Department Informationstechnologie und Informationsmanagement

Strategies for Promoting Releases in GitOps Environments

Master Thesis

for obtaining the academic degree

Master of Science in Engineering

Master Program Cloud Computing Engineering

Submitted by:	Thomas Stadler, BSc
Matriculation number:	2110781014
Date:	June 1st 2023
Supervised by:	Thomas Schütz, MSc, BSc

Acknowledgements

Special thanks are due to my family, who enabled me to devote myself fully to this work, as well as to the rest of my studies. Furthermore, I would like to thank all the lecturers and students at the University of Applied Sciences who have had a strong influence on me during the past five years. I would also like to express my gratitude for the many new friendships I have made with students and lecturers.

Besondere Danksagungen gebühren meiner Familie, die mir es ermöglicht hat, mich voll und ganz dieser Arbeit, sowie der restlichen Studienzeit zu widmen. Des weiteren bedanke ich mich bei allen Lektoren und Studenten an der Fachhochschule, die mich während der vergangenen fünf Jahre stark geprägt haben. Ich möchte mich auch für die vielen neuen Freundschaften bedanken, die ich mit Studenten und Lektoren geschlossen habe.

Thomas Stadler, BSc

Eisenstadt, June 1st 2023

Abstract

Following a core concept of DevOps - reducing friction between engineering teams within the software development lifecycle (SDLC) - a deployment practice has emerged, which leverages the version control system Git for IT operations. GitOps is a set of principles for operating and managing software systems. The desired state of the managed system is - in its entirety - defined declaratively as code, which is continuously reconciled with the actual state by a controller. GitOps provides greater visibility into infrastructure state, a single source of truth with built-in audit history, reduced time to recovery and improved security, among other benefits.

A problem with the GitOps approach is the promotion of new software releases between environments, due to the asynchronous nature of GitOps deployments. Currently there is no standard practice and tooling for achieving promotions in a GitOps-native way. Hence, users are prone to use workflow/pipeline systems to achieve promotions. This thesis aims at addressing the problem of promotion of releases in GitOps environments. The problem statement was identified and motivated by interviewing practicing professionals who work in the GitOps field. Distinct problem items were defined, from which solution objectives were inferred. Abstract models of deployment environments as well as promotion workflows were designed. Based on these models, a standardized solution for the promotion of releases was designed and developed prototypically, adhering to the GitOps principles. The research was evaluated by comparing the functionality of the proposed prototype to the research objectives.

The results of this research address the given problem by providing a vendor-neutral solution for modeling environments and promoting releases between them with a GitOps-native approach. The proposed operator prototype and its implementation along with the demonstrated use in the proof of concept, describes one possible way of how the promotion of releases in GitOps environments can be designed. For future work, the prototype should be improved by doing research on its user experience and desired capabilities, within the framework of the design science research methodology.

Kurzfassung

In Anlehnung an ein Kernkonzept von DevOps - die Verringerung der Reibungsverluste zwischen Entwicklungsteams innerhalb des Softwareentwicklungszyklus (SDLC) - hat sich eine Deploymentpraxis herausgebildet, die das Versionskontrollsystem Git für IT-Operations nutzt. GitOps ist eine Reihe von Prinzipien für den Betrieb und die Verwaltung von Softwaresystemen. Der gewünschte Zustand des verwalteten Systems wird - in seiner Gesamtheit - deklarativ als Code definiert, der von einem Controller kontinuierlich mit dem tatsächlichen Zustand abgeglichen wird. GitOps bietet u. a. eine größere Transparenz des Infrastrukturzustands, eine sogenannte Single-Source-of-Truth mit integrierter Audit-Historie, kürzere Wiederherstellungszeiten und verbesserte Sicherheit.

Ein Problem des GitOps-Ansatzes ist die Promotion neuer Software-Releases zwischen Umgebungen aufgrund der asynchronen Charakteristik von GitOps-Deployments. Derzeit gibt es keine Standardverfahren und -werkzeuge für die Durchführung von Promotions in einer GitOps-nativen Weise. Daher neigen Anwender dazu, Workflow-/Pipeline-Systeme zu verwenden, um Promotions durchzuführen. Diese Arbeit zielt darauf ab, das Problem der Promotion von Releases in GitOps-Umgebungen zu adressieren. Die Problemstellung wurde durch die Befragung von Fachleuten, die im Bereich GitOps tätig sind, identifiziert und motiviert. Es wurden eindeutige Problemstellungen definiert, aus denen Lösungsziele abgeleitet wurden. Es wurden abstrakte Modelle von Deployment-Umgebungen sowie Promotion-Workflows entworfen. Auf der Grundlage dieser Modelle wurde eine standardisierte Lösung für die Promotion von Releases entworfen und prototypisch entwickelt, die den GitOps-Prinzipien entspricht. Die Forschung wurde evaluiert, indem die Funktionalität des vorgestellten Prototyps mit den Forschungszielen verglichen wurde.

