomes and analysis of the DevOps transformation implemented at Treon. The results combine qualitative observations with available quantitative metrics to evaluate the effectiveness of the implemented solutions.

5.1 ArgoCD Implementation Outcomes

The implementation of ArgoCD as a GitOps solution at Treon produced observable improvements in deployment management and visibility. This section presents the outcomes observed after transitioning from manual deployment processes to the GitOps-based approach. The approach allowed for clear auditing of all changes through Git history, automated validation of configuration changes before deployment, simplified rollbacks through Git reversion when necessary, and consistent deployment processes across all services and clients.

The GitOps implementation with ArgoCD appeared to enhance deployment reliability across client environments. Prior to implementation, deployments frequently encountered issues related to configuration errors or environment inconsistencies. After implementation, several improvements were observed, including reduction in deployment failures caused by configuration errors, faster recovery from failed deployments through simplified rollback processes, and increased consistency between development and production environments. These improvements can be attributed to ArgoCD's declarative approach to configuration management and its ability to detect and reconcile configuration drift between actual and desired states.

The ArgoCD implementation fundamentally transformed operational visibility across multiple dimensions. The web-based ArgoCD dashboard provided immediate, real-time visibility into deployment status across all client environments, enabling team members to monitor progress without requiring specialized CLI expertise or command-line access. This democratization of deployment monitoring significantly improved team accessibility to critical operational information. Configuration changes became fully traceable through Git history integration, enabling precise identification of when changes were made and by whom. This audit trail capability addressed previous challenges with tracking configuration modifications and provided accountability for all system changes. Additionally, ArgoCD's health status visualization capabilities allowed rapid identification of problematic components, substantially reducing the time required to diagnose issues during deployments and ongoing operations. This enhanced visibility appeared particularly valuable during complex multi-service deployments, where the dependency visualization in ArgoCD helped teams identify ordering issues or failed dependencies.

One of the most notable improvements came from ArgoCD's ability to detect and report configuration drift. Before implementing ArgoCD, configuration drift between environments was a recurring issue that often remained undetected until it caused deployment failures or application errors. After implementation, ArgoCD provided automatic detection of differences between desired state in Git and actual state in the cluster, visual indicators of out-of-sync resources in the dashboard, detailed comparisons showing exact configuration differences, and optional automatic reconciliation of drift through self-healing capabilities. These capabilities helped identify configuration inconsistencies that would have previously gone unnoticed, allowing for proactive correction before they impacted services.

Beyond visibility and drift detection, the implementation of ArgoCD significantly improved team accessibility to deployment information and controls. Prior to implementation, deployment knowledge was concentrated among a small number of team members with the necessary expertise and access. The ArgoCD implementation democratized access to deployment information through role-based access control integrated with existing Azure AD authentication, a web-based interface accessible to all team members with appropriate permissions, simplified deployment controls that reduced the learning curve for new team members, and comprehensive visualization of application relationships and dependencies. Initial feedback from team members indicated improved access to deployment information and increased confidence in understanding the current state of deployments.

The GitOps implementation through ArgoCD transformed operational workflows in several observable ways. The time required for client environment deployments appeared to decrease significantly compared to the previous manual process, while the number of manual steps required during deployments decreased, reducing the opportunity for human error. The Git-based workflow facilitated collaboration between development and operations teams by providing a common platform for configuration management. Furthermore, the team was able to perform deployments more frequently, enabling more timely feature delivery and bug fixes. Perhaps the most significant operational improvement was the ability to quickly respond to production issues, with faster deployment of fixes through the streamlined process.

Overall, the ArgoCD implementation successfully transformed Treon's deployment processes from manual, error-prone procedures to automated, transparent workflows that enhanced both operational reliability and team collaboration.

5.2 Deployment Strategy Evaluation Results

This section presents the findings from the evaluation of different deployment strategies for Treon's microservice-based IoT SaaS environment. It's important to note that this evaluation was conducted through theoretical analysis and limited proof-of-concept testing rather than full production implementation. The evaluation focused on identifying the most suitable strategy for Treon's specific architecture and requirements.

The systematic evaluation of deployment strategies demonstrated that architectural constraints and resource limitations significantly influence strategy selection, with namespace-level rolling updates emerging as the optimal balance between operational simplicity and deployment reliability for Treon's single-tenant IoT environment.

5.2.1 Canary Deployment Analysis

Canary deployment was evaluated as a potential strategy through theoretical analysis and a proof-of-concept implementation using Istio service mesh in a test environment. The evaluation revealed several limitations that made this approach unsuitable for Treon's specific architecture.

The first major limitation was an architectural mismatch between canary deployment principles and Treon's infrastructure. The percentage-based traffic splitting central to canary deployments proved incompatible with Treon's single-tenant architecture. In a model where each client operates in isolation within their own namespace, the traditional approach of directing a small percentage of users to a new version could not be effectively applied. This fundamental incompatibility meant that the core benefit of canary deployment—gradual exposure of changes to a subset of users—could not be realized in the single-tenant context.

The evaluation also revealed significant monitoring requirements that would need to be addressed. Effective canary deployment requires comprehensive service-level metrics to evaluate the performance of the canary version. The evaluation identified gaps in the current monitoring infrastructure that would need to be addressed to support this strategy, representing a substantial investment beyond the deployment mechanism itself.

Furthermore, the operational complexity of implementing canary deployments presented another barrier. Implementing canary deployments would require additional infrastructure components, particularly a service mesh for traffic management. This added complexity was deemed disproportionate to the potential benefits in Treon's context, especially given the architectural limitations already identified.

The test implementation with Istio provided valuable insights into these challenges. For example, configuring appropriate traffic splitting rules at the namespace level proved particularly difficult, as shown in the configuration in Listing 5.1.

Listing 5.1. *Sample Istio configuration for canary routing (proof-of-concept)*

```yaml
# Treon proof-of-concept: Istio traffic splitting configuration
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: treon-tenant-svc  # Target service for canary routing
spec:
  hosts:
    - treon-tenant-svc
  http:
    - route:
        # Stable version receives majority of traffic
        - destination:
            host: treon-tenant-svc
            subset: stable
          weight: 90
        # Canary version for testing (10% exposure)
        - destination:
            host: treon-tenant-svc
            subset: canary
          weight: 10
```

Based on these findings, the canary deployment strategy was determined to be suboptimal for Treon's architecture and was not pursued further. This evaluation led to the conclusion that canary deployment, while valuable in multi-tenant or high-volume services, was not well-suited to Treon's single-tenant architecture where each client has a dedicated environment.

5.2.2 Blue-Green Deployment Analysis

Blue-green deployment was evaluated as a potential zero-downtime deployment strategy through theoretical analysis and a proof-of-concept implementation in a test environment. When evaluating deployment strategies for IoT environments, both performance and resource utilization must be carefully balanced. Research by Veith et al. [22] demonstrates that deployment strategies requiring duplicate environments, similar to blue-green deployment, can significantly increase resource requirements in large-scale deployments. Their findings show that while such approaches can achieve near-zero downtime, the resource overhead may be prohibitive at scale, with resource utilization doubling during deployment periods. This aligns with our evaluation of blue-green deployment, where we observed promising results for achieving zero-downtime deployments but encountered significant challenges with resource overhead and operational complexity.

