University of Tartu
Faculty of Science and Technology
Institute of Computer Science
Software Engineering Curriculum

Mohamed Moustafa Nagy Mohamed Salem

Context-Aware GitOps

Master’s Thesis (30 ECTS)

Supervisor:
Bruno Rucy Carneiro Alves de Lima, MSc

Tartu 2025

Context-Aware GitOps

Abstract:

GitOps is a process that allows for declarative management of infrastructure objects by using Git as the source of truth, GitOps is built on top of DevOps principles which emphasize automation, collaboration, and continuous delivery - Current GitOps tools lack contextual awareness: treating all Git changes as equally significant, triggering deployments for syntactic changes and not understanding dependencies between infrastructure objects, resulting in sequential rather than parallel execution of changes.
This thesis proposes a context-aware approach to GitOps that utilizes incremental Abstract Syntax Tree (AST) analysis using the Tree-sitter library. After analyzing the changes, it generates parallel execution plans. A prototype is implemented to demonstrate the feasibility of the approach.

Keywords: GitOps, ArgoCD, Tree-sitter, Incremental analysis

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

Kontekstiteadlik GitOps

Lühikokkuvõte:

GitOps on protsess, mis võimaldab infrastruktuuriobjektide deklaratiivset haldamist, kasutades Git-i kui ainsat tõeallikat. GitOps põhineb DevOpsi põhimõtetel, mis rõhutavad automatiseerimist, koostööd ja pidevat tarnet. Praegustel GitOpsi tööriistadel puudub aga kontekstitundlikkus – kõik Git-i muudatused käsitletakse võrdselt olulistena, mistõttu käivitatakse juurutused ka süntaktiliste muudatuste korral ega mõisteta infrastruktuuriobjektide vahelisi sõltuvusi. See viib muudatuste järjestikulise, mitte paralleelse täitmiseni. Käesolev lõputöö pakub välja kontekstitundliku lähenemisviisi GitOpsile, kasutades inkrementaalset abstraktse süntaksipuu (AST) analüüsi Tree-sitter teegi abil. Muudatusi analüüsides loob süsteem paralleelsed täitmisplaanid. Prototüüp on rakendatud, et demonstreerida selle lähenemise teostatavust.
Võtmesõnad: GitOps, ArgoCD, Tree-sitter, Incremental analysis
CERCS: P170 Arvutiteadus, arvutusmeetodid, süsteemid, juhtimine

Table of Contents

1 Introduction 4
2 Background 7
2.1 Git 7
2.2 DevOps 8
2.2.1 Software Environments 10
2.3 CI/CD 10
2.4 Containers 11
2.4.1 Container Orchestration 13
2.5 Kubernetes 14
2.5.2 Kubernetes Architecture 17
2.6.1 Tree-sitter 19
2.7 GitOps 20
2.7.1 ArgoCD 21
3 Applying incremental analysis to GitOps 23
4 Results 33
4.1 Hypothesis and Objectives 33
4.2 Observations 34
4.3 Test Dimensions: 36
4.4 Comparison with Traditional GitOps Systems 37
4.5 Possible Improvements 38
References 39
Appendix 42
I. Source Code 42
II. License 43

1 Introduction

Modern software and infrastructure systems are often managed through code. This approach of infrastructure management is known as Infrastructure as Code (IaC), the configuration files describe how the system should behave. Software teams rely on version control systems such as git to store their software and configuration files and monitor the changes.

GitOps is a practice where teams apply version control to infrastructure management, aiming to make sure that the desired state of their infrastructure objects such as servers, applications or security policies match the configuration in the version control system.

Infrastructure is updated by modifying files in the Git repository that then are picked up by tooling like ArgoCD or FluxCD automatically, which then apply the changes to the infrastructure.

By treating Git as the single source of truth, it provides highly valuable features like change traceability, reproducibility, and automation.

However, it has limitations that impact performance and scalability, especially in large systems or time-critical deployments.

A core limitation of GitOps tools like ArgoCD and FluxCD is the lack of contextual awareness. These tools apply changes sequentially, applying every modified file in the repository without understanding the relationships between the resources or the impact of those changes.

For example, a small change in a file like adding a whitespace is considered a change and would result in a deployment. This results in unnecessary deployments and resource usage.

These limitations could cause delays when deploying large amounts of changes and may lead to downtime while some resources fail because some changes conflict with them.

Since the GitOps tools are unable to identify the dependencies between Kubernetes resources or whether the changes are semantic or syntactic, they can't execute the resource changes in parallel groups, restricting deployment speed and creates overhead in environments that need high-velocity deployments.
The changes in files could either be semantic or syntactic changes:

- Semantic changes are changes that affect the actual configuration of a resource, such as changing a value from 0 to 1
- Syntactic changes are changes that do not affect the actual configuration of a resource, such as adding whitespaces at the end of the file

Abstract Syntax Tree (AST) could be used to address the limitations of GitOps tooling.
ASTs are structured representations of the source code that store the meaning of the files rather than the formatting of the files. Applying them to GitOps systems would enable the system to detect whether a change is a semantic change or is a syntactic change.

The main goals of the research are to develop and evaluate a system that:

- Parses infrastructure code and understands the structure and dependencies of the code.
- Differentiates between semantic and syntactic changes.
- Groups resources with related dependencies.
- Generates execution plans that allows for simultaneous deployments of unrelated changes.

The following questions are the main guiding questions:

- Can syntactic changes be filtered out to reduce redundant deployments?
- Can Kubernetes resources be modeled in a dependency graph to understand the relationship between different resources?
- What would be the improvements in speed or reliability gained by running parallel deployments?

By answering these questions, the thesis would improve the scalability and reliability of GitOps deployment workflows, which reduce deployment time and provide more predictable deployment results in environments where time and reliability are critical.

2 Background

The chapter aims to introduce the key concepts around this problem, and the tools involved in addressing the research problem.

This chapter will start by introducing Git, and DevOps then go on to introduce Kubernetes and GitOps tools like ArgoCD then finally discuss incremental analysis and the Tree-sitter tool.

2.1 Git

Git is an open-source tool used for source code management, it is a distributed version control system (VCS), It was developed by Linus Torvalds in 2005 to manage changes in the Linux kernel. It enables contributors to work on the same codebase at the same time by maintaining a complete history of changes to files [progit].

Git requires developers to commit their changes to their local repository and then later sync their local repository with the remote repository which contains other developers' changes as well. Each commit represents a snapshot of the changes committed in it, showing the additions, deletions, and merges in this commit.

Git is tightly integrated with websites like GitHub and GitLab which provide interfaces for managing Git repositories and collaborating on code changes. These platforms allow developers from different backgrounds whether it is a closed source program developer or an open source program to collaborate and host their code on the platforms.

These platforms offer features like branch management, commit history browsing, and more all through a web interface. In addition to hosting the code, they allow developers to review the proposed changes through features like pull requests, where developers can collaborate and review or reject or request additional changes for the reviewed changes.

In the context of infrastructure resources management, Git is utilized in managing the infrastructure resources configuration, as it is used to version control Kubernetes manifests and Infrastructure as Code (IaC) configurations. Using git in this way allows infrastructure teams to manage infrastructure resources the same way they manage application code.

The changes to the infrastructure resources are reviewed first by other teammates the same way the application code is reviewed then they get merged, they can also be easily rolled back in case of an incident or a fault occurring.

This method of organizing work with infrastructure objects allows for a bigger team to collaborate easily, track changes and who committed them, and allow for easy rollback which also means that the state of the desired infrastructure objects is stored in Git.

Storing infrastructure objects and versioning them in Git, is the foundation of GitOps workflows.

