UNIVERSITY OF TARTU
Faculty of Science and Technology
Institute of Computer Science
Computer Science Curriculum

Kyrylo Riazantsev

Study on GitOps paradigms

Master's Thesis (30 ECTS)

Supervisor(s): Bruno Rucy Carneiro Alves De Lima, MSc

Tartu 2024

Study on GitOps paradigms

Abstract: GitOps is a recently emerged concept that became popular due to the rise of DevOps methodology and a constant software development struggle to improve deployments and ensure their reliability. It offers an approach that utilizes Infrastructure as a Code and Git as a version control system to ensure that the state of an application and infrastructure always matches the one described in the Git repository (e.g., repository as a source of truth). There are two main ways to do GitOps: pull-based, in which an operator checks for updates in a Git repository, ensuring that the target matches the desired state, and push-based, which states that Git should notify the GitOps system about updates.
This thesis focuses on describing GitOps at its core, explaining why the above two paradigms were extracted from its implementation and going into detail about the principles of their work. It also proposes a prototypical implementation of both paradigms to measure their performance under various scenarios. It analyzes these measurements using the queueing theory model as a theoretical framework.

Keywords:
GitOps, DevOps, Continuous Integration, Continuous Delivery, Kubernetes, CI/CD, ArgoCD, Cloud Native

CERCS: P170 Computer science, numerical analysis, systems, control

Uuring GitOps paradigmade kohta

Lühikokkuvõte: GitOps on hiljuti tekkinud kontseptsioon, mis sai populaarseks tänu tõusule DevOps metoodika ja pideva tarkvaraarenduse võitlusega, et parandada juurutamise ja nende töökindluse tagamise eest. See pakub lähenemisviisi, mis kasutab infrastruktuuri kui koodi ja Git'i kui versioonihaldussüsteemi, et tagada rakenduse olek ja infrastruktuur vastab alati sellele, mis on kirjeldatud Git-repositooriumis (nt repositoorium kui tõe allikas). GitOps'i on võimalik teha peamiselt kahel viisil: pull-põhine, mille puhul operaator kontrollib uuendusi Git-repositooriumis, tagades, et sihtmärk vastab soovitud olekule, ja push-põhine, mille kohaselt peaks Git teavitama GitOps-süsteemi uuendustest. Käesolev lõputöö keskendub GitOps'i kirjeldamisele selle põhiolemuses, selgitades, miks eespool nimetatud kaks paradigmad on selle rakendusest välja võetud ja käsitletakse üksikasjalikult nende töö põhimõtteid. Samuti pakutakse välja mõlema paradigma prototüüpne rakendamine et mõõta nende toimivust erinevate stsenaariumide korral. Selles analüüsitakse neid mõõtmisi kasutades teoreetilise raamistikuna järjekorra teooria mudelit.

Võtmesõnad: GitOps, DevOps, Pidev integratsioon, Pidev kohaletoimetamine, Kubernetes, CI / CD, ArgoCD, Cloud Native

CERCS: P170 Arvutiteadus, arvutusmeetodid, süsteemid, juhtimine (automaatjuhtimisteooria)

Contents

1 Introduction

The term GitOps was first introduced by Weaveworks in 2017 [Wea]. The company describes a case where one of the engineers pushed a change that wiped out the entire system, consisting of multiple Kubernetes clusters. After that, the Weaveworks engineering team restored the system in about forty minutes because it was fully described in various Git config files. The system included the cluster and the app, monitoring, and other pieces. Whenever a change is made to the system, it is first committed and then allowed to propagate automatically into production [Wea21].

Narrowing this to a basic definition, we can state that GitOps is a paradigm or set of practices for managing and automating IT infrastructure and application configurations using Git as the single source of truth. It combines software development practices like version control, collaboration, compliance, and CI/CD (Continuous Integration/Continuous Deployment) workflows to manage infrastructure and application deployments. In GitOps, the system's desired states, including infrastructure provisioning, application deployment, and configuration updates, are declarative and stored in Git repositories. Changes to the system are made through updates to these files. They are automatically applied to the target environments using automated processes, ensuring that the actual state of the system matches the desired state as defined in the repository.

GitOps has recently emerged as a concept and set of tools to utilize this concept as a response to the rising usage of technologies, processes, and tools like DevOps, Kubernetes, IaC (Infrastructure as a Code), and Git. Though the practice and its software implementations are not yet an industry standard, it is already widely adopted by software development companies of different sizes domains, ranging from startups to large enterprises [arg23].

At the same time, due to its novelty (as mentioned earlier, the term was introduced in 2017) and the nature of any emerging field, it has not been extensively researched, described, and structured academically to the same depth as more established fields (for instance, the ones mentioned previously, like Kubernetes or Git, that were the foundation of GitOps).

Another reason might be that GitOps is a highly practical and tool-driven concept rooted in software development and deployment operations. It can be proved by the fact that its foundation was a single use-case that occurred in a commercial software development company. Most existing studies also prove the initial statement by focusing on the application of GitOps tools and how to utilize them in real-world software and infrastructure development rather than providing a formalized description. For instance, describing the application of GitOps in a Cyber-Physical Production Systems (CPPS) environment along with microservices in a real-world case study [KRM+23] or highlighting the benefits of GitOps adoption in Internet of Things Edge computing [LVDP22].

At the same time, the field's development reached a point where a comprehensive and generalized study on the topic is necessary. One of its expected outcomes is that

it might provide a robust theoretical framework for GitOps, helping to standardize its definitions, principles, and best practices. It would also aid in distinguishing GitOps from related methodologies like DevOps, clarifying its unique contributions, and guiding its implementation.

The foundation for this theoretical framework will be the paradigms (also commonly referred to as models or workflows) of GitOps. They describe how changes are propagated from the source control (e.g., Git) to the target environments (like production or staging). This principle of synchronizing changes between code stored in source control and the actual infrastructure is what GitOps is made for. Therefore, it significantly affects how GitOps tooling performs, depending on which way of doing this it chooses.

The thesis aims to give a general overview of GitOps methodology, how it emerged from other, already well-established practices, and to compare the aforementioned approaches to GitOps in terms of performance, scalability, security, and other metrics. We will also review actual implementations and tools that utilize these paradigms. By doing so, it will be possible to identify potential improvements and determine the feasibility of developing a new implementation that could reduce bottlenecks and enhance key characteristics essential for a GitOps system. Comparing approaches implies the development and evaluation of a prototype system alongside simple queue-theoretical formulas that help to model the performance of these systems.

By conducting this, it will be possible to provide a standardized theoretical framework for evaluating GitOps tools and approaches and establish practical guidelines for utilizing GitOps in different fields of cloud-native software development and infrastructure management.

This thesis is organized as follows: In Chapter 2 (background), the fundamental concepts and theoretical foundations of Git and DevOps will be explained, as those are essential definitions for understanding GitOps. In Chapter 3 (semantics of GitOps), we will go over GitOps itself and technologies that are tightly related to it, evolved from it, and utilize it, like IaC (infrastructure as code), ArgoCD, and Flux. In Chapter 4 (Implementation), we will extract discovered kinds of GitOps, describe them, propose prototypical implementation of those approaches, and their estimations on how they can perform based on computer science concepts. In Chapter 4 (experiments), comparisons and evaluations will be presented. Lastly, in Chapter 6 (conclusion), we will list the findings, lessons learned, and what future paths can be discovered related to the study topic.

2 Background

This chapter provides a foundational overview of the key concepts and tools that underpin the GitOps methodology and are used for its implementation, setting the stage for a deeper exploration of GitOps paradigms.

Terms are listed in chronological order, based on the time of their proposed definition and formalization (since one might argue that DevOps, for instance, existed long before the term came up). This approach will give a clear perspective on how each tool evolved from another and how the latter, GitOps, came up.

2.1 Git

Git is a distributed version control system (VCS) created by Linus Torvalds in 2005. Like most other VCSs, it records changes to a file or set of files over time, making it possible to return to specific versions later [CS14].

Besides software development, Git's influence extends into various fields requiring version control. In academia, for instance, Git is used for managing research projects and data and even writing academic papers, offering a solution to maintaining a consistent and traceable history of changes. The integration of Git with online platforms like GitHub and GitLab has further expanded its utility, making it an essential tool for collaborative projects, open-source software development, and educational purposes. These platforms provide a user-friendly interface and additional features like issue tracking, feature requests, and code reviews.

The main features that make Git state-of-the-art in VCSs are essential for implementing many software development practices and which will be many times referred to throughout the thesis, are as follows:

- Distributed nature. Unlike centralized version control systems, Git is distributed, meaning every developer's working copy of the code is also a repository that can contain the entire history of all changes. This distributed nature allows for greater workflow flexibility and redundancy, as there is no single point of failure.

- Branching and Merging: Git's lightweight branching allows developers to create isolated environments for new features or bug fixes, enabling parallel development without interference. Merging integrates these branches back into the main line of development, facilitating collaboration and innovation.

- Speed and Performance: Git is designed to be fast and efficient, even for large projects. Operations like branching, merging, and commit history retrieval are optimized for performance, making Git suitable for projects of any size.

- Data Integrity: Every file and commit in Git is checksummed, ensuring the integrity of the codebase and protecting against corruption. The immutable nature of Git commits also provides a reliable audit trail for changes. This property of commits will be thoroughly utilized in the practical part of the study, as an essential part of any git-based system is distinguishing changes between each other and understanding when and whether the change took place.

Remote code storage platforms such as GitHub, GitLab, and Bitbucket leverage Git for version control, providing a collaborative environment for developers to host, share, and manage their codebases online. These platforms offer several advantages, such as centralized hosting of projects, enhanced collaboration features like pull requests and code reviews, integrated issue tracking, and continuous integration/continuous deployment (CI/CD) pipelines [SABZ17]. Moreover, these platforms often have robust security features, including branch protection, role-based access control, and secure secret management, ensuring code integrity and safety in collaborative projects. By combining the power of Git with additional collaboration and management tools, these platforms significantly streamline the software development process, making them indispensable in modern development workflows.

2.2 DevOps

Though it is hard to track when the term "DevOps" itself came out for the first time, a need for a reliable and seamless software development delivery process might be as old as software development. However, the first documented appearance of the term came out between 2007 and 2008 when IT communities worldwide started expressing concerns about the traditional software development model where the one who writes code (Developers) was separated from one who supports and deploys/delivers it (Operations). In 2009, the first conference named after the concept, DevOpsDays, was held in Belgium, which made the term known to the general public [Mez18].

Therefore, it can be concluded that the DevOps approach is utilized as a methodology for software development, characterized by integrating traditionally separate teams within the IT framework, specifically the development (Dev) and operations (Ops) teams. This integration fosters coordination and collaboration between these teams, replacing manual workflows with automated processes through DevOps tools. The primary aim of DevOps is to enhance the speed, quality, and reliability of product delivery.

2.2.1 CI/CD

DevOps incorporates several key practices, notably Continuous Integration (CI), Continuous Delivery (CD), and Continuous Deployment (CD), which are foundational to its methodologies. "Continuous" refers to the regular and systematic repetition of processes,

from development to deployment, operation, and quality control, facilitating quicker production cycles and feature releases without sacrificing product quality. These core practices are depicted in Figure 1, illustrating the fundamental principles of DevOps methodologies [RUF21].

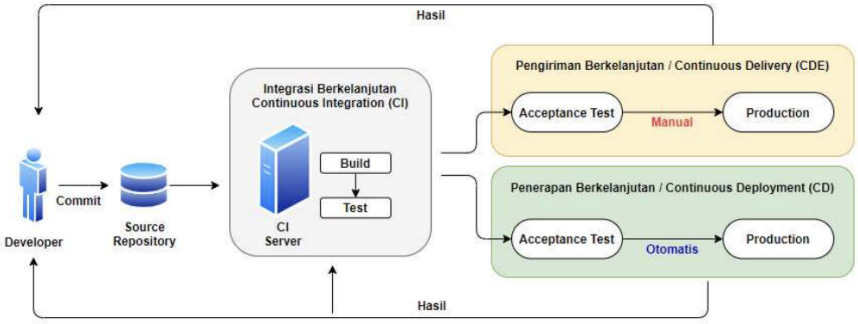

Figure 1. The Basic Principles of DevOps Practice. [LM12]

Continuous Integration (CI) and Continuous Deployment (CD) are practices that streamline the software development process, ensuring a seamless flow from development to production. CI involves frequently integrating code changes into a shared repository, where automated builds and tests are run to detect and fix integration errors quickly, resulting in enhanced software quality and reduced time to release. CD extends this process by automatically deploying all code changes to a testing or production environment after the build stage, enabling rapid and reliable delivery of functionalities to users. These practices make the development process more efficient and error-free. They also represent the collaborative approach of DevOps, breaking down barriers between development and operations teams and promoting a culture of continuous improvement and innovation. Integrating CI/CD into DevOps practices represents a significant evolution in software development, driving the delivery of high-quality software at a faster pace [BKA+23], emphasizing optimization and controlled deployment in cloud-based applications.