The evaluation revealed both promising aspects and significant challenges. The proof-of-concept demonstrated that blue-green deployment could effectively eliminate downtime during updates by maintaining two parallel environments and switching traffic instantaneously. This zero-downtime capability represented a significant potential improvement over existing deployment approaches and validated the theoretical benefits of the strategy.

However, the approach presented substantial resource challenges. The implementation required maintaining duplicate environments during the deployment process, which would significantly increase resource utilization in a multi-client production environment. The resource requirements effectively doubled for each client during deployment periods, creating a prohibitive cost structure given the number of client instances that would need to be managed simultaneously.

The test implementation also revealed significant complexities in managing database connections and ensuring data consistency between blue and green environments, particularly for stateful services. These data synchronization challenges proved more complex than initially anticipated and would require careful architectural considerations and additional tooling to address effectively in a production context.

Furthermore, the strategy would require substantial changes to existing testing and validation processes to accommodate the parallel environment model. The approach required additional coordination and tooling beyond what most team members were familiar with, representing a significant operational complexity increase that would impact both implementation timelines and ongoing maintenance burden.

The proof-of-concept implementation used a namespace-based approach, as shown in Listing 5.2.

Listing 5.2. *Blue-green deployment script excerpt (proof-of-concept)*

```
# Create a new "green" namespace for the updated deployment
kubectl create namespace ${CLIENT_ID}-green

# Deploy the new version to the green namespace
helm upgrade --install ${CLIENT_ID} ./charts/client-deployment \
  --namespace ${CLIENT_ID}-green \
  --set version=${NEW_VERSION}

# Validate deployment in green environment
./validate-deployment.sh ${CLIENT_ID}-green
```

```
# Switch traffic from blue to green (update Ingress)
kubectl apply -f ./manifests/ingress-green.yaml

# Keep blue environment as fallback for potential rollback
```

Based on this evaluation, the company decided not to move forward with blue-green deployment at this time due to the complexity and resource overhead. While the blue-green approach showed promising results for achieving zero-downtime deployments, the evaluation concluded that the resource requirements and operational complexity made it impractical for immediate implementation. This strategy was noted as a potential future direction once resource constraints and data synchronization challenges could be addressed.

5.2.3 Rolling Update Strategy Selection

Based on the evaluation of available options, a namespace-level rolling update strategy was selected as the most appropriate approach for Treon's architecture. This decision was based on a qualitative assessment of several factors that aligned well with the company's operational context and constraints.

The primary advantage of rolling updates was their resource efficiency. Unlike blue-green deployment, rolling updates require minimal additional resources during the deployment process, making them suitable for a multi-client environment with resource constraints. This efficiency consideration proved critical given the need to manage deployments across multiple client namespaces simultaneously without dramatically increasing infrastructure costs.

While rolling updates do not achieve the complete zero-downtime of blue-green deployment, properly configured rolling updates can minimize service disruption to acceptable levels for Treon's use cases. The evaluation determined that brief, controlled service interruptions were preferable to the substantial resource overhead and operational complexity of alternatives.

The operational simplicity of rolling updates represented another significant advantage. The approach aligns well with existing operational workflows and requires less specialized knowledge compared to canary or blue-green strategies. This characteristic would facilitate team adoption and reduce the training burden associated with deployment process changes.

Additionally, the strategy provides adequate rollback capabilities through Kubernetes' built-in version control mechanisms, offering a practical balance between deployment safety and operational complexity.

It's important to note that this was a namespace-level adaptation of traditional rolling updates. In Treon's single-tenant architecture, each client has a dedicated namespace containing their entire service stack. The selected approach would manage updates at the namespace level rather than the traditional pod-level rolling update.

The evaluation included planning for implementation details to ensure the strategy would meet operational requirements. The planned implementation would incorporate advanced health checking through readiness and liveness probes to verify service readiness before routing traffic, pre-deployment validation through automated tests to catch issues before they reached production environments, gradual rollout with carefully tuned parameters to control the pace and risk of deployments, and automated failure detection and rollback capabilities to minimize the impact of problematic deployments.

When implemented, this approach would provide several significant benefits. Deployment time would be reduced compared to manual processes, while service disruption would be minimal, estimated at less than 10 seconds per service. The approach would maintain efficient resource utilization during deployment without the overhead of duplicate environments, and automated failure detection and rollback capabilities would provide safety nets for deployment issues.

An excerpt of the planned rolling update configuration is shown in Listing 5.3.

Listing 5.3. *Planned rolling update configuration*

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: client-service
spec:
  replicas: 3
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxUnavailable: 1
      maxSurge: 1
  template:
    # Pod template specification
```

While rolling updates were selected as the optimal approach, the actual implementation was deferred due to company priorities. The evaluation and design work provides a foundation for future implementation when resources become available. This evaluation confirmed the deployment strategy roadmap developed during implementation, demonstrating that a gradual evolution of deployment capabilities would provide the most practical path forward given Treon's specific constraints and priorities.

5.2.4 Alternative Implementation Approaches

The Blue-Green deployment implementation presented in this research utilized Kubernetes-native capabilities. However, alternative approaches exist that could address some of the challenges encountered in our implementation. Yang et al. [23] proposed a service discovery-based Blue-Green deployment method that offers valuable insights.

Their approach utilizes a service registry (Netflix Eureka) and a dynamic router (Netflix Zuul) to implement Blue-Green deployments with minimal service disruption. In this architecture, service instances register themselves in the service registry with their version information and corresponding IP addresses. The dynamic router queries the registry to retrieve IP hosts for load balancing, eliminating the need for manual IP collection and configuration updates.

This service discovery-based approach offers several key advantages over traditional Blue-Green implementations. First, new service instances automatically register with the service registry, eliminating the manual collection of IP information that often creates bottlenecks and opportunities for configuration errors in traditional approaches. This automatic registration significantly reduces the operational overhead associated with deploying new service versions.

Second, traffic routing decisions are made at request time based on current registry information, avoiding the propagation delays associated with DNS or load balancer rule updates. This dynamic routing capability enables near-instantaneous traffic switching without the waiting periods that can complicate traditional Blue-Green deployments and create windows of inconsistent behavior.

Third, the switch between versions can be controlled programmatically by updating service metadata in the registry, allowing for precise timing of transitions. This programmatic control provides operators with fine-grained authority over when and how traffic shifts between environments, reducing the risk of premature or mistimed cutover events.

While our Kubernetes implementation leverages different tooling, similar principles could be applied by utilizing Kubernetes service discovery mechanisms and custom routing controllers. The service discovery approach measured a significantly lower overlap duration compared to other Blue-Green implementations, with no unstable periods during transitions. This suggests potential optimization opportunities for our implementation if enhanced with more sophisticated service discovery mechanisms.

5.3 CI Pipeline Improvement Results

The standardization of CI pipelines was a key component of the DevOps transformation at Treon. This section presents the outcomes of the improvements made to the Jenkins-based CI infrastructure, focusing on enhancements to code reusability, pipeline reliability, and developer productivity.

5.3.1 Shared Library Implementation Outcomes

The implementation of shared Jenkins libraries successfully addressed several challenges identified in the initial pipeline analysis. Before the improvements, Treon's CI pipelines exhibited moderate code duplication and inconsistent approaches across multiple repositories, creating maintenance overhead and reliability issues.

The enhancement of the existing shared library infrastructure yielded several observable benefits. Common configuration blocks and build scripts previously maintained independently in each repository were centralized in the shared library, reducing duplication and maintenance overhead. The implementation established consistent patterns for pipeline organization, error handling, and resource management across repositories, creating a more uniform development experience. The shared library made it easier to implement CI pipelines for new microservices by providing reusable components and templates, reducing the time required to set up CI for new services. Additionally, centralized configuration management reduced the effort required to update common pipeline elements across multiple repositories, as changes could be made once in the shared library and automatically propagate to all dependent pipelines.

Listing 5.4 shows an example of a shared library function that standardized Docker image building across repositories:

Listing 5.4. *Example of shared Jenkins library function*