Die Ergebnisse dieser Forschung adressieren das gegebene Problem, indem sie eine herstellerneutrale Lösung für die Modellierung von Umgebungen und die Promotion von Releases zwischen ihnen mit einem GitOps-nativen Ansatz bieten. Der vorgeschlagene Operator-Prototyp und seine Implementierung, zusammen mit der demonstrierten Verwendung im Proof of Concept, beschreibt einen möglichen Weg, wie die Promotion von Releases in GitOps-Umgebungen gestaltet werden kann. Für zukünftige Arbeiten sollte der Prototyp verbessert werden, indem die Benutzererfahrung und die gewünschten Funktionalitäten im Rahmen der Design Science Forschungsmethodik untersucht werden.

Contents

1 Introduction

This introductory chapter consists of the primary problem statement of the thesis, the definition of the research questions, a concise outline of the research methodology, as well as an overview of the thesis structure.

1.1 Problem Statement

Increasingly more organizations are adopting a DevOps culture (Sánchez-Gordón & Colomo-Palacios, 2018) to develop new applications and services at high velocity. After all, a culture that encourages shared responsibility, communication, transparency, and continuous and immediate feedback, helps to narrow the gaps between teams and thus accelerate the development process. In order to reduce friction between engineering teams who are involved in the software development lifecycle (SDLC), a practice called GitOps has emerged. It allows developers who are already familiar with the version control system Git, to easily deploy their applications to target environments in a self-service model. All sorts of IT infrastructure can be managed, purely by interfacing with declarative state definitions stored in Git. 84 percent of organizations that have embraced cloud native technologies have adopted practices and tools that adhere to GitOps principles (weave.works, 2023) (cncf.io, 2023).

GitOps as a practice for deploying and managing software systems has an unresolved problem, which is the process of promoting releases between multiple deployment environments (fig. 1.1).

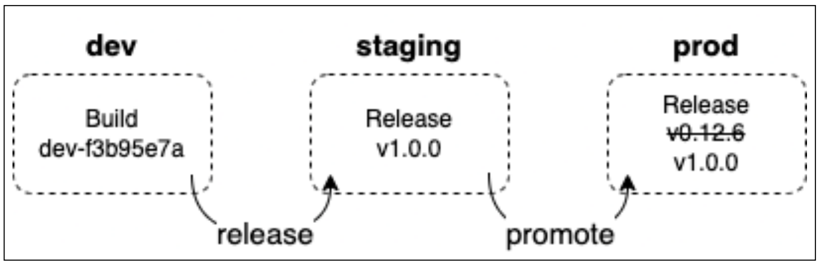


Figure 1.1: Promotion between environments.

Current GitOps tools do not provide an integrated solution for this process, nor do they provide any sort of abstraction for defining environments. Promotions are often achieved via hard-coded file copy operations, which is done manually or with a workflow/pipeline system. Furthermore, for each configuration/templating tool which is used, the modeling of different deployment environments, as well as the process of promotion, is unique. This results in the process of promoting releases with GitOps not being a streamlined task. A GitOps-native way for doing automated promotions between environments is not provided by the currently available open-source tools.

The given problem could be addressed by providing standardised models for defining deployment environments and promotion processes. An application programming interface (API) extension for Kubernetes, namely a custom resource definition, could provide abstract representations for these models. This would allow users to define their environments, and how they want releases to be promoted between them. Additional logic could be introduced into the promotion process, like specifying a rule which ensures that new releases must first pass certain environments or other objectives before being promoted to production. The abstraction would also enable transparent replacement of the configuration or templating tool, while keeping the desired state definition intact. Following the principles of GitOps, an operator would ensure the continuous reconciliation between the desired and the actual state of the resources.

The proposed solution of the problem should present a possible way of defining environments and promotion processes abstractly, onto which future work could build upon. Additionally the solution should provide a protoype of a toolkit, which could serve as an optional component in addition to existing tooling. Solving the problem of release promotion natively within the GitOps toolkit, would make the adoption of GitOps more appealing, especially for organisations, which have the need for many different environments. As a result this could generally accelerate the widespread use of GitOps and thus enable more organisations to develop higher quality software.

