**Educational Administration: Theory and Practice**
2024, 30(5), 14579-14588
ISSN: 2148-2403
https://kuey.net/

**Research Article**

# Optimizing Application Migration and Scaling through GitOps Methodology

Fazia Fatima[1], Gaurav Tyagi[2]

[1*]Master of Technology (Computer Science Engg.) Sir Chhotu Ram Institute of Engineering and Technology, CCS University, Meerut, India fazia.fatima@gmail.com
[2]Professor, Department of CS & IT, Sir Chhotu Ram Institute of Engineering and Technology, CCS University, Meerut, India, gauravtyagi.ccsu@gmail.com

**Citation:** Fazia Fatima, Dr Gaurav Tyagi (2024), Optimizing Application Migration and Scaling through GitOps Methodology, Educational Administration: Theory and Practice, 30(5), 14579-14588
Doi: 10.53555/kuey.v30i5.7087

**ARTICLE INFO**

**ABSTRACT**

Though the prevalence of GitOps usage is at peak nowadays, still there are certain institutions that face challenges in incorporating the GitOps methodology. The concept of GitOps is still developing. This research explores GitOps' transformative impact on software deployment, emphasizing automation and version control. Through a case study, GitOps proves instrumental in streamlining continuous deployment, addressing challenges, and fostering collaboration. Beyond deployment, GitOps facilitates collaboration and traceability throughout the software development lifecycle. This paradigm shift offers increased efficiency, reliability, and adaptability, making it pivotal in modern software engineering. The compatibility with microservices architectures is demonstrated in a Spring Boot migration case study. GitOps, with its continuous improvement focus, guides teams toward integrated and responsive deployment processes in the evolving software landscape.

**Keywords—**GitOps, Kubernetes, Scaling, CI/CD, Migration, IaC

## I. INTRODUCTION

GitOps is a modern operational model that brings the principles of version control to infrastructure and application delivery. It was first introduced by Weaveworks in 2017 and has since gained popularity in the world of cloud-native development and operations. GitOps is a transformative model for managing Kubernetes clusters and cloud-native applications by positioning Git as the central source of truth [4]. It builds upon and extends the principles of Infrastructure as Code (IaC) and introduces a robust methodology for handling continuous delivery and automated updates in a cloud-native environment.

### A. Key Concepts [2,3,9]

Let's see some key concepts which the GitOps possess:

**a) Foundation in Version Control:** GitOps relies on version control systems, with Git being the primary choice. This ensures that changes to both infrastructure configurations and application code are tracked, recorded, and managed effectively.

**b) Extension of IaC Principles:** GitOps is an evolution of Infrastructure as Code (IaC) principles. It emphasizes representing not only infrastructure configurations but also the entire system's desired state in code.

**c) Continuous Delivery and Automation:** One of the primary objectives of GitOps is to facilitate continuous
delivery by utilizing Git's pull request and merge mechanisms. This allows for automated updates to both infrastructure and application components.

**d) Desired State Definition in Git Repositories:** In GitOps, the entire system's desired state is defined in Git repositories. This includes configurations for Kubernetes clusters, cloud resources, and application code. Git becomes the single source of truth for the entire system.

**e) Automation for State Synchronization:** Automation tools are employed to ensure that the actual state of the system aligns with the desired state defined in Git. This involves continuous monitoring and reconciliation to maintain consistency.

### B. GitOps Workflow [12,13,14]

The GitOps workflow is a continuous and automated approach to manage and deploy infrastructure and applications. It revolves around using Git as the single source of truth, where the desired state of the system is defined, versioned, and stored in a Git repository. Automation tools then continuously reconcile the actual state of the system with this desired state, ensuring consistency and facilitating continuous delivery. Here's a detailed breakdown of the GitOps workflow:

**a) Desired State Definition:** The entire desired state of the system, including infrastructure configurations and application code, is codified and stored in a Git repository. This includes Kubernetes manifests, cloud infrastructure scripts, and other configuration files.

**b) Version Control:** Git serves as the version control system, tracking changes to the desired state over time. This provides a history of modifications, allows for collaboration through branches, and enables rollbacks to previous states if needed.