```
// vars/buildDockerImage.groovy
def call(Map config) {
  def imageName = config.imageName ?: "${env.JOB_NAME}"
  def imageTag = config.imageTag ?: "${env.BUILD_NUMBER}"
  def dockerfilePath = config.dockerfilePath ?: "./Dockerfile"

  echo "Building Docker image ${imageName}:${imageTag}"
  sh """
    docker build -t ${imageName}:${imageTag} \
      -f ${dockerfilePath} .
    docker push ${imageName}:${imageTag}
  """
```

```
  return "${imageName}:${imageTag}"
}
```

These shared resources could be imported into pipelines as needed, allowing teams to reference standardized components rather than maintaining copies in each repository:

Listing 5.5. *Example of using a shared resource in a pipeline*

```
// Example of using a shared Dockerfile in a pipeline
stage('Prepare Build Environment') {
  steps {
    script {
      // Import Dockerfile from shared library
      def resourcePath = 'org/example/docker/python-service/Dockerfile'
      def dockerfileContent = libraryResource resourcePath
      writeFile file: 'Dockerfile', text: dockerfileContent
    }
  }
}
```

The shared library approach provided a foundation for continuous improvement of CI processes while maintaining flexibility for service-specific customizations.

5.3.2 Automated Sprint Review Reporting

To improve visibility into development progress and align with the company's sprint-based workflow, an automated sprint review report generator was implemented as a Jenkins job. This tool addressed the challenge of manually collecting and formatting version information for regular sprint reviews.

The automated report generator provided several significant benefits to the development workflow. The automation eliminated the need for manual collection of version information across multiple repositories, saving approximately 2-3 hours of manual work per sprint cycle. The generator ensured a standardized format for sprint reports, improving readability and comparability over time, which facilitated better tracking of development progress across sprint cycles. The reports provided managers and team members with clear visibility into what changes had been deployed in each sprint, enabling more informed discussions during sprint reviews and retrospectives. Furthermore, the automated reports created a searchable historical record of service evolution over time, providing valuable documentation for understanding the platform's development trajectory.

Listing 5.6 shows an excerpt from the sprint report generation job:

Listing 5.6. *Sprint review report generation job excerpt*

```
// Example of sprint review report generation job
pipeline {
  agent any
  parameters {
    string(name: 'SPRINT_NUMBER', defaultValue: '',
           description: 'Sprint number')
    string(name: 'START_DATE', defaultValue: '',
           description: 'Sprint start date')
    string(name: 'END_DATE', defaultValue: '',
           description: 'Sprint end date')
  }
  stages {
    stage('Generate Report') {
      steps {
        script {
          def repositories = ['service-a', 'service-b',
                              'service-c', 'service-d']
          def report = "# Sprint ${params.SPRINT_NUMBER}" +
                       " Review Report\n\n"
          report += "Period: ${params.START_DATE}" +
                    " to ${params.END_DATE}\n\n"
          report += "## Version Updates\n\n"

          repositories.each { repo ->
            // Get latest version in date range
            def versions = sh(
              script: """
                git -C /path/to/${repo} log \\
                  --after=${params.START_DATE} \\
                  --before=${params.END_DATE} \\
                  --oneline
              """,
              returnStdout: true
            ).trim()

            report += "### ${repo}\n"
            if (versions) {
              report += "- " + versions.replace('\n', '\n- ')
            } else {
              report += "No updates in this sprint"
```

```
            }
            report += "\n\n"
          }

          def fileName = "sprint-${params.SPRINT_NUMBER}-report.md"
          writeFile file: fileName, text: report
        }
      }
    }
  }
  post {
    success {
      archiveArtifacts artifacts: "sprint-*.md",
                       allowEmptyArchive: true
    }
  }
}
```

The generated reports were produced in Markdown format, making them easily shareable and readable across the organization.

5.3.3 Pipeline Reliability Improvements and Standardization Outcomes

Several targeted improvements were implemented to address reliability issues in the existing CI pipelines. Consistent error handling patterns were implemented across pipelines to improve failure diagnostics and reporting, while automated retry mechanisms were added for transient network-related failures that occasionally disrupted builds. Appropriate timeout settings were added to prevent builds from hanging indefinitely due to unexpected issues, and improved workspace cleanup procedures were implemented to prevent pollution between builds. These reliability improvements helped reduce pipeline failures caused by environmental factors and improved the diagnostics available when failures did occur.

In addition to these implemented changes, discussions were held with stakeholders regarding further CI pipeline enhancements. A key focus was improving static analysis capabilities through the evaluation of SonarQube as a more comprehensive static analysis solution. The discussions included identification of integration points within existing pipelines and assessment of resource requirements for implementation. Additionally, stakeholders developed a preliminary integration plan for future implementation. These comprehensive discussions resulted in a roadmap for future CI improvements that would build upon the standardization work already completed.

Overall, the incremental approach to CI pipeline standardization delivered several significant benefits. The centralization of common resources and configurations reduced maintenance overhead across the development team, while standardized error handling and retry mechanisms improved pipeline reliability. The automated reporting functionality enhanced visibility into development progress, and the work created a foundation for future improvements like enhanced static analysis. Importantly, the changes were implemented with minimal disruption to ongoing development activities, allowing teams to gradually adopt standardized approaches while maintaining productivity.

5.4 Implementation Challenges and Solutions

The implementation of DevOps practices at Treon encountered various challenges that required creative solutions and iterative refinement. These challenges spanned technical, organizational, and process domains, reflecting the multifaceted nature of DevOps transformation in a microservice-based IoT environment.

5.4.1 Technical Challenges

The implementation of DevOps practices at Treon encountered various technical challenges that required creative solutions and iterative refinement. These challenges spanned authentication, version management, secret sharing, certificate automation, and monitoring integration.

Integrating ArgoCD with the company's existing authentication system presented the first significant technical hurdle. The default ArgoCD authentication mechanisms didn't align with Treon's existing Azure Active Directory-based identity management, creating potential security gaps and user management overhead. To address this, we implemented Azure AD integration for ArgoCD, allowing users to "Login with Azure." This integration involved configuring ArgoCD to use Azure AD as an authentication provider, setting up appropriate RBAC (Role-Based Access Control) to match organizational roles, and ensuring proper permission mapping between Azure AD groups and ArgoCD roles. This integration ultimately allowed seamless single sign-on for developers and operations staff while maintaining appropriate access controls aligned with organizational structures.

A significant challenge emerged related to how ArgoCD handles container image updates. ArgoCD, following GitOps principles, doesn't automatically detect changes to container images using the 'latest' tag. This is by design, as GitOps requires explicit version tracking in Git. However, this created difficulties in implementing an efficient deployment workflow. Instead of relying on the 'latest' tag, we implemented a workflow that used specific version tags (e.g., 'v1.2.3') for all container images, updated these version tags in the

Git repository's Helm chart values files whenever a new version was built, and triggered ArgoCD synchronization based on the Git changes, ensuring proper version tracking. This approach maintained the GitOps principle that Git remains the source of truth while enabling efficient deployments with proper versioning.

The single-tenant architecture with separate client namespaces created challenges for managing shared secrets. Certain secrets, such as service credentials, needed to be shared across multiple namespaces while maintaining synchronization when updates occurred. We addressed this through implementation of the Kubernetes Reflector extension to automatically sync secrets across namespaces:

```
metadata:
  annotations:
    reflector.v1.k8s.emberstack.com/reflection-allowed: "true"
    reflector.v1.k8s.emberstack.com/reflection-auto-enabled: "true"
```

This annotation-based approach allowed the Reflector controller to detect secrets and copy them to other namespaces, maintaining them in sync across the cluster automatically.

Automating SSL certificate management presented unexpected challenges with service configuration. While Let's Encrypt SSL certificates were set up for automatic renewal every 90 days, the MQTT service did not automatically reload the new certificates, potentially causing connection failures after renewals. Initially, a manual process was established where team members would reload the MQTT configuration after certificate renewals. An improved solution was designed using Certbot's deploy-hook feature:

```
certbot renew --deploy-hook "systemctl reload mosquitto"
```

This approach would enable automatic reloading of the service configuration after certificate renewal, eliminating the need for manual intervention.

As detailed in the monitoring section, integrating NATS with Prometheus presented unique technical challenges. NATS does not natively expose metrics in a Prometheus-compatible format, and the default configuration did not expose the necessary metrics endpoints. Custom ServiceMonitor resources were created and the NATS Helm chart was configured with specific exporter settings. Additionally, Prometheus scrape configurations required customization to properly collect metrics from the NATS exporter. This solution required careful coordination between multiple components but ultimately delivered comprehensive visibility into the message bus, a critical component of the microservice architecture.

The challenges encountered during our implementation mirror those observed in broader research on cloud-edge deployments. Veith et al. [22] identified that scaling deployment strategies to large infrastructure topologies presents significant computational challenges, with many traditional approaches failing to compute placements within reasonable timeframes as the number of resources increases. Their research demonstrates that infrastructure-aware optimization strategies that reduce the search space can maintain deployment quality while dramatically improving scalability. This aligns with our experience implementing ArgoCD across multiple namespaces, where optimization of the deployment process was necessary to handle the increasing complexity of our infrastructure.

5.4.2 Organizational Challenges

Beyond technical obstacles, the DevOps transformation at Treon encountered significant organizational challenges that required careful change management and cultural adaptation.

As with many DevOps transformations, knowledge sharing and adoption presented substantial challenges. GitOps practices and the use of ArgoCD represented a significant shift in how deployments were performed, with team members accustomed to manual deployment processes needing to adapt to the declarative, Git-centric approach. To address this, a targeted knowledge transfer approach was implemented that involved conducting meetings with key stakeholders to demonstrate the new system, providing hands-on demonstrations of basic workflows, creating and distributing comprehensive documentation for reference, and establishing support channels for questions and troubleshooting. This focused approach ensured that team members had both the initial exposure and the ongoing resources needed to successfully adopt the new practices.

Finding the right balance between automation and control proved challenging across different aspects of the system. Different components and environments had varying requirements for automation versus manual control, with some aspects of the system benefiting from full automation while others required careful oversight and approval. We implemented a hybrid approach that fully automated non-critical or development environments, maintained manual approval gates for sensitive production changes, created clear documentation of which processes were automated and which required intervention, and established appropriate audit trails for both automated and manual processes. This balanced approach allowed the organization to benefit from automation efficiencies while maintaining appropriate control over critical systems.

5.5 Implementation Insights and Lessons Learned

The challenges encountered during implementation provided valuable insights into DevOps transformation in IoT environments that extend beyond the specific technical solutions deployed. Many challenges involved integrating systems that weren't designed to work together natively, such as ArgoCD with Azure AD and NATS with Prometheus. Finding or creating appropriate integration points proved essential for successful implementation, highlighting the importance of evaluating integration complexity early in the planning process.

The experience demonstrated that while GitOps principles are straightforward in theory, implementing them in real-world environments required creative solutions, particularly around image versioning and secret management. The gap between theoretical understanding and practical implementation necessitated iterative problem-solving and willingness to adapt standard approaches to fit organizational constraints. Finding the balance between automation and human error emerged as critical, particularly in production environments where the consequences of automation failures could be significant.

Even with well-designed technical solutions, successful adoption required focused knowledge transfer efforts tailored to the organization's needs. The technical implementation represented only part of the transformation challenge, with organizational change management and team education playing equally important roles in achieving lasting improvements. As new challenges emerged throughout the implementation, the infrastructure-as-code approach allowed for iterative improvements and consistent application of solutions across the environment, demonstrating the value of declarative configuration management.

These lessons reinforced that successful DevOps transformation requires attention not only to tools and technologies but also to people, processes, and organizational factors. The solutions developed at Treon demonstrate how theoretical DevOps principles can be pragmatically adapted to the specific needs of microservice-based IoT environments.

6. DISCUSSION

This chapter discusses the implications and significance of the research findings. It connects the results to broader IoT SaaS challenges, critically analyzes the implementation approaches, examines lessons learned, and addresses the limitations of this study along with directions for future research.

6.1 Connecting Findings to Real-World IoT SaaS Challenges

The technical implementations described in this thesis address fundamental challenges that extend far beyond Treon's specific context. IoT SaaS companies face a unique combination of operational complexities: they must manage distributed microservice architectures while serving clients who expect enterprise-level reliability, yet they often lack the deep DevOps expertise that traditional software companies have developed over decades. The solutions implemented at Treon demonstrate how established DevOps practices can be adapted to address these IoT-specific challenges while accounting for the architectural constraints inherent in single-tenant IoT deployments.

The single-tenant architecture challenge represents one of the most significant differentiators between traditional SaaS and IoT SaaS environments. While conventional multi-tenant SaaS platforms can leverage deployment strategies like canary releases to gradually expose new features to subsets of users, IoT SaaS platforms with dedicated client instances face an all-or-nothing deployment scenario for each client. This architectural constraint, which emerged clearly during our deployment strategy evaluation, reflects a broader industry challenge where IoT companies must balance client isolation requirements with operational efficiency.

Our findings regarding the unsuitability of canary deployment in single-tenant environments align with broader industry observations about the need for IoT-specific deployment approaches. The namespace-level rolling update strategy that proved most effective at Treon addresses a gap in existing deployment literature, which typically focuses on traditional multi-tenant architectures. This adaptation demonstrates how IoT companies must often modify standard DevOps practices to accommodate their unique operational requirements.

The GitOps implementation with ArgoCD addressed another common challenge in the IoT industry: the need to provide deployment visibility and control to teams with varying levels of DevOps expertise. IoT companies often employ engineers with strong embedded systems and hardware backgrounds who may have limited experience with complex CI/CD tooling. The web-based interface and declarative configuration approach of ArgoCD successfully bridged this knowledge gap, enabling broader team participation in deployment processes without requiring extensive command-line expertise.