3 Semantics of GitOps

3.1 Kubernetes

Kubernetes is an open-source container orchestration platform that automates containerized applications' deployment, scaling, and management. Google initially started developing it and later donated it to the Cloud Native Computing Foundation (CNCF).

Kubernetes is the de facto standard for container orchestration, supported by a strong community and a wide range of infrastructure providers [BGO+16].

The architecture of Kubernetes is built around the principle of managing groups of containers, which constitute an application in a logical unit called a "pod." Pods are the smallest deployable units in the Kubernetes ecosystem and can contain one or more containers that share storage, network, and specifications on how to run the containers. Operations on pods are managed by higher-level abstractions like "Deployments" or "StatefulSets" which provide declarative updates to applications and maintain the desired state specified by the user[VPK+15].

An important aspect of Kubernetes is its cluster architecture. A Kubernetes cluster has at least one master node and multiple worker nodes. The master node hosts the cluster's control plane components, including the API server, scheduler, controller manager, etcd, a key-value store that stores the cluster's configuration and state. Worker nodes run the actual applications and workloads. Kubernetes automatically handles the placement of pods onto worker nodes and manages the accessibility of applications to users and other services.

When discussing deployment efficiency and infrastructure reliability, Kubernetes' self-healing mechanisms, such as auto-restarting, rescheduling, and replicating containers, should be mentioned. Its service discovery and load balancing features enable traffic distribution across multiple instances of an application, ensuring that new version deployment can happen with minimal or no downtime[Luk17].

In the scope of GitOps, Kubernetes offers wide opportunities to describe infrastructure and application state as a set of manifests that can be put under VCS for tracking. This approach is called Infrastructure as a Code and is covered separately and in relation to Kubernetes in the following subsection.

3.2 Instrastructure as a Code

Infrastructure as Code (IaC) is a practice that allows the management and automatic provision of infrastructure, such as load balancers, containers, orchestrators, etc., using a high-level programming language. This allows developers to eliminate manual methods, like executing command-line commands or using a graphic interface [PT21].

This is where both Git and DevOps come in place. First of all, VCS (which stands for Git in most cases and in this thesis) is an essential part of utilizing IaC. Since all infrastructure specifications are stored in files, they can be tracked with Git, which allows us to follow the history of changes, perform rollbacks, and understand how infrastructure evolved through time. In addition to that, it allows for easier collaborative development, every change to infrastructure might go through an established git flow, which usually includes creating a separate branch, pushing changes to repository, and getting them code reviewed. It adds more visibility and makes configuring infrastructure safer and more reliable.

IaC is also an essential part of DevOps practices. It allows for a more efficient development life cycle and easier and more efficient deployments, improving delivery times. Furthermore, IaC utilizes CI/CD in a way that configurations can be seamlessly integrated (e.g., invalid configurations will not make their way through the pipeline, therefore ensuring infrastructure's reliability) and delivered, as is demonstrated in Figure 1