**c) Pull Requests and Reviews:** Changes to the desired state are proposed through Git pull requests. This collaborative process involves team members reviewing the changes, providing feedback, and ensuring that the proposed modifications align with best practices and compliance requirements.

**d) Approval and Merge**: Once the changes are reviewed and approved, they are merged into the main branch. The merge triggers the GitOps automation process.

**e) Continuous Monitoring:** Automation tools continuously monitor the Git repository for changes. These tools often include GitOps operators or controllers that are responsible for observing the repository and reacting to changes.

**f) Automated Synchronization:** Upon detecting changes, the GitOps automation tools automatically synchronize the actual state of the system with the newly defined desired state. This synchronization involves making the necessary adjustments to infrastructure and application configurations.

**g) Deployment to Target Environment:** The synchronized changes are deployed to the target environment, which could be a Kubernetes cluster, a cloud infrastructure, or any other runtime environment.

**h) Observability and Monitoring:** GitOps provides observability into the deployment process and system changes. Monitoring tools track the status of deployments, allowing teams to gain insights into the health and performance of the system.

**i) Rollback Capability:** In case of issues or undesired outcomes, GitOps allows for straightforward rollbacks. Reverting to a previous known good state is achieved by simply rolling back the changes in the Git repository.

**j) Notifications and Reporting:** GitOps tools often provide notifications and reporting mechanisms to keep teams informed about the status of deployments, changes, and any potential issues. This enhances communication and transparency across the organization.

While the DevOps approach offers numerous advantages, the implementation of Continuous Integration (CI), Continuous Delivery (CDE), and Continuous Deployment (CD) poses significant challenges. Even after successfully instilling the CI, CDE, and CD culture within a project team, persistent challenges linger. Challenges encountered include issues related to architectures [6], tools [8], and the assimilation of new methodologies [1,9] and ensuring the security of the CI/CD pipeline [7].

## II. BACKGROUND

The backdrop for this research stems from the recognition of limitations within the existing application's scalability. The current architecture faces challenges in efficiently handling heavy request loads, especially during peak periods. Manual deployment practices have contributed to inconsistencies in the application environment, resulting in downtime and suboptimal user experiences.

To address these limitations, a strategic decision was made to migrate the existing application to the Spring Boot stack. Spring Boot, renowned for its microservices architecture, is selected for its capability to offer enhanced scalability and modularity. This decision aligns with the broader industry trend favouring microservices-based architectures for their ability to improve flexibility, maintainability, and responsiveness in the face of varying workloads.

The migration to Spring Boot is not merely a technological shift but a deliberate move towards adopting best practices in modern software engineering. The objective is to leverage the microservices architecture, allowing for more granular control over individual components and use GitOps for streamlined deployment processes, and dynamic scaling based on demand.

## III.OBJECTIVES AND CURRENT CHALLENGES

In this study, we aim to tackle some big questions about making our application better. First, we want to understand why our application sometimes struggles to handle a lot of users and how we can make it work better. We're also looking into why we decided to move our application to something called the Spring Boot stack. We believe this move can help our application run smoother and faster.

As part of this journey, we're checking out something called microservices and how they fit with Spring Boot. These are fancy terms, but we're basically exploring new ways of building our application that can make it more flexible and responsive. Another important aspect we're digging into is GitOps, a method that helps us manage our application changes and updates more smoothly.

Now, let's talk about the problems we're facing. Our current way of doing things sometimes leads to confusion because our application doesn't always act the same way in different situations. The process of putting our application out there can also be slow and has room for errors. Plus, we don't always see clearly how our application is doing or catch issues early. We also find it tricky to adjust our resources when our workload changes, and working together in a team spread out in different locations can be a bit tough. By looking at these challenges and exploring new approaches, we hope to make our application work better for everyone.

## IV. IMPLEMENTATION

In this case study, we explore the challenges faced by a prominent company that relies on a monolithic application architecture. The primary issue revolved around the management of heavy request loads, particularly during peak times, where the existing manual deployment processes proved inadequate. The company struggled with inconsistent environments across teams, leading to operational challenges, downtime, and ultimately, suboptimal user experiences. To address these issues, the case study delves into the implementation of GitOps as a solution to streamline continuous deployment, providing insights into how this approach effectively mitigated the identified problems and enhanced the overall efficiency and reliability of the deployment pipeline.