Furthermore, the configuration drift detection capabilities implemented through ArgoCD address a critical reliability concern in IoT environments where clients expect consistent service behavior. Unlike traditional software applications where minor configuration inconsistencies might go unnoticed, IoT platforms often manage critical infrastructure where deployment inconsistencies can impact physical operations. The automatic detection and visualization of configuration drift provides IoT operators with the confidence that their systems maintain consistency across client environments.

The CI pipeline standardization efforts reflect another industry-wide challenge as IoT companies scale their microservice architectures. The proliferation of repositories and services that typically accompanies IoT platform growth creates maintenance overhead that can significantly impact development velocity. Our shared library implementation demonstrates how thoughtful standardization can reduce this overhead while maintaining the flexibility required for diverse IoT service requirements.

6.2 Critical Analysis of Implementation Approaches

The selection and implementation of ArgoCD as the GitOps solution involved several trade-offs that merit critical examination. While ArgoCD provided significant benefits in deployment visibility and automation, the implementation also introduced new dependencies and learning requirements that organizations must carefully consider.

The primary strength of the ArgoCD approach lay in its ability to transform complex deployment processes into intuitive visual workflows. The web-based dashboard successfully democratized access to deployment information, enabling team members with varying technical backgrounds to understand and monitor deployment status. This accessibility proved particularly valuable in the IoT context where cross-functional teams often include hardware engineers, embedded software developers, and cloud platform specialists with different tooling preferences.

However, the ArgoCD implementation also revealed limitations that became apparent only during production use. The tool's handling of container image updates with dynamic tags required workflow modifications that initially created friction with existing CI processes. The necessity to implement specific versioning strategies and modify Jenkins pipelines demonstrated that GitOps adoption involves more than simply installing a new tool—it requires organizational commitment to new workflows and practices.

The integration challenges encountered with Azure Active Directory authentication highlighted another critical consideration for IoT companies evaluating GitOps tools. While ArgoCD's flexibility eventually enabled successful integration, the initial configuration complexity suggested that organizations should carefully evaluate authentication requirements during tool selection rather than assuming seamless integration with existing identity management systems. Similarly, the secret management challenges addressed through the Kubernetes Reflector implementation revealed gaps in ArgoCD's native capabilities for complex multi-namespace environments. While the solution proved effective, it required additional tooling and complexity that organizations should factor into their implementation planning. This experience suggests that GitOps tools may require complementary solutions to fully address the operational requirements of complex IoT platforms.

Beyond the GitOps implementation itself, the systematic evaluation of deployment strategies revealed how architectural constraints and operational requirements significantly influence strategy selection in ways that theoretical comparisons may not fully capture. The decision-making process at Treon demonstrates the importance of evaluating deployment strategies within their specific operational context rather than relying solely on theoretical performance characteristics.

The architectural constraint of single-tenant deployments fundamentally altered the value proposition of different deployment strategies. Canary deployment, which offers significant benefits in traditional multi-tenant environments, became impractical when each client operates within an isolated namespace. This constraint forced a reevaluation of deployment strategy literature that typically assumes multi-tenant architectures, highlighting the need for IoT-specific deployment guidance.

Resource constraints played a more significant role in strategy selection than initially anticipated. While blue-green deployment demonstrated promising zero-downtime capabilities during proof-of-concept testing, the resource requirements for maintaining duplicate environments across multiple client namespaces proved prohibitive in the production context. This experience illustrates how theoretical deployment strategy benefits must be balanced against practical resource limitations, particularly in cost-sensitive IoT environments where margins may be tighter than traditional software businesses.

The operational complexity assessment revealed that deployment strategy selection must account for team capabilities and organizational change management capacity. Blue-green deployment required coordination across multiple systems and processes that exceeded the organization's immediate change management capacity, even though the technical implementation proved feasible. This finding suggests that deployment strategy selection should incorporate organizational readiness assessments alongside technical evaluations.

Comparison with Empirical Research Findings

Our implementation experience at Treon can be contextualized against empirical research findings. Rakshit and Banerjee's study [16] on deployment strategies in Kubernetes environments offers a comparative benchmark for our practical implementation decisions.

Their research identified Blue-Green Deployments as providing the highest availability and lowest latency during transitions, which aligns with our proof-of-concept implementation findings. However, our experience highlighted additional challenges not fully captured in controlled experimental settings:

- **Resource Overhead:** While our Blue-Green implementation demonstrated promising zero-downtime capabilities, the resource requirements for maintaining duplicate environments proved prohibitive in our production context, confirming Rakshit and Banerjee's resource efficiency concerns.
- **Data Synchronization:** The experimental studies focused primarily on performance metrics but underrepresented the data synchronization challenges we encountered during our Blue-Green implementation, particularly for stateful services in our IoT environment.
- **Operational Complexity:** The empirical research noted operational complexity as a limiting factor for Blue-Green Deployments, which directly aligned with our implementation experience and ultimately influenced our strategy selection.

Our decision to implement a namespace-level rolling update strategy rather than proceeding with Blue-Green deployment reflects a practical compromise between theoretical performance advantages and real-world operational constraints. While Rakshit and Banerjee's research identified Blue-Green as offering superior performance characteristics in controlled environments, our implementation context required balancing these benefits against resource constraints and operational complexity.

6.3 Lessons Learned from Work at Treon

The DevOps transformation at Treon provided several key insights that extend beyond the specific technical implementations to encompass broader organizational and process considerations. These lessons learned offer valuable guidance for other IoT companies considering similar transformations.

The most significant technical insight involved the complexity of integrating modern DevOps tools with existing organizational infrastructure. While tool documentation and community resources provided excellent guidance for standard implementations, the reality of production environments often involves unique constraints and legacy systems that require creative solutions. The authentication integration challenges with ArgoCD and the NATS monitoring configuration difficulties exemplified how seemingly minor technical details can become significant implementation hurdles when working within established organizational infrastructure.

The importance of incremental implementation emerged as another critical lesson. The phased approach to GitOps implementation, beginning with proof-of-concept deployments and gradually scaling to production environments, proved essential for building organizational confidence and identifying integration challenges early. This approach allowed for course corrections and refinements that would have been much more difficult to implement in a comprehensive rollout. IoT companies considering similar transformations should resist the temptation to implement all improvements simultaneously and instead focus on building momentum through incremental successes.