A term that would be used later on is a monorepo, which is a git repository that may contain code and configuration files for different services, environments or applications at once, this is sometimes used in large organizations to simplify dependency management and to enable teams to work more closely together. [atlasmono]

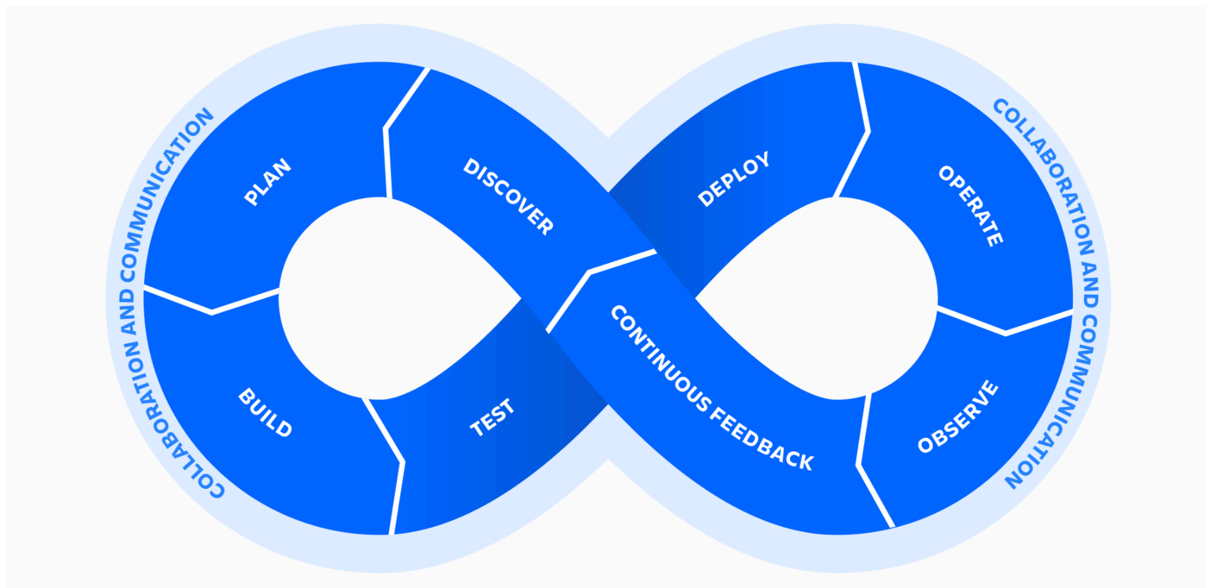
2.2 DevOps

The term DevOps consists of combining development (Dev) and Operations (Ops) to increase the speed of delivery compared to other traditional development methods [devgitlab]. It was first coined by the Belgian IT consultant Patrick Debois around 2009 [devrelic] when he organized the first DevOpsDays conference in Belgium. He wanted a memorable name for the event so he decided to combine the words “Dev” and “Ops” to highlight the collaboration between these 2 different teams which usually work separately from each other. He started online discussions with the hashtag #DevOps which quickly became the standard name for this movement. He later on said he picked this name as it was shorter as he initially thought that “Agile System administration” was too long.

DevOps is a set of practices that aim to increase delivery through automation, collaboration, continuous integration (CI), continuous development (CD), fast feedback, and iterative development.

DevOps started as a response to the demand for faster and more reliable software delivery, especially in large-scale systems that require frequent deployments with complex deployment

logic. DevOps encourages software teams to work together at all stages of development and deployment as it focuses on incremental development and rapid deployment of changes.



Listing 1. DevOps lifecycle [devatlas]

The DevOps culture which is often represented as an infinity loop as seen in Listing 1, is more than just pushing the developers to work closely with the operations team, it's a mindset where teams change their way of working to be more agile.

DevOps culture means developers work more closely with the users by understanding the user requirements and goals. Operations teams get closer to development teams and understand their requirements and needs. The DevOps mindset aims to deliver software at a faster pace and higher quality than the traditional software development model.

A key principle of the DevOps mindset is the automation of all the processes that could be automated around the software delivery lifecycle, focusing on Continuous Delivery (CD) and Continuous Integration (CI).

2.2.1 Software Environments

It is necessary to discuss first software environments to proceed with the other topics. A software development environment is a collection of all the resources needed to run a software system. [devenv]

It can include many parts ranging from specific hardware, a specific set of servers to applications running, and specific configurations for that environment. It can also include libraries, and system software required for the testing, deployment, and execution of the software application. It also typically includes databases and whatever is necessary to support the lifecycle of the application. To summarize, it is the complete set of tools, software, configuration, and hardware that work together to run a complete software system.

It’s possible to run different kinds of software environments consisting of mostly the same components but with different configurations, this approach is used in companies to test applications first before pushing them. Environment types can typically be broken down into 3 kinds:

1. Production environment: A software environment used to run the production version of the software which is used by the end users and the business depends on, this is the most critical software environment typically software teams try to keep bugs away from it and operations teams try to make sure it is stable and available.
2. Staging environment: The environment used usually for testing by the developers, sometimes a company may have only 1 or a handful or tens of them, and a company may need many of them in case many developers want to work on bigger features and test them together or if they have multiple production environments so they need to replicate the differences in multiple staging environments as well.
3. Demo environment: A Demo or beta environment is usually used to test the application fully before releasing it to the production environment.

   Teams usually replicate the production environment by copying data, such as replicating the production database, into the demo environment and running the new version of the application for testing.

2.3 CI/CD

Continuous Integration (CI) and Continuous Delivery or Continuous Deployments (CD) are the core practices of DevOps workflows. Continuous integration is a process where

developers continuously integrate new code in the same shared repository, with each new commit triggering a new build and test process in the pipeline. This method of work ensures that code changes are as validated as they come and makes it easy to pinpoint which new change is causing issues through automated tests. This reduces integration issues and improves code quality and collaboration between development teams.

Listing 2. CI/CD pipeline [ciredhat]

As seen in Listing 2. CI/CD pipeline starts with building the application then testing it and if the changes are reviewed and approved merging the changes after merging comes to the CD stage where a new release is made automatically then the application gets deployed to the production environment. This process provides a smooth experience from writing the code to building and testing it then getting it to production in a way that is much more efficient than the traditional method of providing releases to operations teams to deploy the new release manually.

2.4 Containers

Containers are software packages that include a given application with all its dependencies and necessary components, such as the application code or binary, runtime libraries, system tools, and configuration files. [contaka]

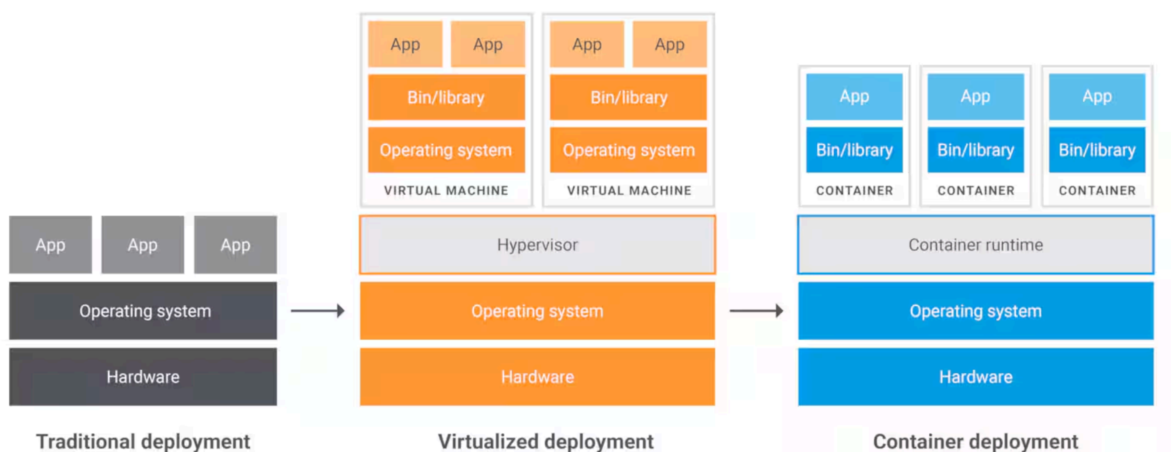
The container ensures that the application can run smoothly without issues in different kinds of computing environments. It can be a personal device, a public cloud, or a private data center. Containers virtualize the operating system, which allows the applications to be deployed and executed reliably and consistently across different platforms.

The concept of containerization tackles the challenges that were faced when deploying applications across different environments and different computing platforms. Deploying applications directly on the computing platform caused issues where one server may have

different configurations or different dependencies versions that may go unnoticed and may lead to inconsistent management and performance across different servers.

Containers solve the problem of inconsistency between different computing platforms by bundling the application with its dependencies and environment, which ensures that the application behaves the same way regardless of where it is deployed.

This approach of building the application in container once then running it anywhere makes the development and deployment process streamlined, which allows developers and operations teams to deploy the application without editing the code for different environments.