### A. Problem: Inconsistent Environments Across Teams

Our journey towards streamlining continuous deployment began with a critical examination of the challenges posed by inconsistent environments across development and operations teams. Divergent setups and discrepancies in configurations often resulted in deployment issues and hindered the overall reliability of the application. Recognizing the need for a more cohesive and standardized approach, a decision was made to transition to the Spring Boot stack, known for its microservices architecture and scalability features.

### B. Strategizing the migration

The decision to migrate to Spring Boot was not arbitrary; it emerged from a careful evaluation of the existing technology stack's limitations and a desire to enhance scalability and modularity. We engaged in a thorough analysis of industry trends, best practices, and the specific needs of our application, ultimately concluding that a microservices-based approach with Spring Boot would align with our objectives[15,16]. This decision was backed by a cross-functional collaboration involving developers, operations teams, and key stakeholders.

The deployment uses GitOps for managing deployments. GitOps is a way to do Kubernetes cluster management and application delivery. GitOps works by using Git as a single source of truth for declarative infrastructure and applications. The fundamental shift the GitOps brings in while deploying applications is that the User Interface of the DevOps team is Git rather than a console on the server where the deployments are done. Hence, DevOps teams can use familiar tools to make pull requests to accelerate and simplify both application deployments and operations tasks to Kubernetes.

### To manage GitOps, following things have been put in place:

*a)* The entire system described declaratively: The application deployment is defined using a set of facts in a YAML file rather than a set of instructions to be executed directly on the server (e.g., using a series of shell scripts).
*b)* The desired system state is versioned in Git: As these declarations are kept in Git, we have a centralized location from which everything is derived and controlled. This simplifies rollbacks; you can utilize a Git revert to return to your previous application state. Leveraging Git's robust security features, you can also employ your SSH key to sign commits, ensuring strong security assurances regarding the authorship and origin of your code.
*c)* Approved changes that can be automatically applied to the system: Since the stated condition is stored in Git, any modifications to that condition can be automatically implemented in the application system. This is noteworthy because making changes to the system doesn't require cluster credentials, given that the definition of the system state exists externally. Moreover, this enables the segregation of "what to do" and "how to do it," akin to an interface definition in an object-oriented programming (OOP) language.
*d)* Software agents to ensure correctness and alert on divergence: Because the system's state is declared and managed through version control, software agents like Flux2 can monitor and identify disparities between the intended state and the current state in the cluster. Subsequently, the agent can ensure that the current state aligns with the desired state. For instance, if there's a commitment in Git to scale out a service by increasing the replication of pods from 2 to 3, but the server is running only 2 pods, the software agent can rectify this by adjusting the server's state to the desired configuration of 3 pods. This mechanism is also beneficial in the

event of human errors. For instance, if a server admin accidentally removes a service, the software agent can automatically recreate it by reconciling the server's state with the one declared in Git, ensuring the actual state aligns with the intended state.

### 1) Pull based Gitops Pipeline [10]

The application deployment uses a pull based GitOps pipeline and consist of two key components: a "Deployment Automator" that watches the image registry and a "Deployment Synchronizer" that sits in the cluster to maintain its state. Both the deployment automator and deployment synchronizer are a part of the Flux2 package .

At the center of our pull pipeline pattern is a single source of truth for manifests (or a the GitRepo). Developers push their updated code to the code base repository; where the change is picked up by the CI tool and ultimately builds a Docker image. The 'Deployment Automator' notices the image, pulls the new image from the repository and then updates its YAML in the config repo. The Deployment Synchronizer, then detects that the cluster is out of date, and it pulls the changed manifests from the config repo and deploys the new image to the cluster. The workflow is shown in the following image:

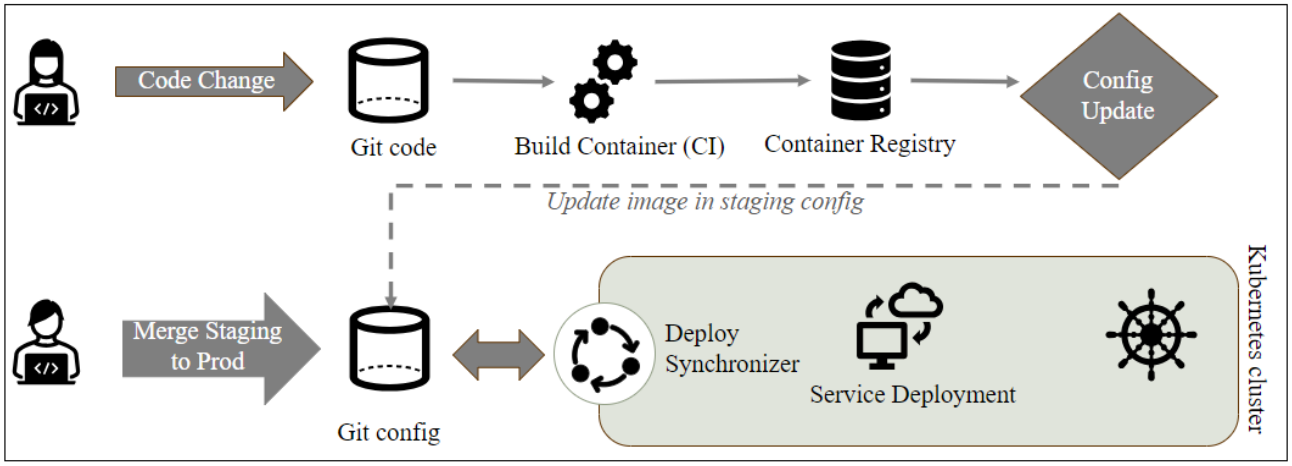


Fig1. GitOps In Action

### 2) Flux

Flux, a Kubernetes management tool, encapsulates key principles for effective orchestration. GitOps, at its core, defines infrastructure and applications declaratively in a version-controlled Git repository, automating deployment synchronization [17]. The application deployment system utilizes Flux (specifically Flux2) for keeping the Kubernetes clusters (AKS) in sync with sources of configuration in Git.
Below is the flux component diagram.

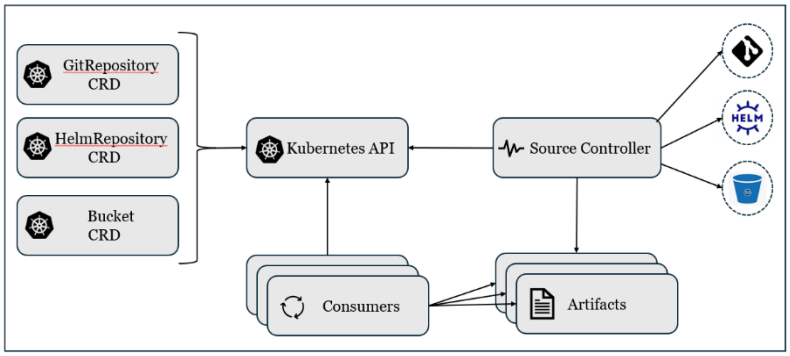


Fig 2: Flux Components

The source controller "monitors" a source (e.g., a git repository, helm repository etc) to detect any changes in them which triggers a reconciliation, thereby bringing the actual state on the cluster to the desired state mentioned in these repositories. The deployment system uses the GitControlers to watch Git Repository and Helm Controllers for watching Helm Repositories and Charts used in the project.

## C. Bootstrapping the application environment

1) Creating an Azure Cluster

*a)* Required Command Line Tools

- az - Azure CLI
- kubectl - Kubernetes command-line tool
- helm - Kubernetes package manager
- jq - command-line JSON processor

*b)* Create an Identity

- Create AD service principal.

```
az ad sp create-for-rbac --skip-assignment -o json > auth.json appId=$(jq -r ".appId" auth.json) password=$(jq -r ".password" auth.json)
```

- Use the appId from the previous command's output to get the objectId of the new service principal.

```
objectId=$(az ad sp show --id $appId --query "objectId" -o tsv)
```

- Create the parameter file that will be used in the Azure Resource Manager template deployment later.

```
cat <<EOF > parameters.json
{
  "aksServicePrincipalAppId": { "value":
"$appId" },
  "aksServicePrincipalClientSecret": {
"value": "$password" },
  "aksServicePrincipalObjectId": { "value":
"$objectId" },
  "aksEnableRBAC": { "value": false }
}
EOF
```

c) Deploy Components

- Download the Azure Resource Manager template and modify the template as needed.

```
wget
https://raw.githubusercontent.com/Azure/application-gateway-kubernetes-ingress/master/deploy/azuredeploy.json -O template.json
```