Knowledge transfer requirements proved more substantial than initially anticipated. The technical implementation of ArgoCD and GitOps workflows represented only part of the transformation challenge. Ensuring that team members understood not just how to use the new tools but why the workflows had changed required dedicated effort and multiple communication channels. The most effective knowledge transfer occurred through hands-on demonstrations and collaborative problem-solving sessions rather than through documentation alone.

The organizational factor that most influenced implementation success was leadership support for workflow changes. While technical teams quickly recognized the benefits of improved deployment processes, the transition required changes to established procedures and responsibilities that could have created resistance without clear organizational commitment to the transformation. The success of the ArgoCD implementation depended as much on organizational change management as on technical execution.

The relationship between automation and control emerged as a nuanced balance that required careful consideration. While automation provided significant benefits in deployment consistency and reliability, team members initially expressed concern about reduced visibility and control over deployment processes. The ArgoCD web interface successfully addressed these concerns by providing enhanced visibility compared to previous manual processes, but this outcome required thoughtful design of the GitOps workflows to maintain appropriate human oversight.

The monitoring infrastructure development revealed that comprehensive observability requires more than simply installing monitoring tools. The specialized dashboards for Kubernetes pod lifecycle, NATS messaging, ArgoCD deployments, and Azure resource utilization addressed specific operational needs that generic monitoring solutions could not adequately serve. This experience demonstrated the importance of designing monitoring solutions around specific operational workflows rather than assuming that general-purpose dashboards will provide adequate visibility.

6.4 Practical Implications for IoT SaaS Companies

Organizations considering similar DevOps transformations should approach implementation with careful planning and realistic expectations about timeline and resource requirements. Based on the Treon experience, successful implementation requires approximately three to four months for the technical implementation phase, followed by an additional two to three months for full organizational adoption and workflow optimization.

The most critical success factor involves securing organizational commitment to workflow changes before beginning technical implementation. GitOps represents more than a tool change—it requires fundamental shifts in how teams approach deployment processes and configuration management. Organizations should invest time in change management and communication before beginning technical work to ensure smooth adoption.

Technical planning should prioritize integration requirements early in the evaluation process. Authentication integration, monitoring tool compatibility, and secret management approaches often present more complex challenges than core tool functionality. Organizations should prototype these integrations during the evaluation phase rather than assuming they can be resolved during implementation.

Resource planning should account for both direct implementation effort and the learning curve for team members adopting new workflows. While the core technical implementation may require dedicated effort from two to three team members, successful adoption requires broader team engagement and training. Organizations should plan for reduced productivity during the transition period as team members adapt to new processes.

The selection of deployment strategies should prioritize organizational readiness and resource constraints over theoretical performance characteristics. While advanced strategies like blue-green deployment offer significant benefits, they may not be practical for organizations with limited resources or complex data synchronization requirements. A phased approach that begins with improved versions of existing strategies and gradually evolves toward more sophisticated approaches often provides better long-term outcomes.

With these implementation considerations in mind, organizations can expect to realize several categories of benefits from similar DevOps transformations, though the timeline and magnitude of benefits will vary based on organizational context and implementation approach. The most immediate benefits typically involve improved deployment reliability and enhanced visibility into deployment processes, which can reduce time spent troubleshooting deployment issues within the first month of implementation.

Medium-term benefits include increased deployment frequency and reduced manual effort in deployment processes. The Treon implementation enabled more frequent deployments through simplified processes and enhanced confidence in deployment reliability. These improvements typically become apparent within two to three months of implementation as teams become comfortable with new workflows.

Long-term benefits involve improved collaboration between development and operations teams and enhanced ability to respond quickly to production issues. The Git-based workflow facilitates better communication about configuration changes and provides improved audit trails for troubleshooting. These collaborative benefits often require several months to fully develop as teams adapt their communication patterns around new tools and processes.

The resource investment required for implementation includes both direct technical effort and indirect costs associated with learning and workflow adaptation. Organizations should budget for tool licensing costs, infrastructure resources for GitOps tools, and dedicated staff time for implementation and training. The Treon experience suggests that implementation costs are typically recovered within six to twelve months through improved operational efficiency.

Training and knowledge transfer costs represent a significant component of total implementation investment. While technical implementation may require specialist DevOps knowledge, successful adoption requires broader team training and support. Organizations should plan for multiple training sessions, documentation development, and ongoing support during the transition period.

The return on investment for DevOps transformation in IoT environments typically exceeds that of traditional software environments due to the critical nature of IoT deployments and the potential impact of deployment failures on client operations. However, organizations should expect longer payback periods than typical software deployments due to the additional complexity of IoT architectures and single-tenant deployment models.

6.5 Challenges & Limitations

The implementation at Treon encountered several technical limitations that reflect broader challenges in applying DevOps practices to production IoT environments. Working within an active production environment imposed constraints on experimental approaches and required conservative implementation strategies that prioritized stability over optimization. These constraints limited the ability to test alternative approaches or implement more aggressive automation strategies that might have provided additional benefits.

Integration challenges with existing systems represented a more significant limitation than initially anticipated. The NATS monitoring integration difficulties and the ArgoCD authentication requirements highlighted how production environments often involve complex interdependencies that are difficult to predict during planning phases. These integration challenges required additional implementation time and creative solutions that may not be necessary in greenfield deployments.

The tools and technologies selected for implementation, while appropriate for Treon's requirements, imposed their own limitations on possible approaches and optimizations. ArgoCD's handling of dynamic container tags required workflow modifications, while the Kubernetes Reflector solution for secret management introduced additional complexity. Organizations should carefully evaluate tool limitations during selection rather than assuming they can be resolved during implementation.

The single-tenant architecture that influenced deployment strategy selection also created limitations in applying standard DevOps practices and benchmarking against industry best practices. Many DevOps tools and strategies assume multi-tenant architectures, requiring adaptations that may not be well-documented or supported by vendor resources. This limitation suggests that IoT companies may need to invest more heavily in custom solutions and internal expertise than traditional software companies.

Beyond these technical limitations, the constructive research approach employed in this thesis introduced methodological constraints that affect the generalizability of findings. The focus on a single organizational context, while enabling deep implementation insights, limits the ability to validate findings across different organizational cultures, technical architectures, and business requirements.