What is a container vs. a virtual machine?

Listing 3. Container vs a virtual machine [contaka]

As seen in Listing 3 in a traditional deployment the application runs directly on the operating system while in a container deployment, it runs on the container runtime which then runs on the operating system, it takes care of the operating system communication and virtualizes the operating system for the container, which enables the application to run on various platforms.

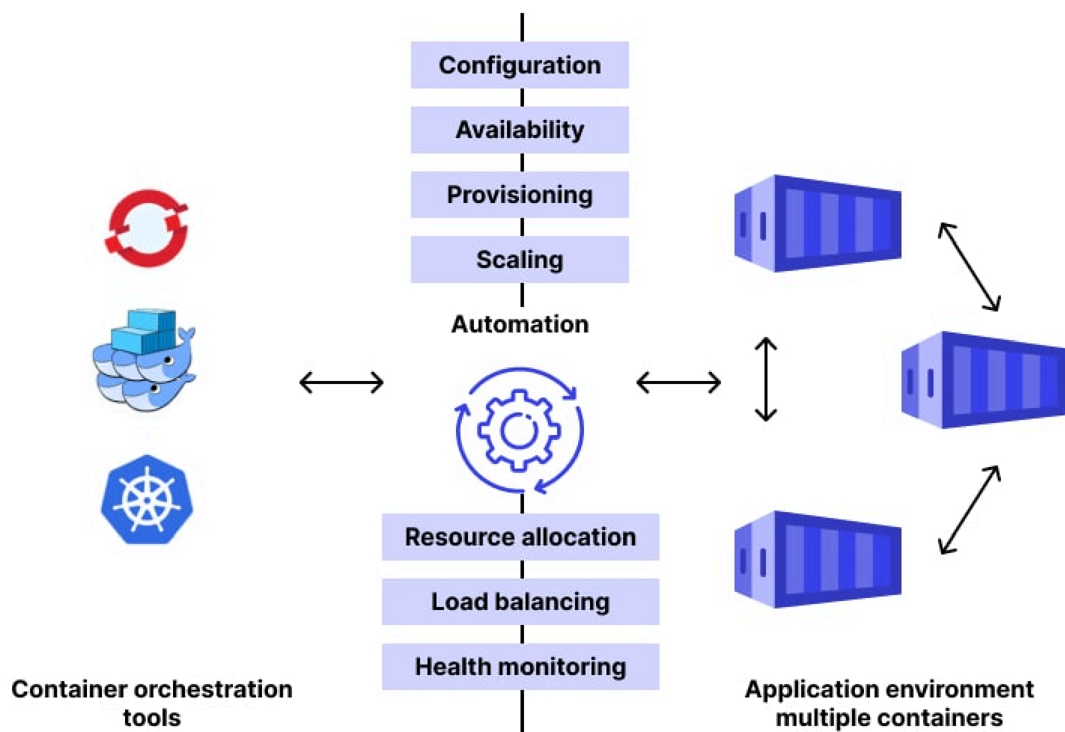
Container images have several possible formats but the most common one is the Open Container Initiative (OCI), Container runtime engines like Docker, CRI-O, and Containerd are OCI-compliant.

These engines are responsible for running the container images, they run on a wide variety of computing platforms, configure and control container networking, and resources, and monitor the containers.

The operating system enforces the configured resources per container, killing the container for example if it tries to use more memory than it is allowed, and enforces separation between containers which ensures secure and efficient operation.

The management and deployment of containerized applications requires an orchestration tool which is introduced in the following section.

2.4.1 Container Orchestration



Listing 4. Container orchestration tools explanation [wallarmorch]

Container orchestration tools play an important role, they manage container deployments across different computing platforms and automate the lifecycle of a container including but not limited to provisioning a container, monitoring its state, scaling it up and down, managing resources for it, connecting it with other applications via internal networks and connecting it to the public internet, service discovery, load balancing and security as seen in listing 4.

These tools

They manage the lifecycle of containers and provide visibility and control over how the containers are deployed, and which machines they are allocated to and based on what.

They automate a large portion of the work of the operations team by choosing the servers, monitoring the servers' health, automatically restarting the container if it fails, and ensuring minimum downtime.

The most well-known container orchestration tools are Docker Swarm, Kubernetes, and Open Shift (which is built on top of Kubernetes).

2.5 Kubernetes

Kubernetes is a widely adopted container orchestration platform that deploys, runs, and manages containerized applications across many computing platforms in a distributed manner and at a scale. It automates configuring the containerized applications and manages their resources. [k8sredhat]

Kubernetes is built with 3 core design principles in mind:

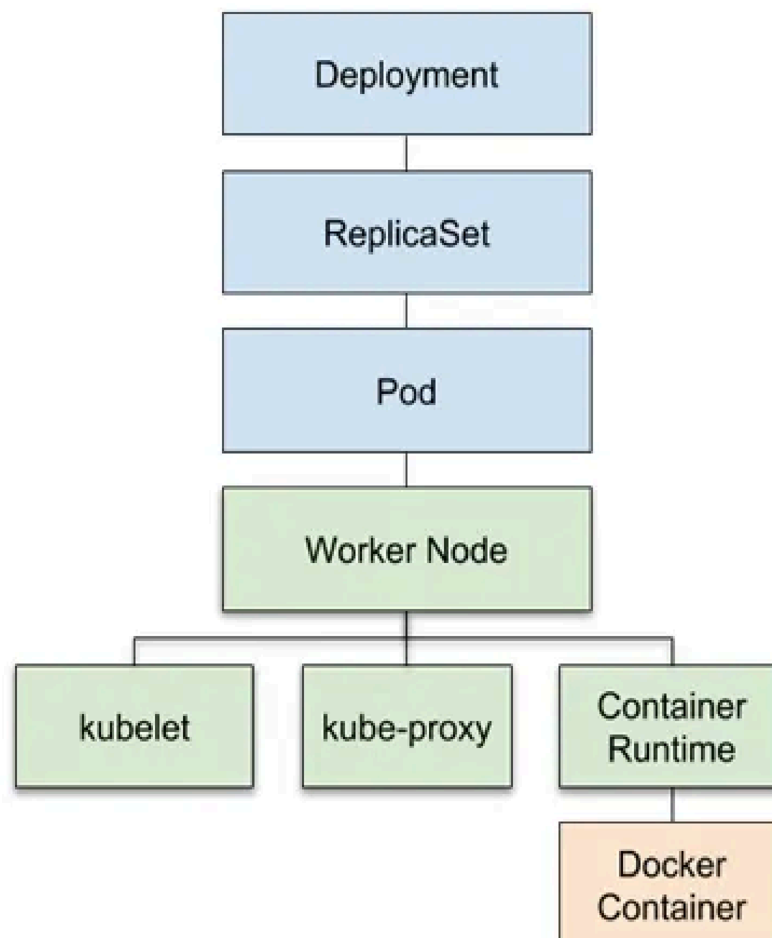
1. Secure: it should always follow the best security practices.
2. User-friendly: it should be easy to understand and operate.
3. Extendable: it allows for various providers of different tools and services and does not favor one over the other.

Kubernetes is a versatile platform for container orchestration, it has many benefits including but not limited to:

1. Portability: Kubernetes can run on any kind of computing platform, whether it’s a privately hosted server, public cloud, or a virtual machine, it can run in a container or directly on the machine or in a virtual machine or on a mix of all the previously mentioned types.
2. Scalability: the internal components are designed to be scaled up in case of a need to meet most demands
3. Consistent: The Kubernetes deployments are consistent across the available Kubernetes nodes since containers are complete static versions of the application which has all its dependencies included with it.