1.2 Research Questions

The overall goal of the thesis is to provide a solution to the problem of release promotion in GitOps environments. The solution shall consist of the abstract design of an operator capable of doing GitOps-native promotions between environments, as well as its implementation.

Large organizations in particular typically have many non-production and production environments such as: Development (Dev), Quality Assurance (QA), Staging-US, Staging-EU, Production-US, Production-EU. Usually new releases are automatically deployed to an environment, such as QA, by a workflow/pipeline system. A common task is to promote new changes, which are introduced by a new release, into subsequent or other environments.

To achieve the goal of the thesis, the following research questions (RQ) were identified:

- RQ 1: How can the promotion of releases in GitOps environments be designed?
  - RQ 1.1: How can deployment environments, as well as promotion processes be modeled abstractly?
  - RQ 1.2: How can the abstract models be used to implement a standardized solution for promoting releases?

1.3 Research Methodology

To achieve the main goal of the thesis and answer the identified research questions, a mix of different scientific methods is used. The primary method for creating empirical value - design and development of a software prototype - is the prototyping method as described by Riedl (2019). It is supported, especially for defining and motivating the problem statement, by semi-structured qualitative interviews according to the method of (Gläser & Laudel, 2010). In order to help with recognition and legitimization of the conducted research, the methodology for conducting design science (DS) research in information systems (IS) (Peffers et al., 2007) is applied. It consists of six activities:

- Activity 1: Identify Problem & Motivate
- Activity 2: Define Objectives of a Solution
- Activity 3: Design & Development
- Activity 4: Demonstration
- Activity 5: Evaluation
- Activity 6: Communication

In activity 1, the research problem of release promotion with GitOps is defined with the help of practicing professionals with working proficiency in the GitOps field. The problem is split into distinct problem items.

In activity 2, research objectives are inferred from the problem definition in activity 1. Each objective maps to a distinct problem item and provides a solution for it. This direct mapping helps with later evaluation in activity 5.

In activity 3, the design and development of the artifact, namely the *GitOps Promotions Operator* prototype, is described. The functionality of the prototype provides solutions to the defined research objectives.

In activity 4, the in-context use of the artifact is demonstrated in a proof of concept. The functionality of the prototypical operator alongside a practical use case is described.

In activity 5, the implementation of the artifact, and the demonstrated functionality, and how well it supports a solution to the problem, is evaluated. This is done by comparing the qualitative descriptions of the demonstrated functionality with the previously defined solution objectives.

In activity 6, as a final step, the whole conducted research is communicated by means of publishing it as a master thesis. Additionally, the prototype implementation is communicated to the Cloud Native Computing Foundation (CNCF).

1.4 Thesis Structure

This thesis is structured as follows. The contents of each major chapter are presented. Figure 1.2 outlines the structure of the thesis visually.

Chapter 1 - Introduction: After introducing the reader to the topic of GitOps, which can be seen as a good practice pattern within DevOps for deploying applications and generally managing systems and infrastructure as code, the problem is briefly stated and its importance is drawn attention to. The main goal of the thesis is outlined by stating the scientific research questions. The research methodology, namely the used scientific methods and the general approach, is presented in a short overview. Finally, to end the introductory chapter, the structure of the thesis as a whole is described.

Chapter 2 - Related Work: Existing literature and related work on the topic are discussed. The suggested best practices for handling the concrete problem of release promotion with the GitOps approach, are presented. Furthermore the related and similar tools that are available for doing GitOps promotions are presented. The related work chapter concludes with a summary of the chapter and draws attention to the differences that divide the research within this thesis from other work.

Chapter 3 - Theoretical Background: This chapter consists of general definitions of terms that are needed for further comprehension of the thesis. Moreover, some important concepts like DevOps and GitOps are described. Latest and ongoing trends like progressive delivery, as well as the role of Kubernetes in the current cloud native ecosystem, are discussed. The chapter Theoretical Background is finally rounded off with a summary, which draws attention to the key points.

Chapter 4 - Methodology: Firstly, the motivation and objectives of the chapter are outlined. Then the general approach for conducting the research of the thesis is presented, which comprises six activities. The initial two activities are supported by conducting interviews with practicing professionals, who have working proficiency in the GitOps field. The latter activities are primarily supported by the use of the prototyping method. All used scientific methods and their concrete applications are described lastly.

Chapter 5 - Interviews: The research problem of promoting releases in GitOps environments is defined and motivated, with the help of practicing professionals from conducted interviews. The problem is split up into distinct problem items. Next, for each problem item, a solution objective is defined, which provides a possible solution to a problem statement. Each objective defines clear requirements, which have to be met by the developed artifact, and which later help with the evaluation.