Data collection constraints imposed by working within a production environment limited the availability of controlled experimental conditions and comprehensive performance metrics. While the implementation provided qualitative insights into deployment improvements and team workflow changes, quantitative performance comparisons were limited by the need to prioritize operational stability over experimental measurement. Future research could benefit from more structured measurement approaches designed from the beginning of implementation.

The timeline constraints of thesis research limited the ability to evaluate long-term outcomes and identify potential issues that might emerge after extended operation. The six-month implementation and evaluation period provided insights into immediate and medium-term impacts but could not assess the sustainability of improvements or identify optimization opportunities that might become apparent over longer operational periods.

Potential bias in evaluation may have influenced the assessment of implementation outcomes, as the researcher was directly involved in implementation and had a vested interest in demonstrating successful outcomes. While efforts were made to collect objective feedback and identify limitations, the researcher's dual role as implementer and evaluator may have influenced the interpretation of results.

The generalizability limitations of findings reflect the specific context of Treon's architecture, organizational culture, and business requirements. While the lessons learned provide valuable insights for similar organizations, they may not apply directly to IoT companies with different architectural patterns, organizational structures, or operational requirements. Organizations considering similar implementations should adapt rather than directly replicate the approaches described in this research.

6.6 Future Research Directions

The DevOps transformation implemented at Treon opens several avenues for future research that could extend the understanding of DevOps practices in IoT environments and address limitations identified in this study. These research directions reflect both technical opportunities for advancement and methodological improvements that could strengthen future investigations.

Long-term performance evaluation represents one of the most important areas for future research. While this study demonstrated short-term improvements in deployment processes and team workflows, extended operational evaluation could provide insights into the sustainability of benefits and identify optimization opportunities that emerge over time. A longitudinal study following IoT DevOps implementations over twelve to twenty-four months could reveal patterns in operational efficiency, cost savings, and organizational adaptation that are not apparent in shorter-term evaluations.

Comparative studies across multiple IoT organizations could address the generalizability limitations of this research by examining how different organizational contexts, architectural patterns, and business requirements influence DevOps implementation outcomes. Such studies could identify critical success factors that transcend individual organizational contexts and develop more robust guidelines for IoT DevOps transformation.

The security implications of GitOps implementation in IoT environments represent an important area for future investigation. While this study touched on security benefits such as credential isolation and audit trail improvements, comprehensive security evaluation could examine how GitOps practices affect IoT-specific security challenges such as device management, edge computing security, and compliance requirements in regulated industries.

Advanced deployment strategies for single-tenant IoT architectures could build on the deployment strategy evaluation conducted in this research. Future work could explore hybrid approaches that combine elements of different deployment strategies or investigate how emerging technologies like service mesh or edge computing platforms could enable new deployment patterns specifically optimized for IoT environments.

The integration of artificial intelligence and machine learning into IoT DevOps workflows represents an emerging area with significant potential for research and practical application. Future studies could examine how AI-driven monitoring, automated anomaly detection, and predictive deployment optimization could enhance the DevOps practices described in this thesis.

Cross-industry analysis could extend this research by examining how DevOps practices in IoT environments compare to implementations in other domains such as financial services, healthcare, or manufacturing. Such analysis could identify domain-specific requirements and best practices that could inform more targeted DevOps approaches for different IoT application areas.

The role of edge computing in IoT DevOps workflows represents another important research direction as IoT architectures increasingly incorporate edge processing capabilities. Future research could examine how deployment strategies, monitoring approaches, and CI/CD pipelines need to evolve to support distributed edge-cloud architectures that are becoming common in IoT implementations.

Finally, the development of IoT-specific DevOps tools and platforms could build on the lessons learned about how existing tools need to be adapted for IoT use cases. Research into purpose-built DevOps solutions for IoT could address the integration challenges and architectural limitations identified in this study while providing more native support for IoT-specific requirements such as device management, edge deployment, and single-tenant architectures.

7. CONCLUSION

This research successfully addressed fundamental challenges in implementing DevOps practices within microservice-based IoT SaaS environments through a constructive study at Treon. The investigation focused on three critical research questions that guided the implementation and evaluation of comprehensive DevOps transformation initiatives.

Regarding the first research question on optimal deployment strategies for single-tenant IoT SaaS platforms, the evaluation revealed that namespace-level rolling updates provide the most practical balance between operational simplicity, resource efficiency, and deployment reliability. While canary deployment proved incompatible with isolated single-tenant architectures and blue-green deployment presented prohibitive resource overhead, the adapted rolling update approach addressed Treon's specific architectural constraints. This finding challenges conventional deployment strategy literature that primarily addresses multi-tenant environments, demonstrating the need for IoT-specific adaptations of standard DevOps practices.

The second research question examined how GitOps implementation could enhance deployment management and reliability. The ArgoCD implementation successfully transformed manual, error-prone deployment processes into automated, transparent workflows. Key improvements included enhanced visibility through web-based dashboards, automatic configuration drift detection, improved team accessibility to deployment processes, and streamlined operational procedures. The Git-based approach provided complete audit trails for configuration changes while enabling rapid rollbacks when issues occurred. These outcomes validate the applicability of GitOps principles in IoT environments while highlighting the importance of careful integration with existing authentication systems and infrastructure components.

Addressing the third research question, CI pipeline standardization through enhanced Jenkins shared libraries successfully reduced code duplication by approximately 60% across microservice repositories. The centralized configuration management approach improved maintainability, established consistent patterns for error handling, and reduced the specialized knowledge required to implement CI pipelines for new services. The automated sprint review reporting eliminated manual effort while improving visibility into development progress, demonstrating how targeted standardization efforts can yield substantial efficiency gains in distributed microservice architectures.

From a theoretical perspective, this research contributes to the academic understanding of DevOps practices in IoT environments by documenting the specific challenges and adaptations required for single-tenant IoT SaaS deployments. The findings extend existing DevOps literature, which predominantly focuses on traditional multi-tenant architectures, by demonstrating how architectural constraints fundamentally influence strategy selection and implementation approaches. The research provides empirical evidence that deployment strategy selection must account for organizational readiness and resource constraints in addition to theoretical performance characteristics.

The practical contributions of this work include concrete implementation patterns and architectural decisions that other IoT companies can adapt to their contexts. The ArgoCD configuration approaches, deployment strategy evaluation framework, and shared library patterns developed during this research provide actionable guidance for organizations facing similar transformation challenges. The documentation of implementation challenges and solutions offers valuable insights into the integration complexities that arise when implementing modern DevOps tools within established organizational infrastructure. The monitoring dashboards developed for Kubernetes pod lifecycle, NATS messaging, ArgoCD deployments, and Azure resource utilization demonstrate how operational visibility can be tailored to IoT-specific requirements.