4. Enables DevOps: Kubernetes enables DevOps by allowing the operation team to automate running the containers and together with CI/CD it allows developers to go independently from committing their changes, having it automatically tested and built then shipping it to production environments where Kubernetes will take care of running the new version of their application.
5. Support for complex tasks: Kubernetes can run different complex kinds of tasks, whether it's a production application needing many applications to run or multiple different environments

2.5.1 Kubernetes abstractions



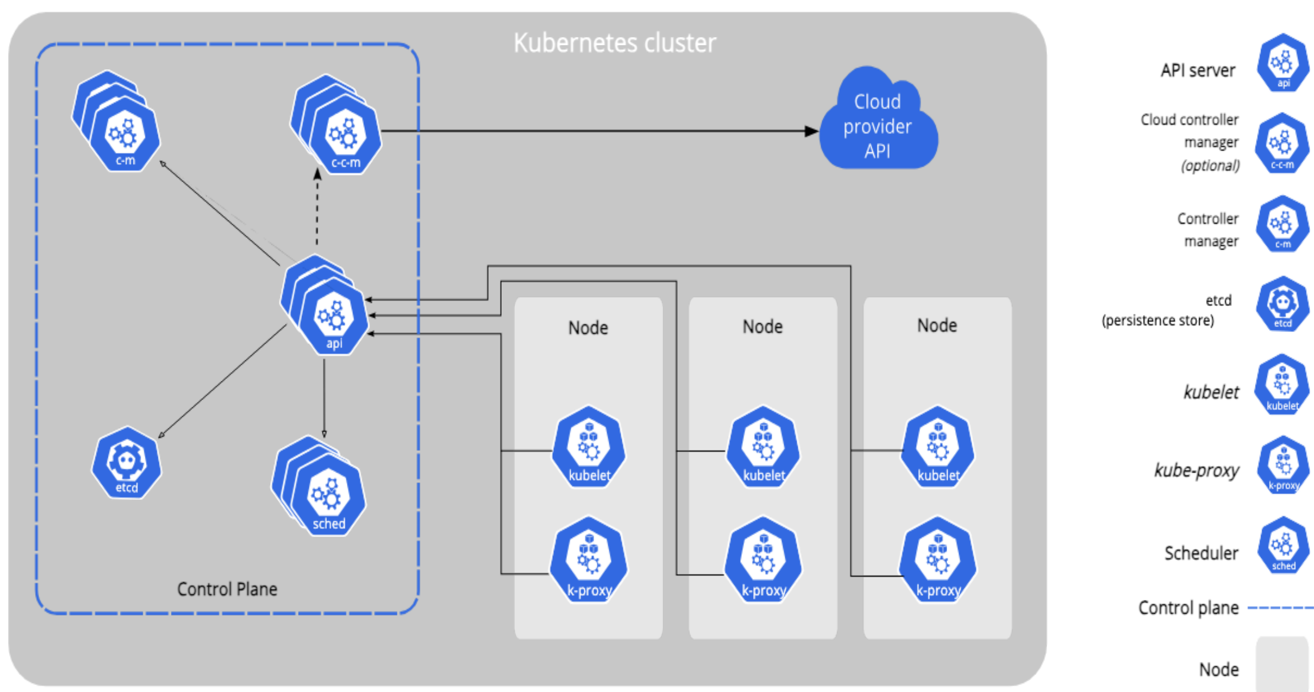
Listing 5. Kubernetes levels of abstraction [k8sabst]

Kubernetes has many terms for the applications, it is necessary to understand them properly before continuing:

1. Pod: A pod is a group of one or more containers, it is the smallest deployable unit in Kubernetes. A pod runs its containers in a shared context on the same node, it's almost like a group of applications running on the same host. [k8spod]
   Pods can have multiple containers, a container running alongside the main application controller is called a sidecar container which is usually running to perform some task related to the main application container.
2. ReplicaSet: A replica set controls a group of one or more pods, and aims to maintain the healthy pods count to be equal to the desired configured number of replicas. [k8srs]
3. Deployment: A deployment manages a set of pods via replica sets, DevOps teams define deployment and the desired state of the deployment which then creates the replica set, and the replica set eventually creates the pods that run the application containers. [k8sdeploy]
4. Service: It's a way to expose the application/pods, it creates a network endpoint that can be used to access one or more pods. It can be linked to your application pods dynamically so that when a new pod is created, traffic would go to that pod as well. [k8svc]
5. Secret: A Kubernetes secret is an object that contains a secret value, it can be mounted into the pod in different ways, this allows to keep the confidential info away from the container image. [k8ssec]
6. ConfigMap: it's a key-value pair object that is used to store a value that is not a secret, it can be mounted into the pod in different ways such as environment variables or configuration files in a volume or command-line arguments. [k8scfg]

2.5.2 Kubernetes Architecture

Kubernetes consists of a handful of internal components. When it is properly deployed, it is called a Kubernetes cluster.



Listing 6. Kubernetes architecture [k8scomp]

Listing 6 shows the architecture of a Kubernetes cluster, each component is critical to the operation of the Kubernetes cluster thus it's necessary to list each component and what they do:

1. API server: The API server is the front end of the Kubernetes control plane as it is responsible for serving API requests.
   kube-apiserver is the official API server for Kubernetes. it is made to be easily scalable by just increasing the replica count which allows it to take in more traffic, different replicas can also be run on different servers as long as it has connectivity with other Kubernetes cluster components. [k8sarch]

2. etcd: etcd is a distributed key-value store, which makes etcd the place where Kubernetes stores data and state.
3. Kube-scheduler: The component that is responsible for watching when new pods with no node assigned are placed on the cluster and trying to assign them a node to run on.

4. kube-controller-manager: In Kubernetes, controllers are processes that run constantly in a non-terminating way to watch for at least one resource type and try to make the resource reach the desired state. [k8scont]
   The kube-controller-manager runs controller processes.
5. cloud-controller-manager: is responsible for running controllers related to the cloud where the Kubernetes nodes run, if Kubernetes nodes are not running on a cloud then the cluster may not have a cloud-controller-manager.

In listing 5 we can see node-specific components which are also important to understand:

1. Kubelet: It is the agent that runs on each node, it ensures that the scheduled pods are running and healthy as expected.
   Kubelet only manages containers that are created through Kubernetes.
2. Kube-proxy: a network proxy for handling the networking with the rest of the cluster, it is an optional component. It handles the network rules that allow network communication between the pods on that node and the rest of the cluster.
3. Container runtime: As discussed in subsection 2.4 a container runtime like containerd is needed to run on each node to run the container images on that node.

Kubernetes cluster size can vary, can be a super small cluster where everything runs on a single personal use computer or it can be a large-scale cluster that spans different data centers.
Kubernetes can accommodate up to 110 pods per node, 5,000 nodes in a given cluster, 150,000 pods per cluster, or 300,000 containers. [k8slim]

Kubernetes objects are expressed in JSON or YAML files.
Kubernetes takes the input from the command line tools and updates its stored objects once they pass the validation and the different components of Kubernetes try to match the state of the objects to the state desired.

2.6 Incremental Analysis

Incremental analysis is a technique that is used in software engineering to optimize systems that need to constantly process input which changes over time.
Instead of processing the entire input every time, incremental analysis aims to identify only the sections that have changed and recompute the outputs. [waganalysis]

The incremental analysis's initial clear use was in software development environments (IDEs) as it can provide the foundation for consistent analysis results at interactive speed.
It can be used so that when a new change happens then it analyzes only the new changes and updates their outputs accordingly in the context of a syntax tree or a data structure that maintains relationships between elements.

Abstract Syntax Tree (AST) is a data structure that can be used to represent the current state of the input so that when a new change happens to the input such as changing the order of objects in a YAML file then the system doesn’t have to update the whole tree but only the subtrees which have been changed.
Each node in an Abstract Syntax Tree (AST) represents a syntactic construct from the language which could have subtrees under it that represent the components or sub-elements of that construct.
They are often used in compilers or static code analyzer tools because they provide a structured form of source code that can be understood semantically.

ASTs store the elements of the input in a tree where each nested relationship is stored in a subtree recursively and ignore syntactic elements such as whitespaces or quotations. [alsucptt]

2.6.1 Tree-sitter

Tree-sitter is a tool that is used to parse inputted files in different grammars and build an Abstract Syntax Tree (AST). It provides incremental parsing which depends on the concept of incremental analysis and the research in this field. [trestr]

Tree sitter supports parsing many different kinds of text files, ranging from programming languages like C or Go to key-value store languages like YAML. It makes the process of supporting a new language to parse and build an Abstract Syntax Tree (AST) a simple process, it needs formal grammar specifications to be defined for that language then it can be supported, the website for tree-sitters has a section on building a new parser.

It can be used in different programming languages through language bindings to the tree-sitter library which is written in pure C11 but has official support for usage in many languages such as Go, C#, and more.