One of the most common use cases of IaC, which will be extensively referred to throughout the thesis, is configuring and describing Kubernetes resources. Kubernetes API allows CLI to create resources, like deployments or services, but the most common way of describing them is declaratively by writing configuration files. As we already figured, these files can be put under version control, shared, and properly maintained, allowing for a smoothly managed Kubernetes cluster. Below is an example of how Kubernetes deployment is described in code:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: example-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: example
  template:
    metadata:
      labels:
        app: example
    spec:
      containers:
      - name: example-container
        image: example/image:latest
        ports:
        - containerPort: 80
```

However, even when following the IaC approach, direct changes to k8s cluster resources or any other type of infrastructure can still be made without modifying the code, creating inconsistencies. This is where our next main topic comes in.

3.3 GitOps

As previously described, GitOps is a relatively new concept introduced by Weaveworks in 2017. They define it as a set of practices for managing and automating IT infrastructure and application configuration with Git as a single source of truth. "Git as a single source of truth" is one of the core principles of GitOps described by its founders, with the others listed as follows:

1. Declarative. GitOps utilizes IaC, which enables Git to be the single source of truth. This approach simplifies deployments and rollbacks (which can be described as simple git actions, like *git push* or *git revert*). It also allows for rapid recovery of infrastructure if needed.

2. Versioned and immutable. Storing system's configuration in a version-controlled repository as the source of truth simplifies rollbacks to previous states using 'Git revert'. Git's security features allow commits to be signed with an SSH key, ensuring code authenticity and origin. The immutability of this version history is crucial for maintaining a reliable audit trail.

3. Pulled automatically. By storing the desired state in Git, GitOps automates system updates without requiring cluster credentials, offering a clear division between action and implementation. Although manual reviews are possible for specific deployments, the objective is direct, automated deployment from Git to Kubernetes following successful tests and checks, all governed by specific policies.

4. Continuously reconciled. With system's state defined and tracked, software agents can detect and correct deviations, including human errors, beyond Kubernetes' automatic handling of node or pod failures. These agents act as essential feedback and control mechanisms for maintaining system integrity. [Wea]

Approaches to implementing and adopting GitOps (as we will see later on) might differ depending on the tools used in infrastructure configuration, the code-storing platform (as long as it uses Git), and the implementation tools themselves. However, sticking to these principles is essential to keep the approach effective and performing as it is intended to, and evaluating their implementation is important for ensuring that the GitOps system stays reliable and compliant with all the needs of such a system. In this subsection, we will primarily focus on general guidelines on how GitOps is done, which is followed by a section dedicated to particular tools and approaches whose tools are utilized.

An entry point and main building block of GitOps is the Git repository. At least two of them should exist: application one and environment one. The application repository houses the application's source code and may include a Dockerfile for container creation.

The environment repository, on the other hand, holds deployment manifests detailing the desired infrastructure setup for the intended environment. This includes specifications on which applications and infrastructure services (like message brokers, service meshes, and monitoring tools) should be deployed, along with their configurations and versions. There is also an option to store application deployment manifests within the application repository.

Let us expand definitions of two main GitOps paradigms:

Push-based. Popular CI/CD platforms like Jenkins, CircleCI, and Travis CI employ a push-based deployment methodology. In this setup, the application's source code is stored in the same repository as the Kubernetes YAML files required for deployment. Updates to the application code initiate the build pipeline, creating container images and subsequent updates to the environment configuration repository with the latest deployment descriptors as shown in (see Figure2).

Any modifications to the environment configuration repository activate the deployment pipeline, which then applies all the manifests from the repository to the infrastructure. This method necessitates providing access credentials to the deployment environment, essentially granting the pipeline full access. In scenarios where automated cloud infrastructure provisioning is necessary, a push-based deployment becomes essential. In such instances, employing the cloud provider's detailed and adjustable authorization system is highly advised to enforce more stringent deployment permissions.

It is important to remember that the deployment pipeline is only activated by changes in the environment repository and does not automatically detect discrepancies between the current environment and its intended state. Therefore, a monitoring mechanism must be in place to allow for intervention if the environment deviates from what is outlined in the environment repository [BKH21].

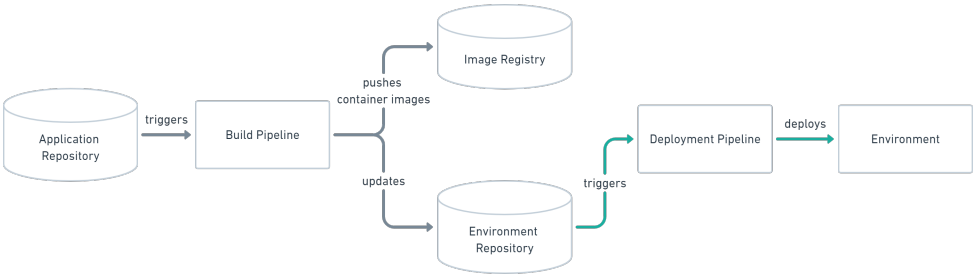

Figure 2. Push-based approach. [BKH21]

Pull-based. The pull-based deployment method incorporates the principles of its push-based counterpart but varies in the operation of the deployment process. Unlike traditional CI/CD pipelines activated by an external trigger, such as new code commits to a repository, the pull-based approach employs an operator. This operator assumes the

role traditionally held by the pipeline, consistently monitoring the desired state outlined in the environment repository against the actual state of the infrastructure. In case of discrepancies, the operator adjusts the infrastructure to align with the specifications in the environment repository. Additionally, it can monitor the image registry for new image versions to deploy(see Figure3).

Similar to the push-based method, this approach ensures the environment is updated in response to changes in the environment repository. The operator, however, also detects and corrects unilateral modifications to the infrastructure that deviate from the repository's specifications, thereby maintaining traceability of all changes within the Git log and preventing unauthorized direct modifications to the cluster.

This approach targets a fundamental limitation of push-based deployments, which rely solely on updates to the environment repository to initiate changes. However, this does not eliminate the need for monitoring. Operators typically offer notification capabilities, such as email or Slack alerts, for instances where achieving the desired state is hindered, such as issues in pulling a container image. Monitoring the operator is also advisable due to its critical role in automated deployment.

For optimal security and functionality, the operator should reside within the same environment or cluster as the application it deploys. This arrangement avoids the 'god-mode' associated with push-based methods, where the CI/CD pipeline has extensive deployment credentials. By situating the deployment mechanism within the target environment, external services are not privy to sensitive credentials. Deployment platform authorization features can be leveraged to control deployment permissions, enhancing security finely. This can be achieved in Kubernetes environments through RBAC configurations and service accounts [BKH21].

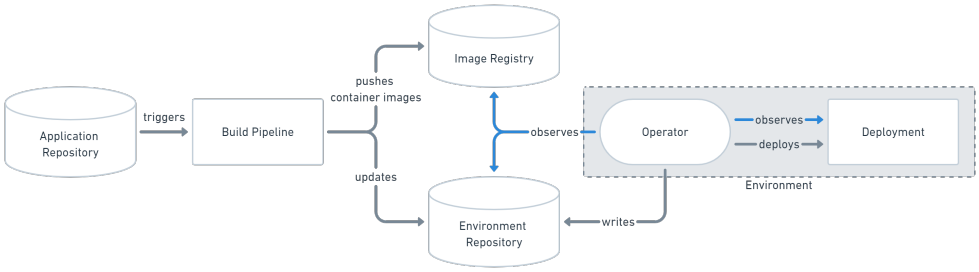

Figure 3. Pull-based approach. [BKH21]

3.3.1 ArgoCD

ArgoCD is a declarative continuous delivery open-source tool mostly targeting Kubernetes. It is currently of the most popular GitOps implementations officially adopted by

more than 300 companies [arg23]. It is also part of the Argo suite, a collection of tools for Kubernetes to facilitate job execution and application management. Argo CD, along with Argo Workflows, Rollouts, and Events, addresses application deployment needs, enabling seamless integration of services, workflows, and event-driven architectures. In 2020, the Cloud Native Computing Foundation (CNCF) recognized Argo CD's contributions to the cloud-native ecosystem by bringing it on board as an incubation-level project.

ArgoCD is also a good example of pull-based GitOps paradigm. It works by pulling updated code from Git repositories and deploying it directly to Kubernetes resources. This allows developers to manage both infrastructure configuration and application updates in one system.

The general ArgoCD flow looks like this: after the developer pushes the changes to the main branch of the repository (either directly or by merging a dedicated branch to a main one), a webhook is triggered, which notifies ArgoCD that the change was made. ArgoCD then clones the repo and compares application state with the state of kubernetes cluster, applying required changes to cluster configuration. By utilizing Kubernetes controllers, reconciliation happens, making the cluster achieve the desired configuration. After cluster readiness, ArgoCD reports that the application is in sync.

But since ArgoCD, as already mentioned, implements a pull-based model, it also works the other way around by monitoring the actual state of the Kubernetes cluster and making sure that it matches the state defined in Git. If not, it can discard them, preventing an out-of-sync state.

To get a better understanding of how the above general flow is implemented, we need to understand the two main concepts of ArgoCD:

- **Application** The Application serves as a structured way to organize Kubernetes resources; its main properties are source and destination. The Application's source specifies the repository's URL and the path to the relevant directory within that repository. It is a common practice to have separate directories inside repositories for different application environments, such as QA and Production. The destination for the Application specifies the deployment location for resources, which includes the URL for the API server of the intended Kubernetes cluster and the name of the cluster Namespace. By specifying these two properties, the ArgoCD application represents both environment states deployed in Kubernetes and links them to the desired state in the Git repository. The principle of ArgoCD application working is also depicted in Figure5

- **Project**. Projects are abstractions over ArgoCD applications that allow logical grouping. This enables engineers to set specific restrictions and settings for those groups.

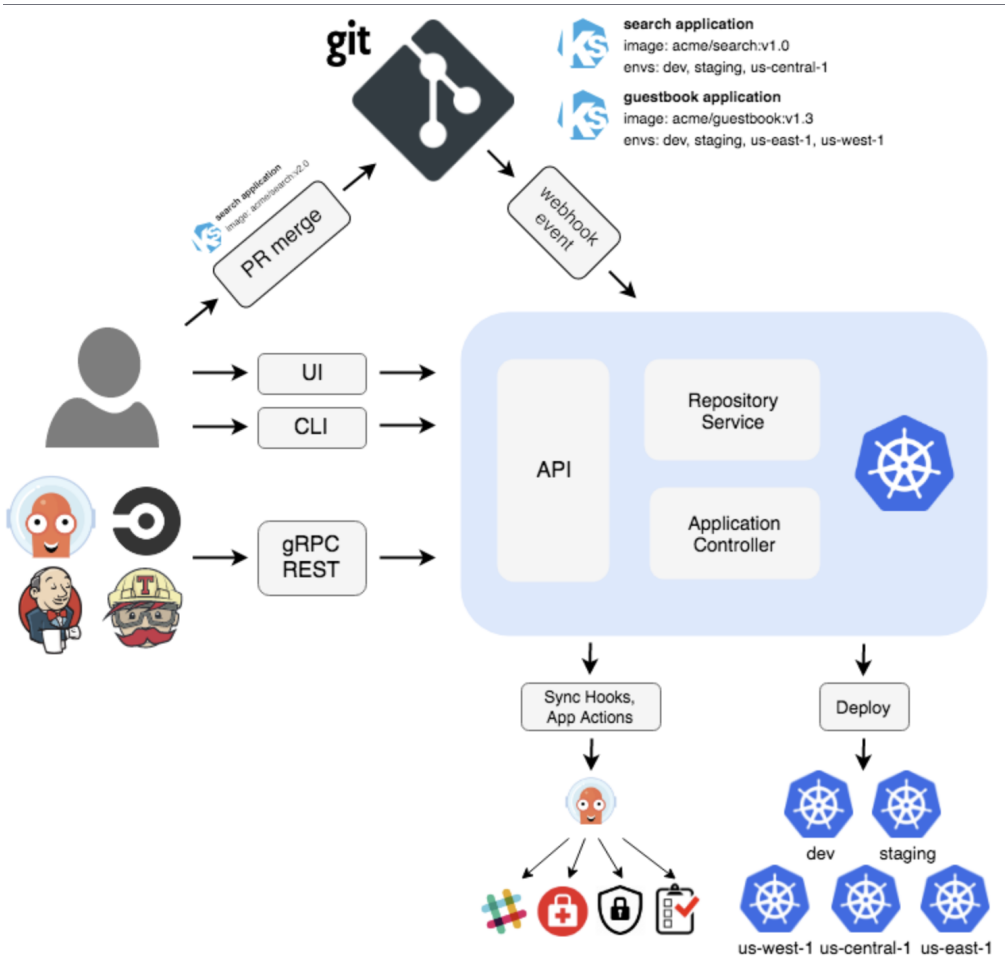

Figure 4. ArgoCD architecture. [arg24a]

3.3.2 Flux

Flux is an open-source tool for implementing GitOps which is already adopted by a variety of companies, which include companies like "Orange" and "Sap" [Flu24a]. It operates by aligning the state of manifests in a Git repository with the desired configuration of a cluster and offers integration with Prometheus and other key Kubernetes ecosystem components, providing multi-tenancy and the capability to synchronize multiple Git repositories.

Flux does this by providing multiple core concepts [Flua]:

- **Sources**. A Source represents the location of a repository that holds the system's intended state and the details needed to access it. Sources generate artifacts other Flux components utilize to execute tasks, such as deploying the artifact's contents

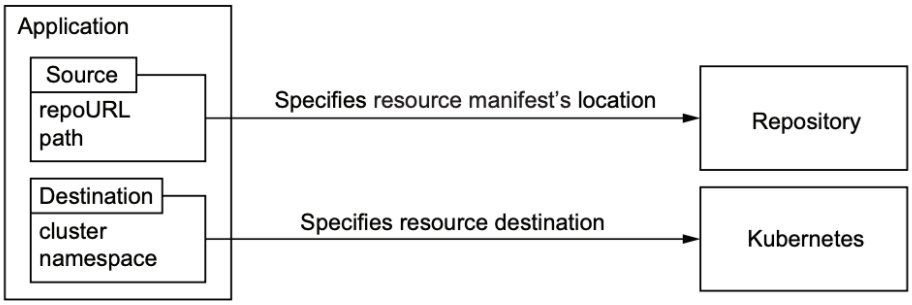

Figure 5. ArgoCD application. [BYS21]

to the cluster. Multiple entities can use these sources to minimize configuration and storage repetition. Sources are monitored at set intervals for updates, producing new artifacts when matching updates are found. In Kubernetes, sources are defined as Custom Resources, including GitRepository, OCIRepository, HelmRepository, and Bucket resources.

- **Reconciliation**. Reconciliation is the process of ensuring that an actual state (e.g., application running in the cluster, infrastructure) matches a desired state declaratively defined, for instance, in a git repository code. This refers to "Continuously reconciled" principle of GitOps and guarantees that it is fulfilled.

  There are various types of Reconciliation in Flux:

  - HelmRelease reconciliation: ensures the state of the Helm release matches what is defined in the resource and performs a release if this is not the case. Verifies and aligns the Helm release's actual state with its predefined resource configuration, initiating a release for any differences, including HelmChart resource revisions.
  - Bucket reconciliation: Periodically retrieves, archives, and stores bucket content as an artifact, logging the artifact's observed revision and details in the resource's status.
  - Kustomization reconciliation: Confirms that the deployment state of an application on a cluster corresponds to the definitions in Git, OCI repositories, or S3 buckets.

- **Kustomization**. The Kustomization custom resource in Flux designates a group of Kubernetes resources for Flux to manage within the cluster. By default, it checks for and enforces the desired state every five minutes, which can be adjusted with

.spec.interval. Any manual changes made directly in the cluster through commands like kubectl edit/patch/delete are automatically undone to maintain consistency with the defined state. This can be avoided by pausing the reconciliation process or applying changes through the designated Git repository.

- **Bootstrap** Bootstrapping in Flux stands for setting up its components within a cluster using GitOps principles. This process includes deploying manifests to the cluster, establishing a GitRepository and Kustomization for managing Flux's own configurations, and synchronizing these manifests with a specified Git repository, which can be either pre-existing or newly created. Flux is capable of self-management in the same way it manages other resources. This setup can be done via the flux CLI or by leveraging the Terraform Provider.

Under the hood, Flux uses GitOps Toolkit, a set of APIs and controllers that allow the creation of GitOps tooling for Kubernetes. It can be used either as an extension to Flux or on its own, allowing for customization and building own Kubernetes GitOps system (see Figure6).

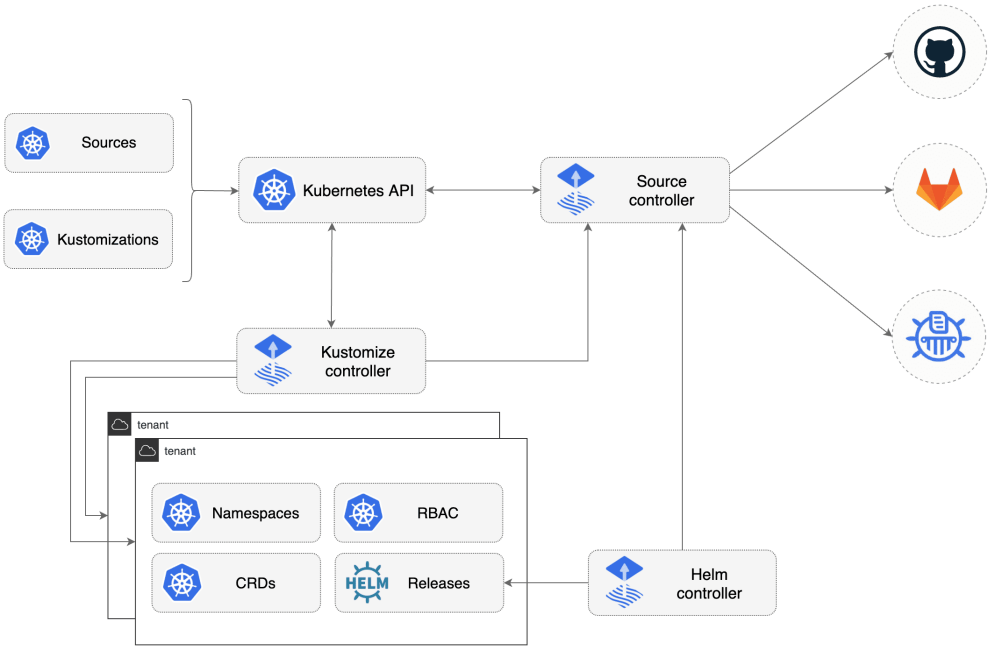

Figure 6. GitOps Toolkit usage in Flux [Flub].

As can be already understood, Flux uses the so-called Operator pattern, which is a popular implementation of the pull-based GitOps model. This model involves having

a dedicated entity that pulls updates from a source and applies them to an infrastructure while also detecting all the synchronization issues thanks to the Reconciliation mechanism.

3.4 GO

Go, also known as Golang, is a statically typed programming language developed at Google by Robert Griesemer, Rob Pike, and Ken Thompson and launched in 2009. The language prioritizes simplicity, efficiency, and readability, featuring a syntax close to C, contributing to its speed and performance. Concurrency in Go is supported by goroutines and channels, making it a solid choice for handling multiple processes simultaneously, particularly useful in server-side programming.

In the context of cloud-native technologies, Go has become increasingly essential and is the language behind several significant projects. Kubernetes, Docker, ArgoCD, and FluxCD, four technologies tightly related to GitOps and covered in some way throughout this research, are all written in Go[kub24][doc24][arg24b][flu24b]. They utilize its ability for efficient and flexible memory handling, concurrent capabilities, and strong typing. While having all these features, GO's syntax is very intuitive, making the development experience relatively simple. This is why GO is the language of choice for prototype GitOps implementation developed in the scope of this study.

3.4.1 Syntax

Variables. In GO, variables store data that can be used and manipulated. They are declared with the var keyword or by using the shorthand := which infers the type:

```
repoUrl := "https://github.com/example/repo.git"
repo, err := CloneRepo(repoUrl)
if err != nil {
    fmt.Printf("Failed to clone repository: %v\n", err)
}
```

Listing 1. Variable usage example

Conditions. Conditional statements in Go control the execution flow based on boolean conditions. The *if*, *else if*, and *else* are the primary conditional constructs:

```
if err != nil {
    fmt.Println("Error")
} else {
    fmt.Println("No error")
}
```

Listing 2. Condition example

Functions. In Go, functions are defined using the *func* keyword followed by the function's name, a list of parameters (enclosed in parentheses), and the return type. For example:

```
func CloneRepo(repoUrl string) (*git.Repository, error) {
    ...
}
```

Listing 3. Function usage exampple

Here, *CloneRepe* is a function that takes a string parameter *repoUrl* and returns a pointer to *git.Repository* and an error.

Error Handling. Go handles errors explicitly using the error type, which is a built-in interface. Functions that can encounter errors return an error object as part of their return values. This is checked in the calling code:

```
repo, err := git.Clone(...)
if err != nil {
    fmt.Println(err)
    return nil, fmt.Errorf("failed to clone repository: %v", err)
}
```

Listing 4. Error handling example

If *err* is not *nil*, an error has occurred.

Printing Output. The *fmt* package is used for formatted I/O with functions like *Println* and *Printf*:

```
fmt.Println("Repository cloned into memory successfully.")
fmt.Printf("failed to apply manifest: %v, output: %s", err, output)
```

Listing 5. Printing example

Println outputs a line of text followed by a newline, while *Printf* allows formatted strings similar to C's printf.

Structs and JSON. Structs are custom data types that group together variables. In Go, structs can be annotated with tags such as *json:"fieldname"* to specify JSON encoding/decoding behavior:

```
type GiteaPushEvent struct {
    Commits []struct {
        Added    []string `json:"added"`
        Removed  []string `json:"removed"`
        Modified []string `json:"modified"`
    } `json:"commits"`
}
```

Listing 6. Struct and handling JSON example

This struct represents a JSON object received from a Git push event, with fields mapped to JSON property names.

Command Execution. Go can execute external commands using the *os/exec* package. This is useful for running system commands like *kubectl*:

```
cmd := exec.Command("kubectl", "apply", "-f", "-")
cmd.Stdin = bytes.NewBufferString(manifestContent)
output, err := cmd.CombinedOutput()
```

Listing 7. Command execution example

Here, *exec.Command* creates a new command, setting its standard input to the content of *manifestContent*. *CombinedOutput* runs the command and returns its combined standard output and standard error.

3.5 Bash

Bash (Bourne Again SHell) is a powerful scripting language widely used in Unix-like operating systems to automate tasks, manage files, and run applications. It provides an environment for executing commands through the command line and standalone scripts.

Bash scripting plays a role in GitOps and cloud-native environments due to its powerful capabilities in automating and orchestrating complex workflows. In GitOps, where the entire operational workflow is centered around Git repositories and automated processes, Bash scripts can automate the deployment, monitoring, and management of configurations across distributed systems. This is particularly useful in Kubernetes, where Bash scripts frequently interact with the kubectl command-line tool to apply configurations, roll out updates, or revert changes based on the state defined in a Git repository.

3.5.1 Syntax

Bash scripts are written as a series of commands to be executed by the Bash shell. A script usually starts with a shebang line, which tells the system which interpreter to use to execute the script:

```
#!/bin/bash
```

Listing 8. Shebang

Variables. Variables in Bash store data that can be used throughout the script. Variables are assigned without the $ symbol but are accessed with it:

```
# Assigning a variable
filename="example.txt"

# Accessing a variable
```

```
5 echo $filename
```

Listing 9. Shebang

Loops. Basic syntax of loop in Bash:

```
1 for var in list_of_elements
2 do
3     commands
4 done
```

Listing 10. Loop example

sed. The sed (Stream Editor) tool parses and transforms text in data streams and files. It is powerful for performing text substitutions, more complex edits, or data extraction. Basic syntax:

```
1 sed 's/search_pattern/replacement_string/' filename
```

Listing 11. sed usage example

4 Implementation

4.1 Git server setup

The efficacy and performance of GitOps paradigms, whether pull-based or push-based, are influenced significantly by the characteristics and behavior of the underlying Git server. Benchmarking different GitOps paradigms requires a controlled environment where variables can be systematically manipulated and observed. Self-hosting a Git server provides this controlled environment, enabling precise measurement of the impact of these variables on the GitOps workflow performance.

It is also important to keep in mind considerations like the ability to control network latency and throughput, flexibility in adjusting load conditions, and custom configuration capabilities. For instance, Github, which is already mentioned in the introduction as one of the default options for external Git repositories management, offers limited customization of the server-side environment. Users cannot modify the server's behavior, install custom hooks directly on the server, or alter the software to better fit specific needs, which might be necessary for specialized GitOps workflows or for integrating unique features. In addition, it does not provide access to the underlying Git repository data at a granular level. Developers are limited to the interfaces and APIs provided by GitHub, which might not expose all data required for thorough GitOps tool testing or detailed performance analysis.

Among the multiple options present on the market, Gitea was chosen as the self-hosted solution due to the below-listed considerations:

1. Lightweight and Easy to Deploy: Gitea is written in Go, making it a lightweight and efficient solution easily deployed on minimal hardware specifications. This efficiency is essential for ensuring that the Git server itself does not become a bottleneck in performance benchmarks. It differentiates Gitea from Gitlab, which also can be used as a self-hosted solution but was considered superficial for this study as its functionality covers a broader range of software development lifecycle, including complex CI/CD capabilities and project management tooling[PCDH23].

2. Customization: As an open-source platform, Gitea allows for extensive customization and modification. Later in this section, it will be shown how flexible the tool actually is and how beneficial this property of Gitea is to the present study.

3. Built-in Webhooks and API: Gitea supports webhooks and provides API, allowing easy integration with GitOps tools and CI/CD pipelines. These features are crucial for automating GitOps workflows and initiating push-based operations based on repository events.

4. Community and Documentation: Gitea has a strong community and well-maintained documentation, providing support and resources that are invaluable during setup, customization, and troubleshooting. This support can significantly reduce the complexity and time of configuring the Git server for any needs.

5. Scalability: despite its lightweight nature, Gitea is scalable and can handle significant loads, making it suitable for high-load testing scenarios often required in performance benchmarking GitOps paradigms.

For hosting the Gitea server, an Azure (cloud provider by Microsoft) VM instance of Standard_B2s size was created. This instance offers vCPU of 2 and Memory of 4GiB, which is above Gitea's system requirements[Git23a] and should allow the server to run under heavy load and without downtime, which are intended during the experiments (see Figure7 for more information).

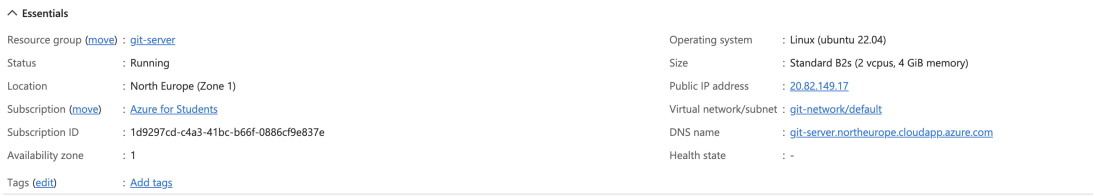

Figure 7. Azure VM configuration.

After connecting to the server, Gitea was installed and initially configured by following the install from a binary guide in the official documentation[Git23b].

4.2 Cluster setup

The next part of the environment setup is creating and configuring a Kubernetes cluster. As discussed earlier, modern GitOps use cases mostly involve using Git repositories as the source of truth for defining the desired state of a system, specifically in a Kubernetes environment. Thus, a Kubernetes cluster becomes an essential part of the testing framework for the GitOps tool.

As with git server, there are two possible ways of hosting cluster[MAB19]:

1. Hosted. Hosted Kubernetes means using a solution provided by one of the cloud providers to work with the cluster. It requires a subscription and is mostly used by enterprises or DevOps companies in production environments due to its reliability and ease of scale. Cloud providers also deliver support capabilities and extensive documentation, which is an important point for big projects and organizations. Some popular hosted Kubernetes services are: Google's cloud Kubernetes Engine, Azure Kubernetes Service, Amazon's Elastic Container Service for Kubernetes, IBM's Cloud Container Service, etc.

2. Self-hosted. Self-hosted Kubernetes engine allows a user to run Kubernetes on a single server usually for testing or educational purposes. It does this by automating steps used to deploy Kubernetes clusters. An example of such a tool is Kubeadm, which is used in various Kubernetes distributions to run locally on a single machine. Another popular solution is MiniKube. MiniKube is essentially a lightweight Kubernetes implementation running minimally by creating a VM on one's local machine and deploying a simple cluster containing only one node.

It can be seen that the self-hosted solution satisfies our needs and should be used in the scope of this study. Minikube was chosen because it is one of the most popular self-hosted implementations; therefore, it has a big community and comprehensive documentation, which is important for initial setup and in case of need in troubleshooting. It also supports different drivers, including Virtual-Box and Hyper-V, making it cross-platform. That is why the described environment can be easily reproduced regardless of the platform or operating system.

In given setup, minikube uses docker as driver, so docker-daemon needs to be started to run minikube. After docker is up and running, minikube can be started with the following command:

```
$ minikube start --profile thesis
```

This command will start the local Kubernetes cluster, which uses "thesis" as the name of the Minikube profile. To check that the cluster is running properly, the following command should be used:

```
$ minikube profile list
```

It lists all present minikube profiles and their properties, like profile name, driver used, kubernetes version used, etc. This is how output of the command looks like for running local cluster:

```
|---------|-----------|---------|--------------|------|---------|---------|-------|
| Profile | VM Driver | Runtime |      IP      | Port | Version | Status  | Nodes |
|---------|-----------|---------|--------------|------|---------|---------|-------|
| thesis  | docker    | docker  | 192.168.49.2 | 8443 | v1.28.3 | Running |     1 |
|---------|-----------|---------|--------------|------|---------|---------|-------|
```

As expected, the running cluster has docker as its driver and contains one node (which serves as the control plane at the same time).

4.3 Repository setup

The GitOps tool should track changes in the Git repository and ensure that it is in sync with the actual infrastructure (a local Kubernetes cluster, in our case). Therefore, it is essential to create an actual repository that will contain all the Kubernetes resources needed to be deployed to the cluster and updated in case of changes. A repository named "thesisCD-infra" was created in Gitea.

As an application to be deployed to the cluster and tracked by GitOps, I chose podinfo. Podinfo, as described by the authors, is a small web application developed in Go that was created specifically to showcase how to deploy and run microservices in kubernetes. Its repository also contains all the manifests needed to deploy this application to the cluster[Pro24]. So the initial content of the thesisCD-infra is:

- *deployment.yaml* Kubernetes manifest describing deployment resource of podinfo. Contains container settings (like image reference and env variables).
- *service.yaml* Manifest describing service resource of podinfo. Specifies which ports are to be used to access the application
- *hpa.yaml* Manifest describing HorizontalPodAutoscaler resource of podinfo used for defining autoscaling policies.

As mentioned in Section 3.1, one of the important features of Gitea is that it allows the use of Webhooks to notify about events in the repository. Push-based GitOps workflow will rely on these events and information from webhooks to know whether the repository was updated and what is the nature of these updates (e.g., were the target files updated that are related to repo configuration). Thus, the webhook was configured on the repository

level, specifying the target URL it is sent to, the push event trigger of it, and the Git branch that it sends the event about (main in our case).

Regarding the target URL, it is important to keep in mind that if the consumer of the webhook is running on the local machine, additional configuration is needed to allow the webhook sent by the Git server to reach the application running on the local server. For this purpose, ngrok application was used, which can create a tunnel from a publicly available domain with an https certificate to a local server. For the application that will expect webhooks on *http://localhost:8080/webhook* it is needed to run:

```
$ ngrok http 8080
```

And use the URL provided by ngrok in Gitea's webhook configuration as shown in Figure 8.

4.4 ArgoCD setup

As one cluster will be controlled and updated by a custom GitOps tool, one more cluster should be created using minikube to do the benchmarking with ArgoCD. After the creation of the cluster, ArgoCD can be installed with the following command:

```
$ kubectl apply -n argocd -f \
https://raw.githubusercontent.com/argoproj/argo-cd/v2.5.5/manifests/install.yaml
```

As described in Section 3.3.1 Application resource is the key part of ArgoCD's declarative setup. Manifest for the deployed application is listed below:

```yaml
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: argocd-app
  namespace: argocd
spec:
  destination:
    namespace: podinfo
    server: https://kubernetes.default.svc
  project: default
  source:
    repoURL:
    http://git-server.northeurope.cloudapp.azure.com/ \
        administrator/thesisCD-infra.git
    path: test
    targetRevision: main
```

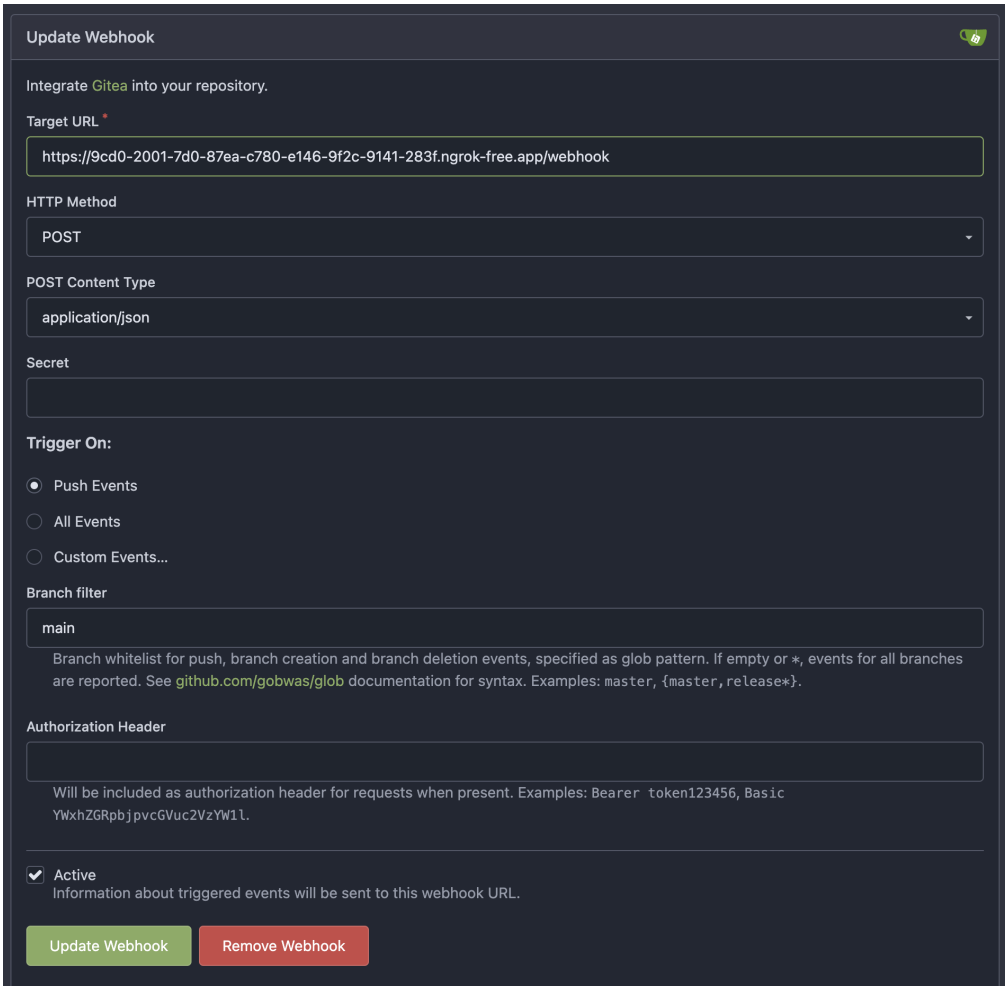

Figure 8. Gitea webhook configuration.

Here, the local kubernetes cluster is specified as the source, and the remote git repository, which ArgoCD will keep track of, is specified as the source. After creating the application by applying the below manifest, ArgoCD will sync the state of the cluster with the repository, and respective podinfo resources will be created in the cluster. It is also reflected in ArgoCD UI, which shows all the resources that are synced within the applications and the application's status (See Figure 9).

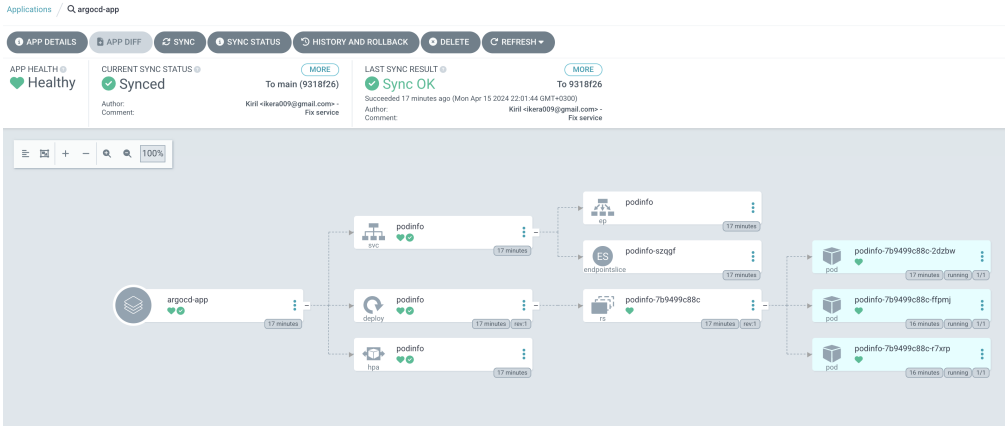

Figure 9. ArgoCD UI.

4.5 GitOps tool

Implemented software is a CLI application intended to run in two modes representing pull and push-based GitOps paradigms. It is written in Go language and extensively uses the go-git library for Git operations. go-git does not have any native dependencies, which makes it less error-prone. It also has some important features, such as a pluggable storage system, which allows it to work with the repository's in-memory copy without a need to clone it on a disk.

4.5.1 Modules

The software consists of two modules:

1. cmd. Contains entrypoints for the CLI which handles parameters defined by the users and bootstraps the application.

2. pkg. Contains functions used in both modes for performing git operations and working with the cluster.

git package from *pkg* module contains logic for performing git operations necessary for the GitOps workflow:

- **CloneRepo**. Function accepts repoUrl as a parameter and performs in-memory clone of the specified repository and returns go-git's reference of the repository, which we can later work with:

```
1 func CloneRepo(repoUrl string) (*git.Repository, error) {
2   repo, err := git.Clone(memory.NewStorage(), memfs.New(), &git.
        CloneOptions{
```

```
3       URL:              repoUrl,
4       Progress:         os.Stdout,
5       ReferenceName:    plumbing.ReferenceName("refs/heads/main"),
6       SingleBranch:     true,
7     })
8
9     if err != nil {
10      fmt.Println(err)
11      return nil, fmt.Errorf("failed to clone repository: %v", err
            )
12    }
13
14    fmt.Println("Repository cloned into smemory successfully.")
15    return repo, nil
16  }
```

Listing 12. CloneRepo implementation

- **PullAndApplyChanges**. This function accepts *repo* instance obtained previously as a parameter along with *path* in which infrastructure configurations are to be located.
  Then, on line 2, the Worktree instance of repository is get, and its pull method is executed on line 12.
  If no new commits were detected during the pull, the function exits, and the system keeps waiting for the next poll, as seen on line 16. If changes in the remote repository were detected, the previously obtained head's hash is compared with the new one (line 25), which was pulled to ensure that the change was done.
  If hashes are different, the system proceeds with checking if changes were done in a specified path and handle the deployment (line 26).
  An important consideration here is that we are deploying only the latest state of the repository by obtaining changes from the Head. So, no matter how many commits were done in between pollings, only the latest state will be processed in contrast with the push-based approach, where every commit needs to be handled:

```
1  func PullAndApplyChanges(repo *git.Repository, path string)
       error {
2    w, err := repo.Worktree()
3    if err != nil {
4      return fmt.Errorf("failed to get worktree: %v", err)
5    }
6
7    oldHead, err := repo.Head()
8    if err != nil {
9      return fmt.Errorf("failed to get HEAD before pull: %v", err)
10   }
11
```

```
12    err = w.Pull(&git.PullOptions{RemoteName: "origin"})
13    if err != nil && err != git.NoErrAlreadyUpToDate {
14      return fmt.Errorf("failed to pull: %v", err)
15    } else if err == git.NoErrAlreadyUpToDate {
16      fmt.Println("No new changes to pull.")
17      return nil
18    }
19
20    newHead, err := repo.Head()
21    if err != nil {
22      return fmt.Errorf("failed to get HEAD after pull: %v", err)
23    }
24
25    if newHead.Hash() != oldHead.Hash() {
26      err := checkPathChanges(repo, oldHead, newHead, path)
27      if err != nil {
28        return err
29      }
30    } else {
31      fmt.Println("No new commits were merged during the pull.")
32    }
33
34    return nil
35  }
```

Listing 13. CloneRepo implementation

webhook package from *pkg* module contains logic for handling incoming webhook. The structure of the webhook is described in Gitea documentation[Git23c], but in the code only the fields that are needed for commit handling are defined:

```
1  type GiteaPushEvent struct {
2    Commits []struct {
3      Added    []string `json:"added"`
4      Removed  []string `json:"removed"`
5      Modified []string `json:"modified"`
6    } `json:"commits"`
7  }
```

Listing 14. Gitea webhook type

kubernetes package from *pkg* module contains logic for applying manifests to local kubernetes cluster. It uses *exec* module from Go's *os* module to execute kubectl commands as passing manifests's content as BufferString:

```
1  func ApplyManifest(manifestContent string) error {
2    cmd := exec.Command("kubectl", "apply", "-f", "-")
3    cmd.Stdin = bytes.NewBufferString(manifestContent)
4
```

```
5     output, err := cmd.CombinedOutput()
6     if err != nil {
7       fmt.Printf("failed to apply manifest: %v, output: %s", err,
            output)
8     }
9
10    fmt.Printf("Manifest applied successfully: %s\n", output)
11    return nil
12  }
13
14  func DeleteResource(manifestContent string) error {
15    cmd := exec.Command("kubectl", "delete", "-f", "-")
16    cmd.Stdin = bytes.NewBufferString(manifestContent)
17
18    output, err := cmd.CombinedOutput()
19    if err != nil {
20      fmt.Printf("failed to delete resource: %v, output: %s", err,
            output)
21    }
22
23    fmt.Printf("Resource deleted successfully: %s\n", output)
24    return nil
25  }
```

Listing 15. Kubernetes operations implementation

4.6 Queue theory application

To evaluate the performance and throughput of push-based and pull-based implementations and compare their performance a reliable framework is needed to model the behavior of each paradigm, helping to understand and predict the dynamics of deployment tasks under varying conditions. A structured approach is also needed to quantify and compare the efficiency and effectiveness of each paradigm.

Queueing theory provides such a framework since it is widely used for systems where line or queue form due to a demand for resources exceeding available supply.

A standard used within queueing theory to classify queueing nodes is a Kendall notation, which proposes describing queueing models using three factors written as A/S/c where A represents the time between arrivals to the queue, S is the service time distribution, and c is the number of service channels open at the node[Ken53].

Arrival process A can be represented as follows:

Service time distribution S:

Symbol	Description
M	Markovian (Poisson or random arrival process).
E	Erlang distribution.
D	Degenerate distribution (Deterministic/fixed-time arrival process).
G	General distribution.

Table 1. Arrival process notations

Symbol	Description
M	Markovian (Exponential service time).
E	Erlang distribution.
D	Degenerate distribution (A deterministic or fixed service time).
G	General distribution.

Table 2. Service time distribution notations

c is represented as a whole positive number.

Pull-based paradigm can be modeled as a D/G/1 system (deterministic arrivals, general service times, one server). Deterministic arrival can be applied to this paradigm because the pull-based system operator polls the repository for changes at regular, pre-configured intervals. Thus, the model helps us understand how the frequency of timed checks affects the deployment process's stability and performance. Service times might vary based on the complexity of the changes needed to match the desired state, including the time taken to detect differences and apply updates. The service process might also include the time to check for and deploy new image versions from the image registry.

Push-based paradigm can be modeled as an M/G/1 system (Poisson arrivals, general service times, one server), reflecting the stochastic nature of commits to the repository. Each commit triggers the deployment pipeline, making the arrival process random. Similar to the pull-based model, service times depend on the complexity of the deployment manifests to be applied. However, push-based systems may also face bursts of arrivals when multiple commits are pushed quickly, potentially leading to queue buildup and increased waiting times.

After modeling, performance metrics should be established which will be used for future evaluation:

- **Utilisation** (ρ): This metric will help assess how effectively the computational resources (servers) are used. In a pull-based system, utilization tends to be smoother and more predictable. Conversely, a push-based system might see highly variable utilization, with potential spikes in resource demand following multiple simultaneous commits.

Utilization is calculated as the ratio of the arrival rate (λ) to the service rate (μ) of the system:

$$\rho = \frac{\lambda}{\mu} \tag{1}$$

If $\rho > 1$ then updates are arriving faster than the system can handle them. This results in queue buildup. $\rho = 1$ means that the service rate matches the arrival rate, e.g., queue is not growing, and the system is fully utilized. $\rho < 0$ means that service is higher than the arrival rate so the queue is stable.

- **Average Queue Length (L)**: In pull-based systems, the queue length is likely to be low due to regular and predictable polling. For push-based systems, the average queue length can vary significantly depending on the frequency and timing of commits.
- **Waiting Time (W)**: This metric is critical in operations where deployment speed is competitive. Pull-based systems might show relatively constant waiting times, while push-based systems could see highly variable waiting times, especially during periods of high commit activity.

By benchmarking using the above metrics, it is possible to make a justified decision on which of the paradigms performs better, what are the bottlenecks of the two GitOps approaches and how they can be improved depending on the use case.

5 Experiments and benchmarks

To perform experiments, it is necessary to simulate a variable load on a GitOps server in order to understand how it performs in different scenarios. To accomplish this, a simple bash script was created, which allows to do small commits to kubernetes manifest *deployment.yaml* and pushes them to Git server. It also gives an ability to configure the number of commits to perform.