- Deploy the Azure Resource Manager template using az cli.

```
resourceGroupName="MyResourceGroup" location="westus2" deploymentName="ingress-appgw"

# create a resource group
az group create -n $resourceGroupName -l $location

# modify the template as needed az deployment group create  -g $resourceGroupName \
      -n $deploymentName \
      --template-file template.json \
      --parameters parameters.json
```

- Once the deployment finished, download the deployment output into a file.

```
az deployment group show -g $resourceGroupName -n $deploymentName -query "properties.outputs" -o json > deployment-outputs.json
```

D. Set up Application Gateway Ingress Controller

*1)* Setup Kubernetes Credentials

```
# use the deployment-outputs.json created after deployment to get the cluster name and resource group name

aksClusterName=$(jq -r ".aksClusterName.value" deploymentoutputs.json)
resourceGroupName=$(jq -r ".resourceGroupName.value" deploymentoutputs.json)

az aks get-credentials --resource-group $resourceGroupName --name $aksClusterName
```

*2)* Install AAD Pod Identity

```
kubectl create -f
```

```
https://raw.githubusercontent.com/Azure/aad-pod-identity/master/deploy/infra/deployment.yaml
```

*3)* Add the AGIC Helm repository

```
elm repo add application-gateway-kubernetes-ingress https://appgwingress.blob.core.windows.n et/ingress-azure-helm-package/ helm repo update
```

*4)* Setup Ingress Controller using Helm Chart

*a)* Create the following variables

```
applicationGatewayName=$(jq -r ".applicationGatewayName.value" deployment-outputs.json)
resourceGroupName=$(jq -r ".resourceGroupName.value" deployment-outputs.json)
subscriptionId=$(jq -r ".subscriptionId.value" deployment-outputs.json)
identityClientId=$(jq -r ".identityClientId.value" deployment-outputs.json)
identityResourceId=$(jq -r ".identityResourceId.value" deployment-outputs.json)
```

*b)* Download helm-config.yaml, which will configure AGIC

```
wget https://raw.githubusercontent.com/Azure/application-gateway-kubernetes-ingress/master/docs/examples/sample-helm-config.yaml -O helm-config.yaml
```

*c)* Edit helm-config.yaml

```
sed -i "s|<subscriptionId>|${subscriptionId}|g" helm-config.yaml
sed -i "s|<resourceGroupName>|${resourceGroupName}|g" helm-config.yaml
sed -i "s|<applicationGatewayName>|${applicationGate wayName}|g" helm-config.yaml
sed -i "s|<identityResourceId>|${identityResourceId}|g" helm-config.yaml
sed -i "s|<identityClientId>|${identityClientId}|g" helm-config.yaml
```

*d)* Install application gateway ingress controller

```
helm install -f helm-config.yaml applicationgateway-kubernetes-ingress/ingress-azure
```

### E. Deployment of the application.

The central component of GitOps lies in its CI/CD tooling, where the linchpin is the continuous deployment (CD) process that facilitates synchronization with Git clusters. Tailored for version-controlled systems and declarative application stacks, it ensures seamless integration. With every team member proficient in Git, the process of making pull requests becomes familiar. Consequently, Git becomes a powerful tool for expediting and streamlining application deployments onto Kubernetes.[11]

Here is a typical developer workflow for creating or updating a new feature: [11]

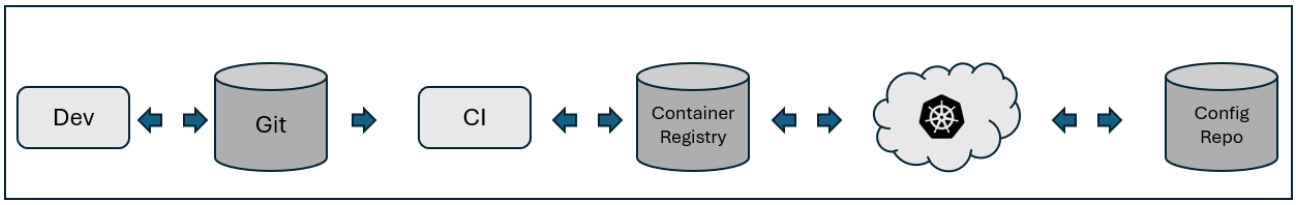


Fig3. GitOps Deployment Workflow

A pull request for a new feature is pushed to GitHub for review.