Integrated development environments (IDEs) can benefit from Tree-sitter as it can work and parse almost any programming language, quickly as it doesn’t have to parse the entire file every time a change is made but only the new changes.

In this thesis, Tree-sitter is used to generate Abstract Syntax Trees (ASTs) for Kubernetes YAML manifests which enables the experiment to analyze the relationships between the Kubernetes objects and construct a dependency graph.

2.7 GitOps

GitOps is an operational framework that defines a set of practices that aim to automate most of the IT infrastructure objects' lifecycle, it is the DevOps answer for infrastructure automation. [gitops]

GitOps is used to manage modern infrastructure needs, it automates organizations' needs for frequent deployments and increases the deployment speed. GitOps aims to apply best practices to infrastructure objects such as version control using Git, code review, and CI/CD pipeline which automatically build, test, and deploy the new changes as they come.

GitOps depends on 3 core objects to be applied properly:

1. Infrastructure as Code (IaC): GitOps relies on storing the infrastructure objects' code in a Git repo, which protects from having different unknown manual changes which can be disastrous. The GitOps tool would try to match the infrastructure state with the desired state stored in the Git repository.
2. Merge requests/Pull requests: GitOps advises using merge requests for introducing new changes, as this ensures that new changes are tested and reviewed before merging and updating the actual environment state.
3. CI/CD: When a merge request is merged then the CI/CD would start to apply this change to the infrastructure objects. It should always make sure that the state in the Git repository is the actual desired state of the infrastructure objects.

GitOps tools such as ArgoCD and FluxCD take care of applying the desired state in the Git repository to the Kubernetes clusters and continuously monitor for changes in the Kubernetes cluster and try to overwrite to ensure always that the desired state is the currently applied state in the cluster.

These tools play a critical role during incidents as the DevOps teams update the infrastructure objects and due to their lack of contextual understanding of the changes that are being applied then they may waste time trying to apply changes in the wrong order of execution or apply syntactic changes which don’t affect the final state of the object.

This thesis explores working on a solution for this important which could DevOps teams time and save the organization time and costs during incidents where each second of outage or issue can cost potentially a lot of money.

2.7.1 ArgoCD

ArgoCD is a declarative GitOps tool, it is a continuous deployment (CD) tool for Kubernetes, and it automates the deployment of applications based on the configuration in the Git repository. [argo]

ArgoCD's most important concept to grasp the application concept, In ArgoCD an application is resources that are configured to watch for changes in a specific git repository at a specific directory so that it can apply the resources in this directory to the specified Kubernetes cluster. [argoconc]

The process of applying the desired state to the destination cluster is called a syncing process. ArgoCD's goal is to always make sure that all the applications are synced with the desired state and if there is any difference then it will automatically start an application (ArgoCD resource) sync.

3 Applying incremental analysis to GitOps

This section discusses the experiment to apply incremental analysis to GitOps, in order to solve the problem of the lack of contextual understanding in GitOps tools which leads to unnecessary deployments and slow deployment, which is an extremely important problem in modern-day DevOps as it affects the speed of deployment where each second of wasted time during an outage could cost a company hundreds or thousands of dollars.

In this context, Contextual understanding is the ability to distinguish between syntactic and semantic changes in a resource, semantic changes are changes that affect the config of the resources such as adding a new container to the pod or changing the docker image used for the container while syntactic changes are changes that do not affect the resources such as new whitespace or a change in the quotation style of a string, without the contextual understanding the GitOps tools would treat all changes as equal changes and try to reapply the syntactic changes.

Structural understanding is the ability to parse the Kubernetes resources and understand the relationship between all the resources, rather than just treating the YAML files as key-value pairs, structurally understanding makes the system understand how fields and resources relate to each other.

GitOps pipelines use tools like ArgoCD or FluxCD to manage Kubernetes clusters using Kubernetes manifests stored in Git repositories. These tools take the YAML files from the git repository and apply them to a Kubernetes cluster. However, this approach lacks contextual and structural understanding, which makes it unable to distinguish between syntactic and semantic changes.

Through the development of the experiment program, we can tackle the problem of the lack of contextual understanding in GitOps systems.

The Go programming language was chosen because both Kubernetes and ArgoCD are implemented in Go, allowing for better future integration with their APIs and internal components.

The system starts with scanning the supplied directory YAML files which are mostly Kubernetes infrastructure resources. It then leverages incremental Abstract Syntax Tree (AST) analysis using the Tree-sitter library.

Tree-sitter can differentiate between a semantic change (like updating a container image) and a syntactic change (like whitespaces or changed quotation style) which is crucial for minimizing unnecessary deployments.

Unlike YAML parsers, which treat documents as flat key-value structures, Tree-sitter constructs a full Abstract Syntax Tree (AST). This allows the system to understand the hierarchical relationships within the Kubernetes manifests and makes it easier to find specific values in the Kubernetes object, such as metadata, name, template, spec.volumes, or other deeply nested fields.
This structural awareness is for reliably extracting resource metadata and dependencies, even when elements are deeply nested.

After parsing, the program analyzes the ASTs to extract what kind of Kubernetes object they are typically one of a few (e.g., Secrets, ConfigMaps, PersistentVolumeClaims (PVCs), ServiceAccounts and can be extended to include more objects). This is necessary for understanding the dependencies since these objects are common to have other objects depend on them.

The program then recursively traverses the parsed AST node to locate possible references to other resources for example, a Deployment may reference a ServiceAccount, which in turn references a secret and the deployment also references a different secret directly itself and a config map.

These references are collected to create a dependency graph, where each node acts as a Kubernetes object and directed edges act as relations between edges.

To identify indirect dependencies between the resources, it is necessary to first compute the transitive closure of the dependency graph.

The transitive closure identifies for each resource, all the other resources that are reachable from it through a series of direct dependencies.

For example, if resource A depends on resource B, and resource B depends on resource C, then the system can infer that resource A indirectly depends on resource C.

Once the dependency information is available, the system can start by generating execution plans. These plans behave like a fully independent group of resources that mostly don't depend on other resources outside that group. These plans are outputted to a results directory. The GitOps tool processes the generated execution groups by running each group in parallel and executing inside the group sequentially, starting with the objects that have the fewest dependencies and ending with the resources that depend on other resources in that same group.

The objects whose relations are not recognized by the program are written as standalone files.

By doing so, the program lays the foundation for context-aware, parallelized GitOps operations, optimizing deployment times compared to the traditional model, in which the GitOps tool sequentially applied files regardless of dependencies and without much parallel processing of files.

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: my-config
data:
  config: "value"
```

Listing 7. my-config configmap

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:latest
          env:
            - name: CONFIG_VALUE
              valueFrom:
                configMapKeyRef:
                  name: my-config
                  key: config
```