```
generate_random_hex_color() {
    printf '#%06X\n' $(( RANDOM % 0xFFFFFF ))
}

YAML_FILE="test/deployment.yaml"

NUM_COMMITS=$NUM_COMMITS

for (( i=1; i<=$NUM_COMMITS; i++ ))
do
    NEW_COLOR=$(generate_random_hex_color)

```

```
13      sed -i '' "49s/.*/                value: $NEW_COLOR/" "$YAML_FILE"
14
15      git add "$YAML_FILE"
16
17      git commit -m "Update color to $NEW_COLOR on iteration $i,
            experiment $EXPERIMENT_NAME"
18
19      git push origin main
20  done
```

Listing 16. commit and push script

Another important aspect of tracking metrics and analyzing them is introducing logging, which allows one to follow which process is executing and when a certain event (e.g., pulling, incoming webhook, deployment) was triggered, which is important for performing benchmarks and visualizing results for experiments.

For this *logrus* Golang library is used to dump logs in JSON format in a dedicated log file, and its API is completely compatible with the standard Golang logger[Esk24]. Each log contains a description of the operation (e.g., "Deployment triggered", "Deployment started"), timestamp, and unique identifier of the operation (in our case, commit hash can be used) as shown in Figure 10. These values are sufficient for future calculations.