- The code is reviewed and approved by a colleague.
- After the code is revised and re-approved, it is merged to Git.
- The Git merge triggers the CI and build pipeline, runs a series of tests, and then eventually builds a new image and deposits the new image to a registry.
- The Deployment Automator watches the image registry, notices the image, pulls the new image from the registry, and updates its YAML in the config repo.
- The Deployment Synchronizer detects that the cluster is out of date, pulls the changed manifests from the config repo, and deploys the new feature to production.

Keeping the Git deployment strategy in mind, the following steps were performed for our application.

1) Prerequisite

- Kubectl context should point to the application Production environment.
- Install and setup flux in your local environment.

- An Azure Kubernetes cluster using steps mentioned in the section "Bootstrapping the application environment: Creating an Azure Cluster".
- Clone the application repository.
-

```
git clone https://github.com/application-gitops.git
```

*2)* Export your GitHub personal access token as an environment variable

```
export GITHUB_TOKEN=<your-token>
```

*3)* Run the bootstrap for a repository on your GitHub account

```
flux bootstrap github \     --components-extra=image-reflector-controller,image-automation-controller \
  --context=application-prod \
  --owner=application-ai \
  --repository= application-gitops \
  --branch=main \
  --personal \     --path=clusters/production
```

*4)* Watch the application getting deployed

```
watch flux get kustomizations
```

## RESULTS AND DISCUSSIONS

The transition from traditional application management practices to a GitOps-driven approach represents a significant paradigm shift. This section provides a detailed comparison of the application migration process before and after adopting GitOps, focusing on various aspects. By standardizing and automating the deployment process, GitOps enhances consistency, reduces the risk of errors, and provides robust version control and rollback capabilities. These improvements lead to more reliable and efficient operations, reducing downtime and improving overall system resilience.

We assessed the application based on specific metrics that measure key performance indicators as shown in Table 1

| **Aspect** | **Metric** | **Before Migration** | **After Migration** |
|---|---|---|---|
| Deployment Frequency [a] | Deployments per week | 1-2 | 10-20 |
| Downtime [b] | Mean time to Recovery (MTTR) | 4-6 hours | 30 minutes to 1 hour |
| Version Control [c] | Number of tracked changes | Limited | Comprehensive, all changes tracked |
| Automation [d] | Percentage of automated tasks | 30% | 90-100% |
| Rollback Capability [e] | Time to rollback | In Hours | In Minutes |
| Consistency [f] | Configuration drift incidents | Often | Rare |
| Change Failure Rate [g] | Percentage of failed changes | 15-20% | 5-10% |

[a] Deployment Frequency: This metric indicates how often changes can be deployed to production. GitOps practices, with their automated pipelines and version control, typically allow for more frequent deployments by reducing the manual overhead and risk associated with each deployment.

[b] Mean Time to Recovery (MTTR): MTTR measures the average time taken to recover from a failure. GitOps' quick rollback capabilities significantly reduce MTTR by allowing rapid reversion to a previous stable state stored in Git.

[c] Number of Tracked Changes: This metric measures the comprehensiveness of version control. GitOps practices ensure that every change, whether it's a configuration update or a new deployment, is tracked in Git, providing a complete history of the system's evolution.

[d] Percentage of Automated Tasks: Automation is a key advantage of GitOps. The percentage of automated tasks reflects the extent to which manual interventions are minimized, thus reducing errors and increasing efficiency.

[e] Time to Rollback: This metric measures the speed at which a system can be reverted to a previous state in case of issues. GitOps' use of Git for version control simplifies and speeds up rollbacks, ensuring minimal downtime.

[f] Configuration Drift Incidents: Configuration drift refers to discrepancies that arise over time between environments due to unmanaged changes. GitOps practices, which use Git as the single source of truth, significantly reduce the incidence of configuration drift by continuously reconciling the desired state with the actual state.