Listing 8. nginx deployment

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: my-config
data:
  config: "value"
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: my-container
          image: nginx:latest
          env:
            - name: CONFIG_VALUE
              valueFrom:
                configMapKeyRef:
                  name: my-config
                  key: config
```

Listing 9. nginx deployment

The following Listing 7 and Listing 8 show a configmap called my-config and a deployment called nginx which are passed to the program and the output is shown in Listing 9.

The output shown in Listing 9 represents the result after applying the context-aware dependency analysis, ensuring that the my-config ConfigMap appears before the nginx Deployment.

Traditional GitOps tools like ArgoCD or FluxCD, Which treat all repository changes in the same way and rely on simple methods to differentiate changed files, lack the understanding of semantic changes (changes which impact the object state such as changing container docker image or adding a new container to the deployment) to objects like quotation marks or indentation and take these syntactic changes into account when applying thus wasting, sometimes even stuck in a loop trying to apply a syntactic change.

In a traditional GitOps system, if a change is pushed to the git repository which changes configmap value from single quotes to double quotes it would trigger a redeployment that doesn't do any real change, in some cases, this can lead to an apply loop where the GitOps tools are constantly trying to apply the changes which make no impact or change in the Kubernetes cluster.

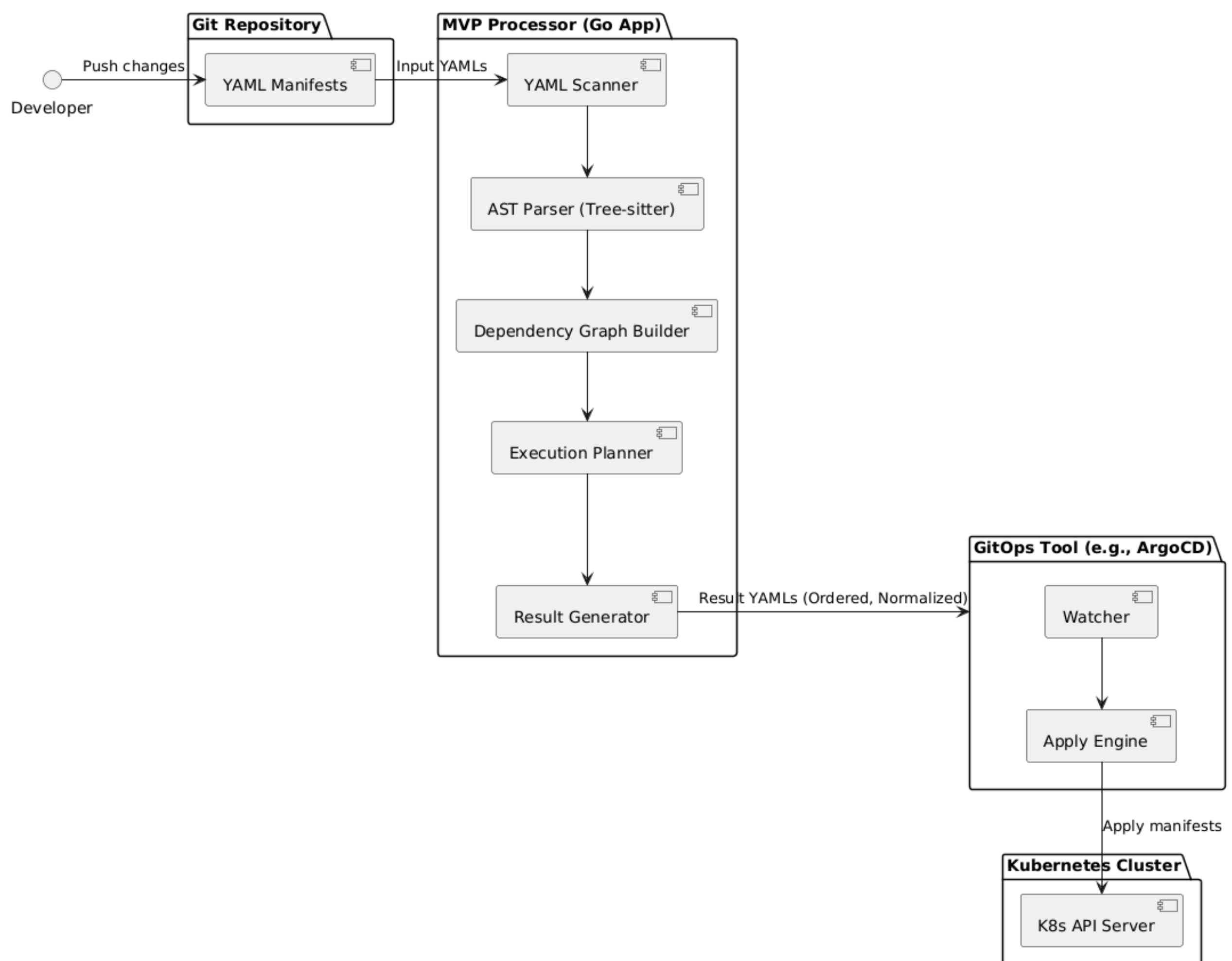
The program gets rid of the problem since it parses all Kubernetes objects and outputs them again after finishing the algorithm thus making sure that indentation and syntactic changes are always overwritten to look the same afterward thus not triggering a change in the GitOps tool.

Using incremental analysis parsing ensures that only newly updated parts of the Kubernetes objects need to be reanalyzed. This opens a big potential for real-time GitOps optimizations as repositories evolve without requiring complete reprocessing of all Kubernetes objects in the repository.

An important part of the hypothesis was that the incremental syntax-aware analysis would allow the program to understand and distinguish between semantic and syntactic changes in the Kubernetes resource definitions:

- Semantic changes: changes that affect the configuration of the resource, such as:
    - Modifying the container image.
    - Changing the deployment replica count.
    - Adding new environment variables.
    - Adding new volumes/volume mounts.
- Syntactic changes: changes that don't impact the resource or affect its behavior or results, such as:
    - Adding or removing whitespaces or changing indentation
    - Reordering fields in the YAML file
    - Changing quotation style for values in the YAML file

  These do not impact the actual state of the Kubernetes resource at all and shouldn't trigger a deployment.



Listing 10. Full system diagram

The full architecture of the system can be seen in Listing 10, the flow is as follows:

1. The developer makes changes and pushes the YAML Kubernetes manifests to a Git Repository.
2. The program runs and starts by running the YAML files through the YAML scanner.
3. YAML content is passed on to the Tree-sitter Abstract Syntax tree (AST) parser to analyze the structure of the Kubernetes objects.
4. The dependency graph builder tries to identify the relationship between the resources.
5. Dependency graph results are passed down to the execution planner which organizes the resources into groups of resources
6. The results generator outputs each execution plan into a single file and ensures that the output is normalized YAML which wouldn't be affected by syntactic changes.

The result files are stored in a separate results directory which is pushed to the Git repository again (from the CI/CD pipeline) and the changes are consumed by the GitOps tool like ArgoCD which has a watcher for Git changes (the results directory changes)

The program supports core Kubernetes objects like secrets, configmaps, PersistentVolumeClaims (PVCs), Services, and ServiceAccounts, and can understand the relationship between the Kubernetes objects using name-based matching, which allows for dependency visualization and an understanding of object interactions within the cluster.

The implementation laid down a foundation for relationship inference between Kubernetes objects by applying name-based matching methods. The program inspects the environment variable definitions in Deployments and reads what type of object the environment variable definition is referencing whether it's a secret or a configmap, then it resolves the object by name. It also detects service and deployment dependency by resolving the service selector labels fields in the service and finding the deployment which defines in its template the labels for the pods that match.

This approach enables static dependency resolution between different types of the most commonly used Kubernetes resources, this approach proved to be a practical method for capturing meaningful relationships between Kubernetes objects without inspecting the objects in runtime. The matching logic can be easily extended in the future with new resource

types and more matching methods could be added with minimal disruptions to system architecture.

The goal of the experiment is to provide a foundation for context-aware execution planning and to allow for adding more complex dependency modeling in GitOps workflows.

The current limitation of the proposed solution is that it lacks dynamic analysis of the resources in runtime, the program performs well in resolving dependencies between objects statically through parsing of the YAML objects but it does not understand the runtime behavior of objects and therefore lacks understanding of dependencies between objects in runtime like for example dependencies between Kubernetes operator and other objects as this can be complex behavior and differs from an operator to another or with objects that use default attributes.

Imagine a scenario where the operations team is trying to initialize a new cluster, they just created a new cluster, added ArgoCD to it, and pointed it to a base template with all essential applications and helm charts added to the repo including but not limited to:

- cert-manager helm chart which is a Kubernetes operator used to automatically generate SSL certificates for applications and ingresses
- Amazon Web Services (AWS) load balancer controller helm chart which is an application that will fail before installing cert-manager

Tools like cert-manager are commonly used as dependencies by other applications to work since they manage SSL-related logic which is commonly needed by many applications.

In standard GitOps workflows, customizing deployment logic and order requires explicit configuration. For example, in ArgoCD, sync phases and waves can be used to install one application before another.

Without such ordering, Helm charts may be installed before their dependencies are ready, which can result in partial or complete failure of the deployment.

If the resources in the Helm chart do not have readiness or health check configured, then the GitOps tool may still mark the deployment as successfully deployed.

Applications may depend on other applications at startup and if those dependencies are not

ready or do not have health checks configured, then they may not be detected by the GitOps tool which can cause silent failures or inconsistent states of the applications and pods that require manual investigation later on.

The issue is less likely to occur in small-scale clusters, but in a large-scale environment with multiple concurrent deployments and a large number of applications it could lead to delays or service disruptions during the execution for the GitOps system to execute all the changes.

Despite the challenges with dynamic analysis, the current solution integrates easily with GitOps tools and workflows. It can be deployed as a man in the middle between the GitOps tool and the Git repository via CD/CD pipeline like with GitHub Actions where it acts as a preprocessor that is triggered on commit and commits the merged and normalized YAML files to be used by the GitOps tool, the GitOps tool like ArgoCD can be configured to monitor the results directory and apply the objects in the results directory instead of the raw source files, which would enable parallelized and context-aware deployment of changes.