```
{"commitID":"be5d2d9e2532867b6e810a8c0f2d4e68ab72b97d","level":"info","message":"Deployment triggered","msg":"Action performed","time":"2024-05-08T16:22:28+03:00","timestamp":1715174544}
{"commitID":"be5d2d9e2532867b6e810a8c0f2d4e68ab72b97d","level":"info","message":"Deployment completed","msg":"Action performed","time":"2024-05-08T16:22:29+03:00","timestamp":1715174549}
{"commitID":"7bcbf31ca1c8389a52fb3966ef13516020217aa9","level":"info","message":"Deployment triggered","msg":"Action performed","time":"2024-05-08T16:22:30+03:00","timestamp":1715174545}
{"commitID":"7bcbf31ca1c8389a52fb3966ef13516020217aa9","level":"info","message":"Deployment completed","msg":"Action performed","time":"2024-05-08T16:22:31+03:00","timestamp":1715174551}
```


Figure 10. Logs format

All the measurements and results presented are relative to the implemented prototypical GitOps system and can differ when the same experiments are run with another tooling. For example, Waiting Time W can vary depending on the efficiency of deployment handling, network latency, and underlying optimizations. The goal is not to provide results comparable to other GitOps implementations but to compare two paradigms in the scope of one system.

5.1 Waiting time

The practical aspect of measuring Waiting Time W, in scope of GitOps system, involves defining a timestamp T_1 when task enters the queue (Enqueued) and when the second timestamp T_2 which is recorded when task starts being processed (which in case of GitOps means right before deployment to kubernetes is triggered). Considering the above, waiting time is calculated as:

$$W = T_2 - T_1 \tag{2}$$

For both push-based and pull-based approaches T_2 will be measured at the time of the same event. Meanwhile, T_1 will be recorded differently. In a pull-based system, a task is enqueued when the polling event takes place, as without this, there is no way the GitOps system might know about updates in the repository. While in a push-based system, enqueued T_2 is the moment when change is pushed to the repository, as this is when the task enters the queue to be processed.

Four commit and push scenarios are simulated for benchmark purposes:

- High frequency, high quantity: 100 commits and pushes N_c are done sequentially representing scenario when there is a high load (in real-world scenarios, this might be a situation when multiple developers are actively working on the project).
- Low frequency, high quantity: 100 commits and pushes N_c is done sequentially with 10 seconds interval I representing scenario when there is high commit activity but commits are not done frequently.
- High frequency, low quantity: 20 commits and pushes N_c are done sequentially without an interval.
- Low frequency, low quantity: 20 commits and pushes N_c are done sequentially with 10 seconds intervals I.

Going over each scenario presented on Figure 11:

1. The waiting time remains relatively stable, around 6-7 seconds, with an average W of 6.85 seconds. This scenario represents a system where the capacity (service rate) closely matches the demand, allowing it to handle bursts of 20 commits without significant increases in waiting time. The relatively stable and low waiting time suggests that the system is efficiently processing deployments without significant backlog or resource constraints.
2. The waiting time shows stable behavior, at around 5 seconds, with an average waiting time of 5.40 seconds. Introducing intervals between commits helps manage the queue more effectively, reducing the average waiting time compared to the no-interval scenario. This suggests that spreading out the deployment requests allows the system to handle each request more swiftly, likely due to better resource availability at each deployment initiation.
3. Waiting time shows a progressive increase, starting from 10 seconds and rising to an average of 11.73 seconds. The trend indicates a gradual buildup in the queue. This scenario highlights the system's limitations under high load conditions. The absence of intervals between many commits (100) leads to increased waiting times as the queue builds up. This suggests that the service process is slower than the

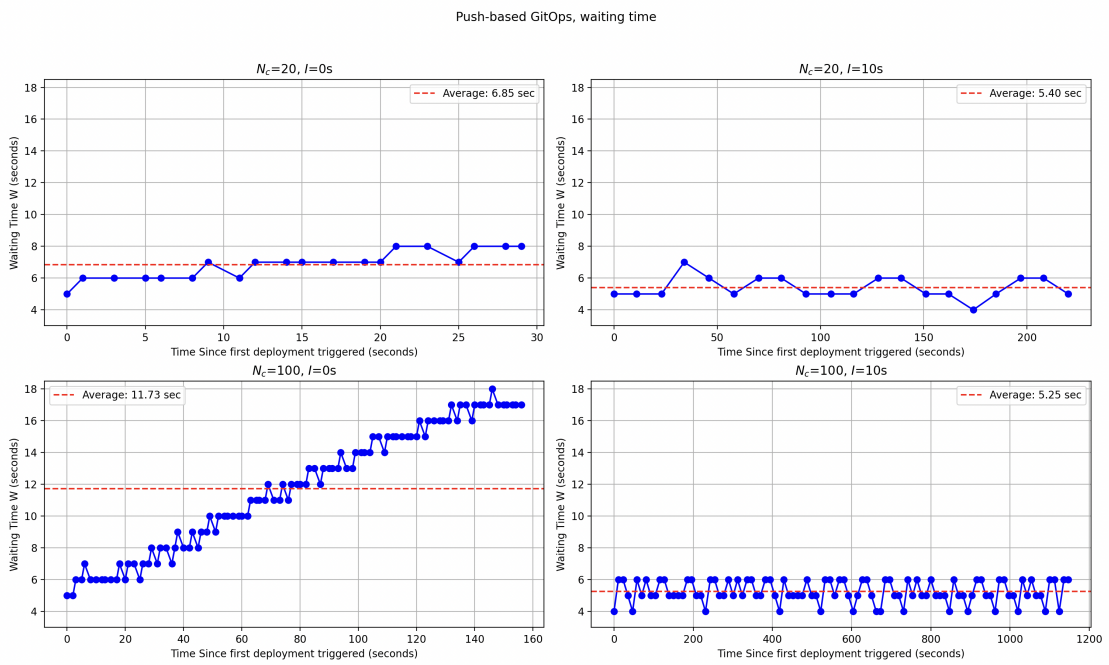

Figure 11. Push-based GitOps, Waiting time measurements

arrival rate of new deployments, leading to queue saturation and increased waiting times.

4. The waiting times are significantly lower and more stable, maintaining close to an average of 5.25 seconds, even with many commits. Spacing out the commits at 10-second intervals significantly mitigates the impact seen in the no-interval scenario with the same number of commits. The system can effectively manage the queue without significant buildup, indicating that the intervals allow the service process enough time to handle each commit before the next one arrives.

For a pull-based system, one more parameter is introduced, which is a polling interval I_p. As shown in Figure 11, for the push-based system, it takes less than 30 seconds to process 20 commits without an interval. So a value of I_p will be established as three and 20s to emulate real-world shorter and longer polling intervals.

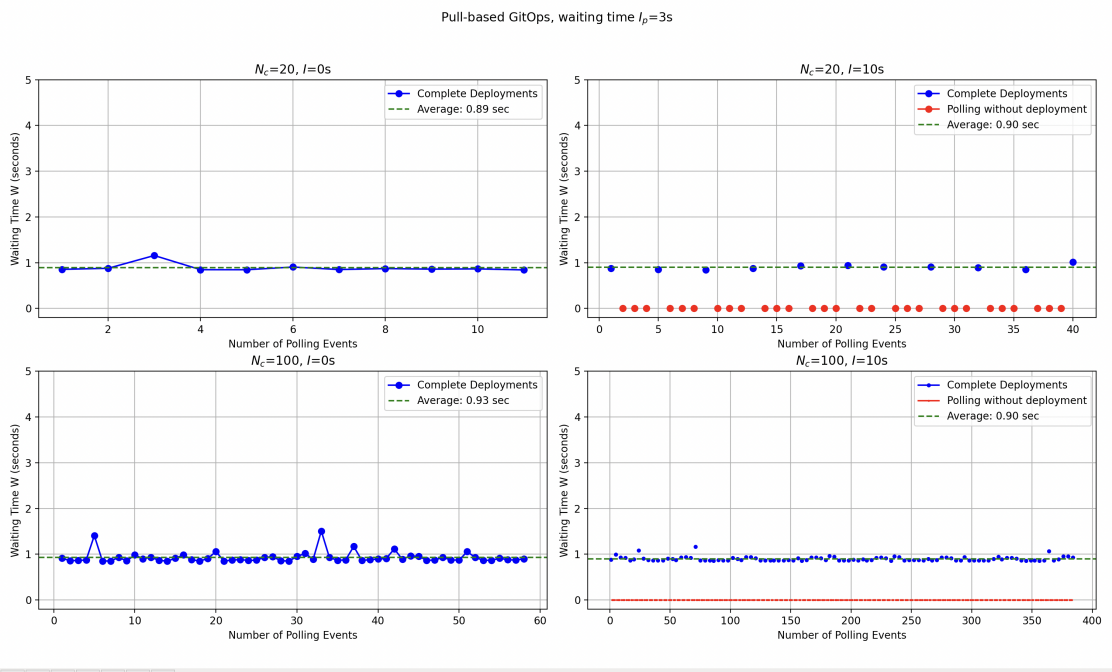

Figure 12. Pull-based GitOps, Waiting time measurements, $I_p = 3s$

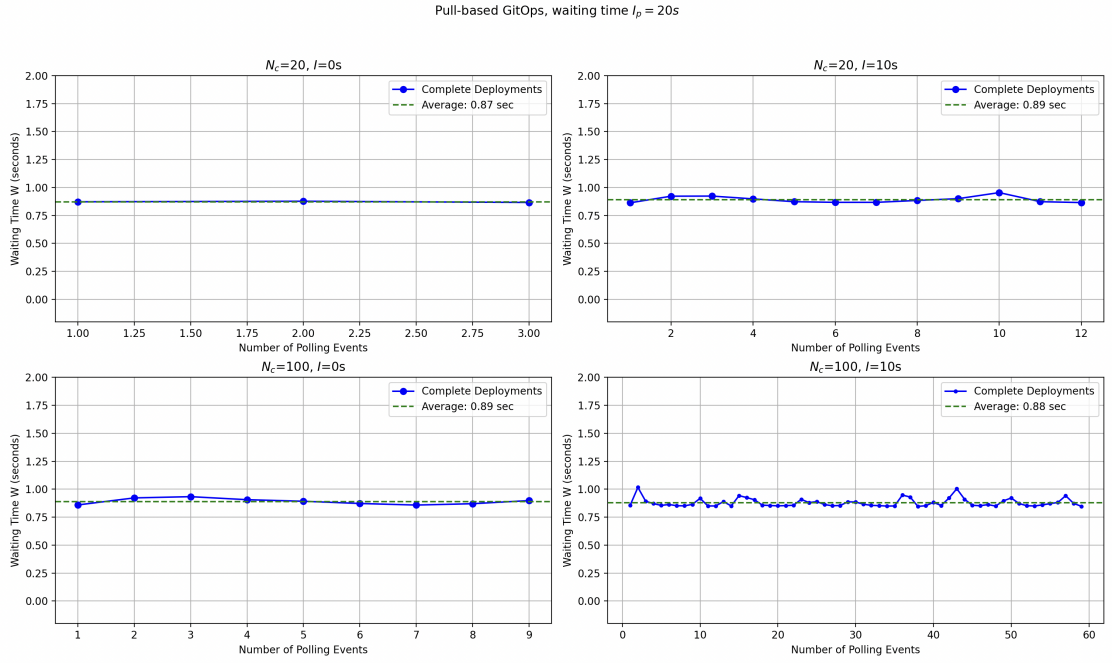

Figure 13. Pull-based GitOps, Waiting time measurements, $I_p = 20s$

As seen in Figure 12 and Figure 13, the waiting time remains consistent across all experiments with overall high stability. However, slight spikes can be observed in scenarios with many commits. Though the polling interval does not affect waiting time, it has an effect on the number of "Polling without deployments" events when polling took place. However, it did not result in actual deployment because no changes occurred in the repository. This is where the utilization ratio comes in place.

5.2 Utilization

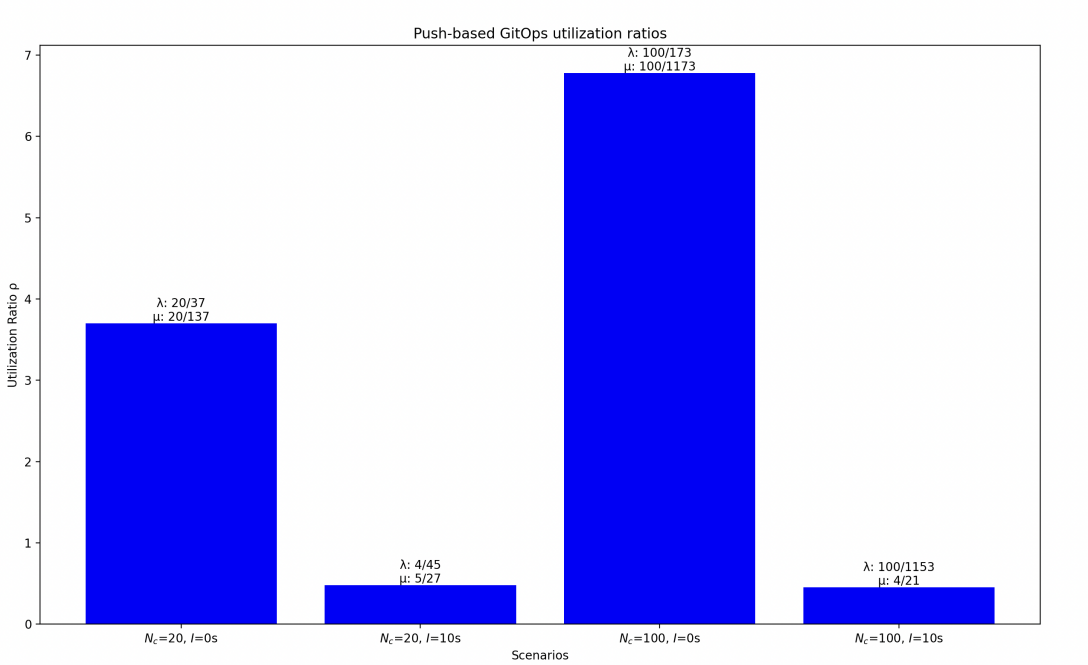

Figure 14. Push-based utilization ratios

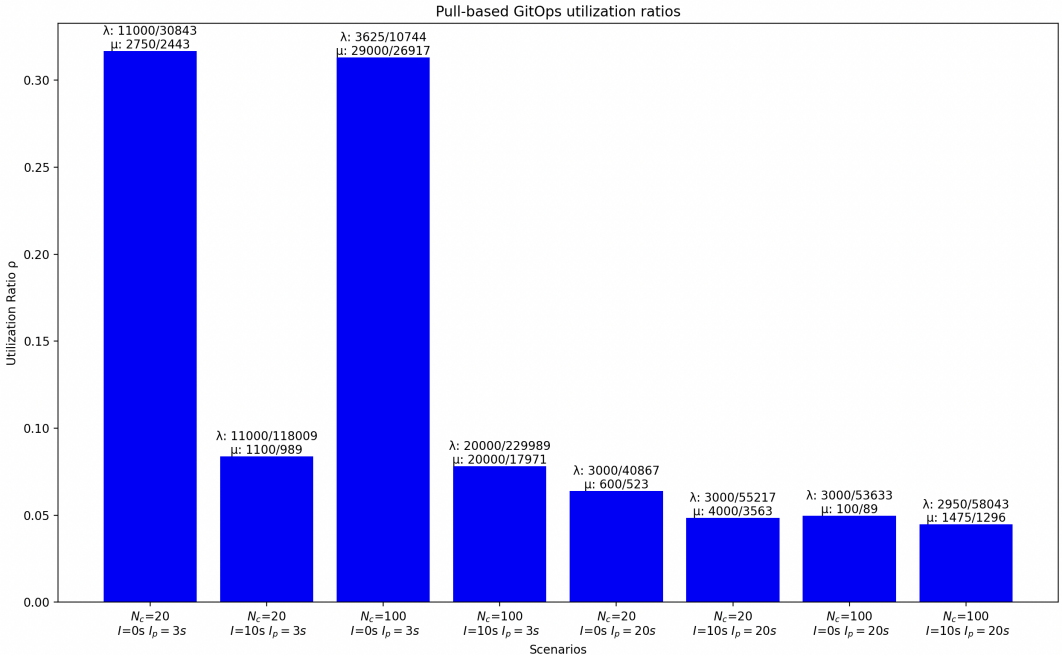

Figure 15. Pull-based utilization ratios

As shown in Figure 14 and 15, utilization ratios are significantly higher than one can be observed, as expected, in a push-based system with low commits interval. This is because, in pull-based systems, arrival rates (that can also be observed in the Figures) are consistently lower than the service rates due to the nature of the pull-based systems. In contrast, for push-based systems handling frequent updates, arrival rates tend to surpass service rates. Therefore, the queue is growing, resulting in increased waiting time, as shown in Figure 11.

While pulling-based system utilization is always lower than 1, the queue is not growing, and the system is not reaching its full capacity. It also tends to be lower for systems where the polling interval increases (because of a decrease in arrival rate).

5.3 Average queue length

For average queue length measurements, Little's Law rule can utilized, which is a theorem determining average queue length based on the waiting time and arrival rates:

$$L = \lambda W \tag{3}$$

Based on the previous measurements, it can be expected that L will be higher for push-based systems in scenarios with low commit intervals.

As shown in Figure 16 assumptions were correct, the more the utilization ratio for the scenario is, the more deployments stay in the queue to be processed.

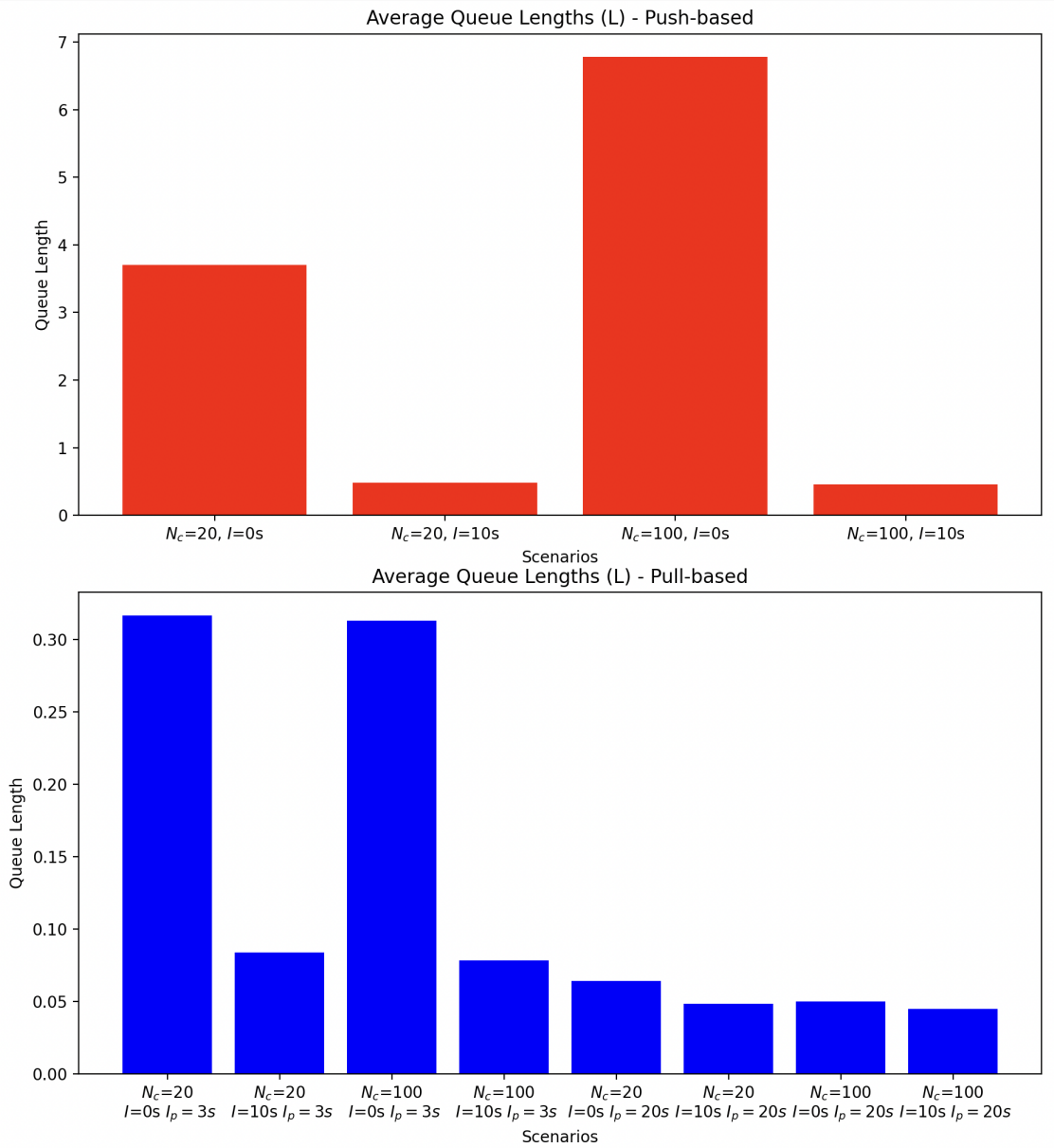

Figure 16. Queue length

5.4 ArgoCD benchmarks

To adjust ArgoCD to act as pull-based GitOps with automatic synchronization within a specified interval, it is necessary to update the app's configuration by running the following:

```
argocd app set <APPNAME> --sync-policy automated
```

And configure ArgoCD ConfigMap by specifying parameter *timeout.reconciliation* which sets the interval for synchronization (value is "3s" in our case).

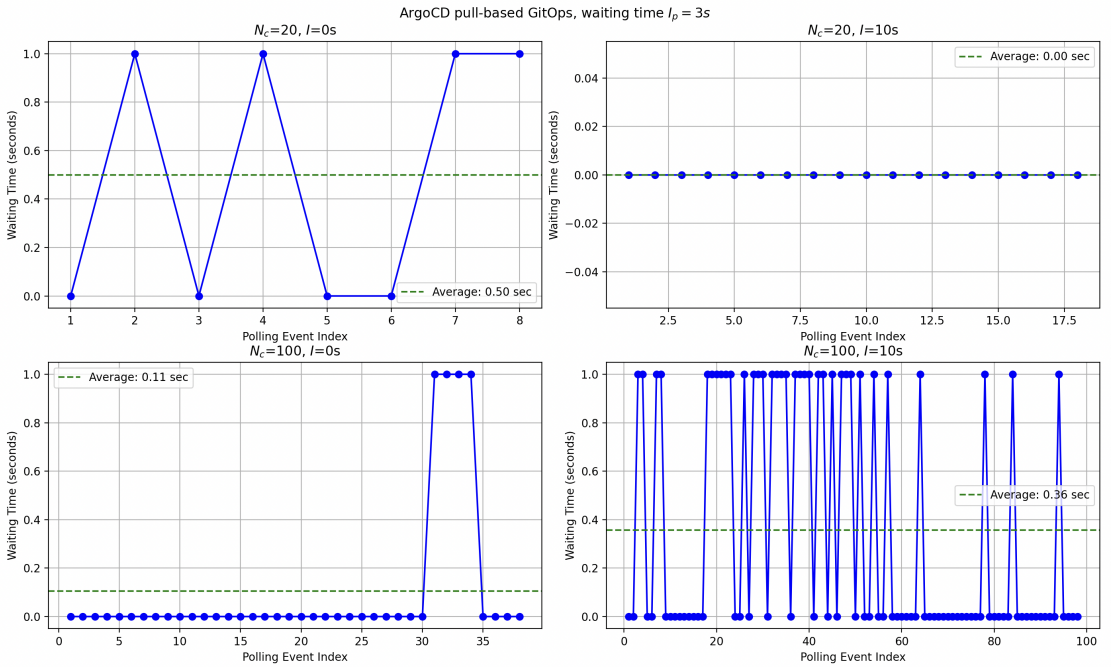

Figure 17. ArgoCD pull-based waiting time Ip = 3s

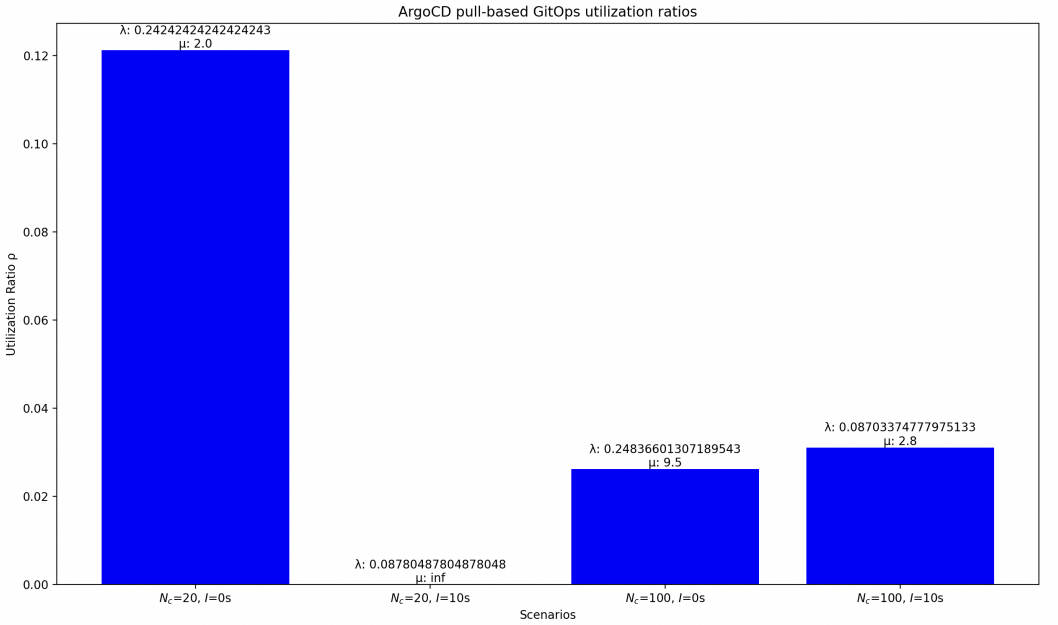

Figure 18. ArgoCD pull-based utilization Ip = 3s

As we can see in Figure 17 and 18 waiting times are significantly smaller for ArgoCD than for prototypical implementation, which might happen due to more efficient deployment handling by ArgoCD. Though spikes in waiting time happen, in utilization measurements, we can see that utilization ratios stay relatively low, with one scenario showing 0 utilization rate (this exact scenario has 0 waiting time, so it is expected).

6 Conclusion

The GitOps concept was introduced, starting with a thorough overview of its foundational technologies like Git and methodologies like DevOps, proceeding with more specific tools like Kubernetes and GO. GitOps itself was covered in detail, focusing on how the concept emerged, which problems it solves, and what its basic building blocks are. Then, based on previously presented information, two main approaches to GitOps were extracted, with in-detail descriptions of how they are implemented and first assumptions on how they might perform in comparison. Two already existing and widely used implementations for GitOps, Flux, and ArgoCD, were presented and described to give the reader an understanding of the state-of-the-art field and how production-ready GitOps is done.

To evaluate the performance of extracted GitOps paradigms, multiple steps were

performed: setting up a self-hosted Git server to have a controlled and configurable environment that can be adjusted to specific benchmarking and testing needs, Kubernetes cluster was set with deployment described in manifests files and put under Git. As the last step, a prototypical implementation of the GitOps operator was developed, which implemented both pull-based (utilizing goroutines) and push-based (utilizing Gitea-provided webhooks) paradigms.

With the help of this setup, thorough testing was performed for both paradigms, with different scenarios and under varying loads. Results were interpreted with a queueing theory, a decent theoretical framework for such systems.

Results have shown that the pull-based GitOps paradigm demonstrates higher stability and is less prone to queue buildup (considering that the polling interval is adjusted accordingly). At the same time, the arrival rate and frequency of deployments are advantages of the push-based paradigm, which makes it a choice for organizations that value these parameters but, at the same time, can ensure stability and quick targeting of issues related to a higher utilization ratio.

Evaluation of GitOps paradigms or different tools utilizing the same paradigm through mathematical models should be considered in the future. As DevOps methodology is becoming mainstream and the time-to-market of software features decreases, any reduction in deployments' latency and reliability improvement will become more valuable. There are several GitOps solutions available on the market, as discussed in this article. These solutions offer a wide range of configurable setups and utilization scenarios. Therefore, it is important for future research in this field to evaluate these solutions and provide developers with insights into which paradigm performs better under specific circumstances.

References

[arg23] argoproj. Who uses argo cd? https://github.com/argoproj/argo-cd/blob/master/USERS.md, 2023. Accessed: 2023-12-30.

[arg24a] argoproj. Architectural overview. https://argo-cd.readthedocs.io/en/stable/operator-manual/architecture/, 2024. Accessed: 2024-02-10.

[arg24b] argoproj. Argocd, 2024. Accessed: 2023-04-11.

[BGO+16] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. Borg, omega, and kubernetes. *ACM Queue*, 14(1):70–93, 2016.

[BKA+23] Ahmed Mateen Buttar, Adeel Khalid, Mamdouh Alenezi, Muhammad Azeem Akbar, Saima Rafi, Abdu H. Gumaei, and Muhammad Tanveer Riaz. Optimization of devops transformation for cloud-based applications. *Electronics (Switzerland)*, 12(2), 2023. Cited by: 3; All Open Access, Gold Open Access.

[BKH21] Florian Beetz, Anja Kammer, and Dr. Simon Harrer. *GitOps: Continuous Deployment for Cloud Native Applications*. 2021.

[BYS21] Todd Ekenstam Billy Yuen, Alexander Matyushentsev and Jesse Suen. *GitOps with Argo CD*. Manning publications, 2021.

[CS14] Scott Chacon and Ben Straub. *Pro Git*. Apress, Berkeley, CA, 2nd edition, 2014.

[doc24] docker. docker, 2024. Accessed: 2023-04-11.

[Esk24] Simon Eskildsen. logrus, 2024. Accessed: 2023-04-11.

[Flua] Core concepts. https://fluxcd.io/flux/concepts/. Accessed: 2024-02-10.

[Flub] Gitops toolkit components. https://fluxcd.io/flux/components/. Accessed: 2024-02-10.

[Flu24a] Flux adopters. https://fluxcd.io/adopters/, 2024. Accessed: 2024-02-26.

[flu24b] fluxcd. flux2, 2024. Accessed: 2023-04-11.

[Git23a] Gitea documentation. https://docs.gitea.com, 2023. Accessed: 2023-04-11.

[Git23b] Gitea documentation - installation. https://docs.gitea.com/installation/install-from-binary, 2023. Accessed: 2023-04-11.

[Git23c] Gitea documentation - webhooks. https://docs.gitea.com/usage/webhooks, 2023. Accessed: 2023-04-11.

[Ken53] David G. Kendall. Stochastic processes occurring in the theory of queues and their analysis by the method of the imbedded markov chain. *The Annals of Mathematical Statistics*, 24(3):338–354, 1953.

[KRM+23] István Koren, Felix Rinker, Kristof Meixner, Moritz Kröger, and Michael Zeng. Implementing devops practices in cpps using microservices and gitops. volume 2023-September, 2023. Cited by: 0.

[kub24] kubernetes. kubernetes, 2024. Accessed: 2023-04-11.

[LM12] Jon Loelinger and Matthew MacCullogh. *Version Control with Git - Powerful Tools and Techniques for Collaborative Software Development: Covers GitHub, Second Edition*. O'Reilly, 2012.

[Luk17] Marko Luksa. *Kubernetes in Action*. Manning Publications, 2017.

[LVDP22] Ramón López-Viana, Jessica Díaz, and Jorge E. Pérez. Continuous deployment in iot edge computing a gitops implementation. volume 2022-June, 2022. Cited by: 2.

[MAB19] Ruchika Muddinagiri, Shubham Ambavane, and Simran Bayas. Self-hosted kubernetes: Deploying docker containers locally with minikube. In *2019 International Conference on Innovative Trends and Advances in Engineering and Technology (ICITAET)*, pages 239–243, 2019.

[Mez18] Steve Mezak. The origins of devops: What's in a name? https://devops.com/the-origins-of-devops-whats-in-a-name/, 2018. Retrieved 6 May 2019.

[PCDH23] Marius Politze, Ulrich Christoph, Barbara Decker, and Petar Hristov. Supporting software development processes for academia with gitlab. In *Proceedings of European University*, 2023.

[Pro24] Stefan Prodan. podinfo, 2024. Accessed: 2023-04-11.

[PT21] Sneh Pandya and Riya Guha Thakurta. *Introduction to Infrastructure as Code: A Brief Guide to the Future of DevOps*. Apress, 2021.

[RUF21] Ramadoni, Ema Utami, and Hanif Al Fatta. Analysis on the use of declarative and pull-based deployment models on gitops using argo cd. page 186 – 191, 2021. Cited by: 3.

[SABZ17] Mojtaba Shahin, Muhammad Ali Babar, and Liming Zhu. Continuous integration, delivery and deployment: A systematic review on approaches, tools, challenges and practices. *IEEE Access*, 5:3909–3943, 2017.

[VPK+15] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at google with borg. In *European Conference on Computer Systems (EuroSys)*. ACM, 2015.

[Wea] Weaveworks. What is gitops? `https://www.weave.works/technologies/gitops/`. Accessed: 2023-12-30.

[Wea21] Weaveworks. The history of gitops. `https://www.weave.works/blog/the-history-of-gitops`, 2021. Accessed: 2023-12-30.

Appendix

I. Source Code

The source code for this thesis is accessible from a public GitHub repositories: *https://github.com/kirilxd/thesisCD* and *https://github.com/kirilxd/thesisCD-infra*.

II. Licence

Non-exclusive licence to reproduce thesis and make thesis public

I, **Kyrylo Riazantsev**,
(author's name)

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to

   reproduce, for the purpose of preservation, including for adding to the DSpace digital archives until the expiry of the term of copyright,

   Study on GitOps paradigms,
   (title of thesis)

   supervised by Bruno Rucy Carneiro Alves De Lima.
   (supervisor's name)

2. I grant the University of Tartu a permit to make the work specified in p. 1 available to the public via the web environment of the University of Tartu, including via the DSpace digital archives, under the Creative Commons licence CC BY NC ND 3.0, which allows, by giving appropriate credit to the author, to reproduce, distribute the work and communicate it to the public, and prohibits the creation of derivative works and any commercial use of the work until the expiry of the term of copyright.

3. I am aware of the fact that the author retains the rights specified in p. 1 and 2.

4. I certify that granting the non-exclusive licence does not infringe other persons' intellectual property rights or rights arising from the personal data protection legislation.

Kyrylo Riazantsev
15/05/2024