[g] Change Failure Rate: This metric indicates the percentage of changes that result in failures or issues. The structured and automated approach of GitOps reduces the change failure rate by ensuring that changes are thoroughly tested and reviewed before deployment.

## Discussion

The comparative analysis as outlined in the table 1 demonstrates clear improvements across several key metrics. This discussion explores the implications of these results, highlighting the benefits and addressing potential challenges associated with adopting GitOps.

a. **Deployment Frequency**: The increase in deployment frequency from 1-2 times per week to 10-20 times per week underlines the enhanced agility provided by GitOps. This improvement is attributed to the automation of deployment processes, which reduces manual effort and risk, allowing teams to deploy more frequently and respond faster to business needs. Increased deployment frequency also facilitates continuous integration and continuous deployment (CI/CD) practices, which are essential for rapid innovation and maintaining a competitive edge.
b. **Mean Time to Recovery (MTTR):** The reduction in MTTR from 4-6 hours to 30 minutes to 1 hour is significant, reflecting GitOps' robust rollback capabilities. In traditional setups, identifying and rectifying issues often involves manual troubleshooting, which can be time-consuming. In contrast, GitOps' use of Git as the source of truth enables rapid detection of discrepancies between the desired and actual states, allowing for quick remediation. This capability is crucial for maintaining high availability and minimizing downtime, thereby improving the overall user experience and operational resilience.
c. **Version Control and Change Tracking:** GitOps' emphasis on comprehensive version control and change tracking ensures that all changes to the system are documented and traceable. This practice enhances transparency and accountability, making it easier to understand the evolution of the system, audit changes, and comply with regulatory requirements. In contrast, traditional methods often suffer from incomplete documentation and inconsistent change tracking, which can complicate debugging and system maintenance.
d. **Automation and Consistency:** The high percentage of automated tasks in GitOps (90-100%) compared to traditional methods (around 30%) highlights the operational efficiency gains from adopting a GitOps model. Automation reduces human error, ensures consistent application of configurations, and streamlines complex workflows. The rarity of configuration drift incidents in GitOps environments further underscores the system's ability to maintain consistency across different stages of the software development lifecycle. This consistency is critical for environments where identical setups are required, such as staging and production.
e. **Change Failure Rate** The decrease in the change failure rate from 15-20% to 5-10% indicates that GitOps improves the reliability of deployments. The structured and automated processes inherent in GitOps facilitate thorough testing and review of changes before deployment, reducing the likelihood of issues. This reliability is essential for maintaining service quality and reducing the cost and effort associated with fixing failed deployments.

## Challenges and Considerations

While the benefits of GitOps are clear, the transition to this model is not without challenges. The initial setup of GitOps tools and processes requires significant investment in time and resources. Teams must be trained in new workflows, and existing systems may need to be reconfigured to fit the GitOps model. Additionally, managing secrets and sensitive data in Git repositories poses security challenges that need to be addressed with proper encryption and access controls. Moreover, the cultural shift towards a GitOps model requires buy-in from all stakeholders, including developers, operations, and management. Organizations must foster a culture of collaboration and continuous improvement to fully realize the benefits of GitOps.

## CONCLUSION

In this research, combining Spring Boot migration and adopting GitOps principles has transformed the deployment process. Shifting to a microservices architecture with Spring Boot brought a significant change to

continuous deployment, creating a more efficient and automated workflow. Continuous Integration/Continuous Deployment (CI/CD) pipelines played a key role. They removed manual steps, ensured consistency, and sped up the software delivery process.

The introduction of GitOps principles, particularly through the integration of Flux2, marked a fundamental shift in collaboration and visibility. Declarative configurations stored in Git repositories brought transparency, traceability, and version-controlled management to the deployment process. This transition not only improved communication and collaboration among cross-functional teams but also minimized the risk of errors and discrepancies. The GitOps approach, with its emphasis on declarative infrastructure and application code stored as version-controlled artifacts, enhanced overall visibility into the deployment pipeline, enabling stakeholders to track changes and understand the state of the system at any point in time.