Due to the solution's simplicity, the GitHub action may take some time (possibly 1-2 minutes) while the runner starts and the job is picked up.

Instead of having a Github action that runs the application and then pushes the results to the Git repo, a more fitting approach could be to start a server that listens for webhooks for newly pushed changes and then parses the changes makes the results into execution plans and with a modified version of the GitOps tool that would directly allow for our processor application to send the execution plan directly to the GitOps tool, giving clear instructions on how many parallel executions to do, what are the execution plans to be executed and in what order to execute the changes in these plans.
Modifying the parallel processing capabilities of a GitOps tool is necessary anyway to utilize the application's benefits and allow it to finish executing changes in a timely manner.

This research introduces a new approach that aims to optimize the GitOps application process by applying incremental syntax-aware analysis using Tree-sitter parsing. Unlike traditional YAML parsers, Tree-sitter builds Abstract syntax trees (ASTs), which allows the modeling

and understanding of the structures and even the deeply nested ones in Kubernetes YAML manifests which is critical for accurately detecting semantic changes and building models for Kubernetes resources dependencies.

Tree-sitter is a general-purpose incremental parsing system that supports a wide variety of programming and data languages through formal grammar definitions, it can build concrete syntax trees for the inputted files. It makes the method developed in this research like dependency extraction and transitive closure possible to apply to different domains where a Tree-sitter grammar exists or if it doesn't exist then new grammar could be written.

It also shows how resource dependency graphs and execution plans can significantly improve deployments' efficiency. It suggests a new solution for handling fixed dependencies, proposes a possible improvement for dynamic dependencies, and discusses a possible implementation of integrating context-aware execution into already existing GitOps tools without modifying the source code for the tools or with modifications that could provide big improvements in the speed of execution.

The novelty part of this research is about combining incremental, grammar-based parsing using Tree-sitter and the focus on semantic changes and dependency graph modeling made specifically for GitOps real-world scenarios, while most GitOps tools just focus on file-level changes. By introducing context-aware analysis, the system can go beyond surface-level YAML which proposes a new foundation for intelligent context-aware GitOps orchestration.

4 Results

4.1 Hypothesis and Objectives

The goal of this research is to validate whether applying incremental syntax-aware AST analysis to GitOps workflows could improve deployment efficiency and reduce unnecessary deployments.

The hypothesis is that applying Abstract Syntax Tree (AST) analysis to Kubernetes objects would allow the system to:

- Detect meaningful (semantic) changes like image changes, configuration updates, or replica count while also ignoring syntactic changes like whitespace field order or quoting styles.
- Identify resources and their dependency map and group resources that depend on each other into one file to generate execution plans that can be used by the GitOps tool
- The grouped resources can be used for parallel deployment, which reduces the overall application time
- Output a normalized version of the input resources, which makes the resources unaffected by syntactic changes and avoids redundant redeployments due to irrelevant YAML changes.

The proposed system can detect unrelated changes and apply them in parallel, avoiding queuing delays and failed apply loops, unblocking high-velocity deployments in manifests repo where frequent commits are pushed.

In companies that have fast-paced environments, multiple teams might push many commits per day, multiple unrelated changes may be included in a single commit, or multiple commits may be pushed at the same time before the GitOps tool has completed processing previous changes. Traditional GitOps tools apply all resources sequentially.

As explained in subsection 2.1 some organizations may utilize monorepos, in this setup, all application teams may commit unrelated changes to the same manifests repo. The changes may affect separate services. GitOps tools will apply the changes sequentially even though they are completely unrelated to each other. The proposed system groups unrelated changes into different execution groups and allows them to be processed in parallel since they don't depend on each other, reducing deployment time and risk of conflicts.

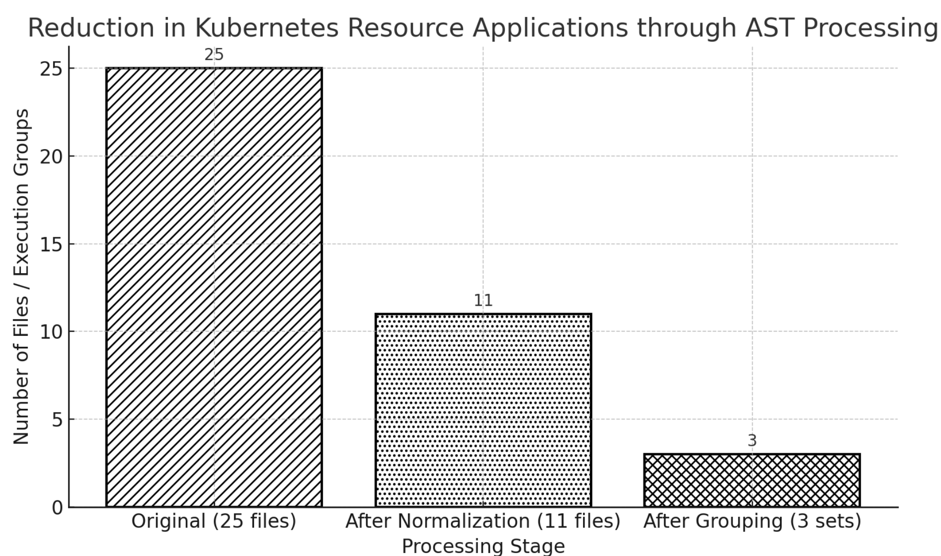
The goal of this section is to realize these improvements using a minimal viable product (MVP) written in Go, utilizing the Tree-sitter parsing library.

4.2 Observations

During the experiment, the program managed to successfully identify semantic changes in deployments (e.g., updated image tags or added environment variables or new configurations) and ignored syntactic changes like added whitespaces, and reordered fields. This behavior made the GitOps tool avoid unnecessary ArgoCD application syncs that would have resulted in no changes that would have been otherwise triggered. The output YAML was normalized across different changes and runs of the program which allowed for better diff readability in Git.

In a test directory of 25 Kubernetes YAML files (including 4 deployments, 3 services, 6 configmaps, 5 secrets, and 7 service accounts) it was observed that:

- ArgoCD attempted to apply all 25 resources sequentially, including ones that have syntactic changes.
- The experiment reduced the resources to be applied to 11 after normalizing the YAML file resources and grouping them, hence the GitOps tool detected the difference in the changes and found only 11 changes instead of the previous 25 which is approximately a 56% improvement in deployment time compared to traditional GitOps setup
- The system grouped the resources into 3 sets of resources to be applied in parallel, in a traditional GitOps system these would be applied sequentially so that would take 11 time units (1 for each resource to be executed) but in our system, this would be executed in 3 parallel runners, both first and second runner execute 3 resources each and the third would execute 4 resources which reduces the deployment time to approximately 4 time units which is approximately 63% improvement in deployment time compared to running the 11 resources sequentially.
- The total deployment time improvement is approximately 4 time units compared to 25 units with traditional GitOps systems.



Listing 11. Deployment time reduction with AST processing

For example, assuming a company offering software in multiple countries may provision a Kubernetes cluster per country, with each staging environment in a separate namespace. This setup allows teams to test country-specific logic independently while enabling multiple teams to work in parallel on tasks targeting the same country.

In the test setup where the company maintained 10 staging environments per country, the company operated in both Estonia and Finland, its structure was organized as follows:

- One Kubernetes cluster per country staging environments
- Each country had its repository where its staging environment changes were pushed
- Services manifest were stored under a directory structure formatted as /stagingX/service_name where stagingX represents the namespace and X is the number of the staging environment (staging1, staging2, etc …)
- Shared resources used by multiple services were placed in /stagingX/shared

The infrastructure team needed to update all pods at once, all pods have a sidecar container that is used for handling network-related tasks.

When a new version of the sidecar image is released, all manifests must be updated to reflect the new version.

- Assuming 10 applications per environment, approximately 100 application manifests required modification
- This delays the deployment process, as the GitOps tool has to sequentially update and wait for all these resources to be marked as healthy.
- The proposed system groups resources that depend on each other. Since most services don't depend on each other then it can group them into 10 groups (assuming in the worst case that all environment services depend on each other) or more
- This approach allowed the Infrastructure team to unblock the deployment queue and improve the deployment speed