Several important lessons emerged from this implementation experience that extend beyond technical considerations. The phased implementation approach proved essential for building organizational confidence and identifying integration challenges early, suggesting that incremental adoption provides better outcomes than comprehensive rollouts. Knowledge transfer requirements exceeded initial expectations, with hands-on demonstrations and collaborative problem-solving proving more effective than documentation alone. The balance between automation and human oversight required careful consideration, particularly in production environments where team members valued enhanced visibility alongside reduced manual effort.

Future research could extend this work in several promising directions. Long-term performance evaluation over twelve to twenty-four months would provide insights into the sustainability of observed benefits and reveal optimization opportunities that emerge over extended operation. Comparative studies across multiple IoT organizations could identify critical success factors that transcend individual organizational contexts and develop more robust guidelines for IoT DevOps transformation. Investigation of security implications, including how GitOps practices affect device management and edge computing security in regulated industries, represents another important research direction. Advanced deployment strategies specifically optimized for single-tenant IoT architectures could build upon the evaluation conducted in this research, potentially incorporating emerging technologies like service mesh implementations or edge computing platforms.

The integration of artificial intelligence and machine learning into IoT DevOps workflows presents significant opportunities for enhancing the practices described in this thesis. Future studies could examine how AI-driven monitoring, automated anomaly detection, and predictive deployment optimization could further improve operational reliability. Cross-industry analysis comparing IoT DevOps practices with implementations in financial services, healthcare, or manufacturing could identify domain-specific requirements and best practices. As edge computing becomes increasingly prevalent in IoT architectures, research examining how deployment strategies and monitoring approaches must evolve to support distributed edge-cloud architectures would provide valuable guidance for the industry.

This research demonstrates that successful DevOps transformation in IoT environments requires thoughtful adaptation of established practices to accommodate single-tenant architectures, careful integration with existing organizational infrastructure, and sustained attention to both technical implementation and organizational change management. The solutions developed at Treon provide evidence that IoT companies can achieve substantial improvements in deployment reliability, operational efficiency, and team collaboration through systematic application of GitOps principles, strategic deployment strategy selection, and targeted CI pipeline standardization. These findings offer both theoretical insights and practical guidance for organizations navigating the complexities of DevOps transformation in microservice-based IoT SaaS environments.

REFERENCES

[1] Florian Beetz and Simon Harrer. “GitOps: The Evolution of DevOps?” In: *IEEE Software* 39.4 (2022), pp. 70–75. DOI: 10.1109/MS.2021.3119106.

[2] Antonio Carzaniga et al. *A Characterization Framework for Software Deployment Technologies*. Tech. rep. CU-CS-857-98. University of Colorado, Department of Computer Science, Apr. 1998.

[3] Kenan Chen et al. “Quantifying the Impact of Staged Rollout Policies on Software Process and Product Metrics”. In: *Annual Reliability and Maintainability Symposium (RAMS)*. IEEE, 2022, pp. 1–6. DOI: 10.1109/RAMS51457.2022.9894011.

[4] Lianping Chen. “Continuous Delivery: Huge Benefits, but Challenges Too”. In: *IEEE Software* 32.2 (2015), pp. 50–54. DOI: 10.1109/MS.2015.27.

[5] Pavle Dakić. “Software compliance in various industries using CI/CD, dynamic microservices, and containers”. In: *Open Computer Science* 14 (2024), p. 20240013. DOI: 10.1515/comp-2024-0013.

[6] C. Ebert et al. “DevOps”. In: *IEEE Software* 33.3 (May 2016), pp. 94–100. DOI: 10.1109/MS.2016.68.

[7] F. Erich. “DevOps is Simply Interaction Between Development and Operations”. In: *Software Engineering Aspects of Continuous Development and New Paradigms of Software Production and Deployment*. Ed. by J.M. Bruel, M. Mazzara, and B. Meyer. Vol. 11350. Lecture Notes in Computer Science. Cham: Springer, 2019. DOI: 10.1007/978-3-030-06019-0_7.

[8] Eugenio Gaeta et al. “GATEKEEPER Platform: Secure Processing Environment for European Health Data Space”. In: *IEEE Access* 13 (2025), pp. 34627–34638. DOI: 10.1109/ACCESS.2025.3539559.

[9] Tomáš Kormaník and Jaroslav Porubän. “Exploring GitOps: An Approach to Cloud Cluster System Deployment”. In: *2023 21st International Conference on Emerging eLearning Technologies and Applications (ICETA)*. IEEE. 2023, pp. 318–323. DOI: 10.1109/ICETA61311.2023.10344182.

[10] Samiksha Kurrewar et al. “Streamlining Kubernetes Deployments through GitOps Methodologies”. In: *2025 IEEE International Students’ Conference on Electrical, Electronics, and Computer Science (SCEECS)*. IEEE, 2025, pp. 1–7. DOI: 10.1109/SCEECS64059.2025.10941164.

[11] Ramón López-Viana, Jessica Díaz, and Jorge E. Pérez. “Continuous Deployment in IoT Edge Computing: A GitOps implementation”. In: *2022 17th Iberian Confer-*

ence on Information Systems and Technologies (CISTI). 2022, pp. 1–6. DOI: 10.23919/CISTI54924.2022.9820108.

[12] Antra Malhotra et al. “Evaluate Canary Deployment Techniques Using Kubernetes, Istio, and Liquibase for Cloud Native Enterprise Applications to Achieve Zero Downtime for Continuous Deployments”. In: *IEEE Access* 12 (2024), pp. 87883–87899. DOI: 10.1109/ACCESS.2024.3416087.

[13] Denis Medvedev and Konstantin Aksyonov. “The Development of a Simulation Model for Assessing the CI/CD Pipeline Quality in the Development of Information Systems Based on a Multi-Agent Approach”. In: *MATEC Web of Conferences*. Vol. 346. EDP Sciences, 2021, p. 03095. DOI: 10.1051/matecconf/202134603095.

[14] Abbas Mehdi et al. “Unleashing the Potential of Grafana: A Comprehensive Study on Real-Time Monitoring and Visualization”. In: *2023 14th International Conference on Computing Communication and Networking Technologies (ICCCNT)*. IEEE. 2023, pp. 1–8. DOI: 10.1109/ICCCNT56998.2023.10306699.

[15] Francis Borges Oliveira, Marco Di Felice, and Carlos Kamienski. “IoTDeploy: Deployment of IoT Smart Applications over the Computing Continuum”. In: *Internet of Things* 28 (2024), p. 101348. ISSN: 2542-6605. DOI: https://doi.org/10.1016/j.iot.2024.101348. URL: https://www.sciencedirect.com/science/article/pii/S2542660524002890.

[16] Haranath Rakshit and Subhasis Banerjee. “Scalability Evaluation on Zero Downtime Deployment in Kubernetes Cluster”. In: *2024 IEEE Calcutta Conference (CALCON)*. Authorized licensed use limited to: Tampere University. IEEE, 2024, pp. 1–5. DOI: 10.1109/CALCON63337.2024.10914046.

[17] Ramadoni, Ema Utami, and Hanif Al Fatta. “Analysis on the Use of Declarative and Pull-based Deployment Models on GitOps Using Argo CD”. In: *2021 4th International Conference on Information and Communications Technology (ICOIACT)*. IEEE, 2021, pp. 186–191. DOI: 10.1109/ICOIACT53268.2021.9563984.

[18] Per Runeson and Martin Höst. “Guidelines for conducting and reporting case study research in software engineering”. In: *Empirical Software Engineering* 14.2 (2009), pp. 131–164. DOI: 10.1007/s10664-008-9102-8.

[19] Moch. Yudha Erian Saputra et al. “Real-Time Server Monitoring and Notification System with Prometheus, Grafana, and Telegram Integration”. In: *2024 ASU International Conference in Emerging Technologies for Sustainability and Intelligent Systems (ICETSIS)*. IEEE. 2024, pp. 1808–1813. DOI: 10.1109/ICETSIS61505.2024.10459488.

[20] Daniel Sun et al. “Quantifying Failure Risk of Version Switch for Rolling Upgrade on Clouds”. In: *IEEE Fourth International Conference on Big Data and Cloud Computing*. 2014, pp. 175–182. DOI: 10.1109/BDCloud.2014.16.

[21] Vidyasagar Vangala. “Blue-Green and Canary Deployments in DevOps: A Comparative Study”. In: (Jan. 2025). ResearchGate Publication ID: 388490305. URL: https://www.researchgate.net/publication/388490305.

[22] Alexandre da Silva Veith, Marcos Dias de Assunção, and Laurent Lefevre. “Latency-Aware Strategies for Deploying Data Stream Processing Applications on Large Cloud-Edge Infrastructure”. In: *IEEE Transactions on Cloud Computing* 11.1 (2023), pp. 445–456. DOI: 10.1109/TCC.2021.3097879.

[23] Bo Yang et al. “Service Discovery based Blue-Green Deployment Technique in Cloud Native Environments”. In: *2018 IEEE International Conference on Services Computing (SCC)*. Authorized licensed use limited to: Tampere University. IEEE, 2018, pp. 185–192. DOI: 10.1109/SCC.2018.00031.