Moreover, the outcomes of this strategic technological shift extend beyond streamlined processes and collaboration, manifesting in a substantial improvement in system reliability. The challenges associated with manual interventions and inconsistent deployments have been effectively mitigated. Declarative configurations stored in Git repositories have ushered in a level of consistency and reproducibility in the deployment process, contributing to a more resilient system architecture. This, in turn, has translated into reduced downtime, enhanced system reliability, and an overall improvement in the robustness of the software delivery pipeline.

The synergistic effects of Spring Boot migration and GitOps adoption have not merely addressed existing challenges but have laid the groundwork for a new era in continuous deployment practices. The outcomes presented in this research reflect a comprehensive and strategic approach to software delivery, characterized by automation, scalability, collaboration, and heightened system reliability. As organizations increasingly seek agile and efficient deployment strategies, the presented case study serves as a compelling model for embracing technological innovations that reshape and optimize contemporary software deployment practices.

## REFERENCES


1. M. K. A. Abbass, R. I. E. Osman, A. M. H. Mohammed, and M. W. A. Alshaikh, “Adopting continuous integration and continuous delivery for small teams,” Proc. Int. Conf. Comput. Control. Electr. Electron. Eng. 2019, ICCCEEE 2019, 2019, doi:
2. 10.1109/ICCCEEE46830.2019.9070849.
3. Alex Williams (May 11, 2023) “4 Core Principles of GitOps” from https://thenewstack.io/4-core-principles-of-gitops/ accessed 28-Aug2023.
4. Á. Sándor, "11 Reasons for Adopting GitOps", Blog.container solutions.com, 2019. [Online]. Available: https://blog.container solutions.com/why-adopt-gitops. accessed 02- Aug- 2023.
5. Alexis Richardson (August 21, 2018) “What is GitOps?” Weaveworks from https://www.weave.works/blog/what-is-gitops-really accessed September 02, 2023.
6. A. Gillis, "What is GitOps and why is It Important?", SearchITOperations. [Online]. Available: https://www.techtarget.com/searchitoperations/definition/GitOps. [Accessed: 11- Dec- 2020].
7. R. Bolscher and M. Daneva, “Designing software architecture to support continuous delivery and DevOps: A systematic literature review,” ICSOFT 2019 - Proc. 14th Int. Conf. Softw. Technol., pp. 27– 39, 2019, doi: 10.5220/0007837000270039.
8. M. Shahin, M. Ali Babar, and L. Zhu, “Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices,” IEEE Access, vol. 5, no. Ci, pp. 3909–3943, 2017, doi: 10.1109/ACCESS.2017.2685629.
9. A. Proulx, F. Raymond, B. Roy, and F. Petrillo, “Problems and
10. Solutions of Continuous Deployment: A Systematic Review,” no. December 2018, 2018, [Online]. Available: http://arxiv.org/abs/1812.08939.
11. L. Leite, C. Rocha, F. Kon, D. Milojicic, and P. Meirelles, “A survey of DevOps concepts and challenges,” ACM Computing Surveys. 2019, doi: 10.1145/3359981.
12. Alexis Richardson (August 07, 2017) “GitOps–Operations by pull request,” Weaveworks from https://www.weave.works/blog/ gitopsoperations-by-pull-request accessed September 02, 2023.
13. Ninad Desai “GitOps for Kubernetes” from https://dzone.com/refcardz/gitops-for-kubernetes accessed September 27, 2023.
14. GitLab “What is a GitOps workflow?” from https://about.gitlab.com/topics/gitops/gitops-workflow/ accessed November 03, 2023.
15. Weaveworks “Guide To GitOps” from https://www.weave.works/technologies/gitops/ accessed November 03, 2023.
16. Jobin Kuruvilla “GitOps Workflows explained” accessed from https://www.adaptavist.com/blog/gitops-workflows-explained on November 04,2023.
17. Raja Anbazhagan “10 Reasons Why You should use Spring Boot” accessed from https://springhow.com/why-use-spring-boot/ on July 18, 2023.

18. yashi_agarwal “10 Reasons to Use Spring Framework in Projects” accessed from https://www.geeksforgeeks.org/10-reasons-to-usespring-framework-in-projects/ on July 18, 2023.
19. Flux, “Core Concepts” accessed from https://fluxcd.io/flux/concepts/ on July 22, 2023.