The result of the experiment was that the proposed system saved at least 90% of the deployment waiting time since it utilized parallel processing of the changes. 90% deployment time reduction would mean that critical updates can reach production environments much quicker which reduces downtime and translates into significant cost savings, especially during incidents.

4.3 Test Dimensions:

The experiment evaluation tested the system under 2 main conditions:

1. **Dependency validation:**
   Resources were written in a way to reflect multi-level dependency between resources, for example:
   a. Deployments use environment variables from ConfigMaps and Secrets. The goal was to ensure that the program could recognize these dependencies and reorder the application of resources correctly.
   b. Custom ServiceAccounts were specified in deployments.
   c. Services for Deployments were linked via label selector

   The reason behind this was to make sure that the system can:

   - Parse the relationship between the resources with the Abstract Syntax Tree (AST)
   - Properly build a transitive closure.
   - Output the execution plan so the resources are executed in the right order, (Secret before deployment and deployment before service)
2. **syntactic change:**
   To test the program's ability to ignore syntactic changes, the same files had different types of changes:
   a. Changes in indentation and whitespace.
   b. Differences in string quotation styles like changing ‘value’ to “value” or value.
   c. changing the order of YAML fields (like spec before metadata)

   These changes shouldn’t trigger deployments. The system should be able to:

   - Ignore syntactic changes
   - Normalize output YAML consistently
   - Prevent the GitOps tool from triggering redundant apply operations

4.4 Comparison with Traditional GitOps Systems

Feature	Traditional GitOps tools (e.g., ArgoCD / FluxCD)	Context-Aware GitOps (With AST + dependency analysis)
Syntax Sensitivity	Sensitive to syntactic differences like whitespaces, quotes, and the order of items in the YAML file	Ignores syntactic changes using AST normalization
Execution Order	Applies files sequentially without a specific order	Group resources that depend on each other and put the resources in the right order for execution in the YAML file
Parallel execution	No	Yes
Dependency Awareness	No	Uses ASTs to build a dependency graph and compute its transitive closure
Integration with other tools	Native Git repository monitoring	Drop-in integration via running in a CI/CD pipeline before passing the files to the GitOps tool
Consistent YAML	Original YAML is kept, noise will affect deployments	Outputs normalized YAML that is clean and unaffected by YAML noise
Prune to errors	Yes, may apply items out of order which could cause objects.	No, as it applies the Kubernetes resources in the right order and respects dependencies
Complexity	Simpler architecture, but is complex and requires manual work in order to do ordered deployments	Slightly more complex, much smarter with understanding application logic, and requires no or little manual work to get ordered deployments to work

4.5 Possible Improvements

To improve system dependency detection and modeling, It is possible to add a new feature to allow users to explicitly pinpoint dependencies between objects using Kubernetes annotation, for example, the annotation could mention the kind and name like “ArgoCD.dependency: Deployment/name”. in addition to that helm chart-level dependencies could be incorporated, for example, “karpenter” helm chart

Future improvements can extend the support of the program to understand different types of object relationships to have better dependency modeling support, for example understanding how ingress objects depend on a service, which makes the ingress indirectly dependent on the deployment for which the service is for, and also on the objects that this deployment depends on like ConfigMaps or Secrets.

References

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

[devgitlab] GitLab. What is DevOps? https://about.gitlab.com/topics/devops/ Accessed: 2025-05-12

[devrelic] New relic. The Incredible True Story of How DevOps Got Its Name https://newrelic.com/blog/nerd-life/devops-name Accessed: 2025-05-13

[devatlas] Atlassian. 5 Key DevOps principles https://www.atlassian.com/devops/what-is-devops Accessed: 2025-05-13

[ciredhat] Red Hat. What is CI/CD? https://www.redhat.com/en/topics/devops/what-is-ci-cd Accessed: 2025-05-13

[k8sredhat] Red Hat. What is Kubernetes? https://www.redhat.com/en/topics/containers/what-is-kubernetes Accessed: 2025-05-13

[contaka] Akami. What Is a Container? https://www.akamai.com/glossary/what-is-a-container Accessed: 2025-05-13

[wallarmorc] Wallarm. Container Orchestration? https://www.wallarm.com/what/what-is-container-orchestration-7-benefits-and-4-best-tools Accessed: 2025-05-14

[collabk8s] Collabnix. What is Kubernetes? https://dockerlabs.collabnix.com/kubernetes/beginners/what-is-kubernetes/ Accessed: 2025-05-14

[devenv] codebots. What are environments in software development? A guide to the development, beta, and production environments. https://codebots.com/app-development/what-are-environments-in-software-development-a-guide-to-the-development-beta-and-production-environments Accessed: 2025-05-14

[k8scomp] Kubernetes. Kubernetes Components https://kubernetes.io/docs/concepts/overview/components/ Accessed: 2025-05-14

[k8sarch] Kubernetes. Kubernetes Architecture https://kubernetes.io/docs/concepts/architecture Accessed: 2025-05-14

[k8scont] Kubernetes. Controllers https://kubernetes.io/docs/concepts/architecture/controller/ Accessed: 2025-05-14

[k8sdeploy] Kubernetes. Deployments https://kubernetes.io/docs/concepts/workloads/controllers/deployment/ Accessed: 2025-05-14

[k8spod] Kubernetes. Pods https://kubernetes.io/docs/concepts/workloads/pods/ Accessed: 2025-05-14

[k8srs] Kubernetes. ReplicaSet https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/ Accessed: 2025-05-14

[k8ssvc] Kubernetes. Service https://kubernetes.io/docs/concepts/services-networking/service/ Accessed: 2025-05-15

[k8ssec] Kubernetes. Secrets https://kubernetes.io/docs/concepts/configuration/secret/ Accessed: 2025-05-15

[k8scfg] Kubernetes. ConfigMaps https://kubernetes.io/docs/concepts/configuration/configmap/ Accessed: 2025-05-15

[k8slim] Kubernetes. Considerations for large clusters https://kubernetes.io/docs/setup/best-practices/cluster-large/ Accessed: 2025-05-15

[waganalysis] Wagner, T. A. Practical Algorithms for Incremental Software Development Environments. Technical Report No. CSD-97-946, University of California, Berkeley, 1998.

[alsucptt] Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2006). Compilers: Principles, Techniques, and Tools (2nd ed.). Pearson.

[trestr] Tree-sitter. Introduction https://tree-sitter.github.io/tree-sitter/ Accessed: 2025-05-15

[k8sabst] Prakash Waikarh. Kubernetes Key Component and Concept https://prakashkumar0301.medium.com/kubernetes-key-component-and-concept-68c18e21cb95 Accessed: 2025-05-15

[gitops] GitLab. What is DevOps https://about.gitlab.com/topics/gitops/ Accessed: 2025-05-15

[argoconc] ArgoCD. Core Concepts https://argo-cd.readthedocs.io/en/stable/core_concepts/ Accessed: 2025-05-15

[argo] ArgoCD. What is Argo CD? https://argo-cd.readthedocs.io/en/stable/ Accessed: 2025-05-15

[atlasmono] Atlassian.Monorepos in Git https://www.atlassian.com/git/tutorials/monorepos Accessed: 2025-05-15

Appendix

I. Source Code

The source code for the experiment program is accessible from a public GitHub repository: https://github.com/muhammednagy/thesis

II. License

Non-exclusive license to reproduce the thesis and make the thesis public

I, Mohamed Moustafa Nagy Mohamed Salem,
(author´s name)

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

   reproduce, for the purpose of preservation, including for adding to the digital archives of the University of Tartu until the expiry of the term of copyright, my thesis

   Context-Aware GitOps ,
   (title of thesis)

   supervised by Bruno Rucy Carneiro Alves de Lima ;
   (supervisor´s name)

2. Grant the University of Tartu a permit to make the thesis specified in point 1 available to the public via the web environment of the University of Tartu, including via the digital archives, under the Creative Commons license CC BY NC ND 4.0, 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. am aware of the fact that the author retains the rights specified in points 1 and 2;

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

Mohamed Moustafa Nagy Mohamed Salem
12/05/2025