Chapter 6 - Prototype: The asynchronous nature of GitOps deployments, and where the developed prototype operator fits within this architecture, is brought forward. Next, abstract models for environment and promotion resources are designed. Then, the design of the Kubernetes custom resources for implementing the abstract models is described, which is followed by a mockup design of the custom resources, along with alternative mockups. As the last step, the prototypical controller logic for the environment and promotion controllers is presented. Once the design of the prototype is clear, and the implementation is developed, the prototype is demonstrated in a proof of concept. The demonstration is then evaluated against the solution objectives defined earlier.

Chapter 7 - Evaluation and Results: Includes qualitative results of the interviews, as well as observed implementations for the solution objectives and learnings from the prototyping process. Furthermore, the research results are evaluated in a holistic manner, by comparing the implemented solutions of the prototype to the research objectives.

Chapter 8 - Discussion and Interpretation: The thus far presented data is discussed with a holistic view, and an interpretation by the researcher on the conducted research as a whole is given.

Chapter 9 - Future Work: Topics and ideas which were not sufficiently handled within the thesis, as well as possible other ideas, which were brought forward by the research results or evaluation, are presented.

Chapter 10 - Conclusion: As a final chapter, a conclusion of the whole thesis is presented, which includes the most important key takeaways and research results.

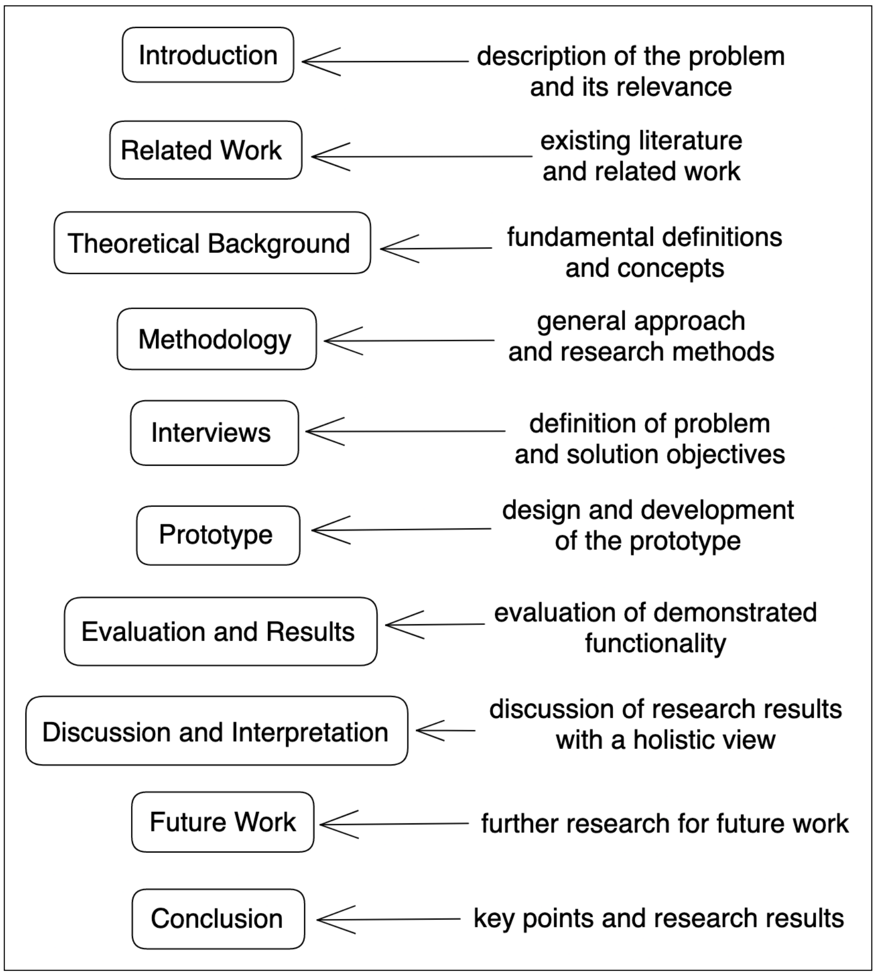

Introduction
description of the problem and its relevance
Related Work
existing literature and related work
Theoretical Background
fundamental definitions and concepts
Methodology
general approach and research methods
Interviews
definition of problem and solution objectives
Prototype
design and development of the prototype
Evaluation and Results
evaluation of demonstrated functionality
Discussion and Interpretation
discussion of research results with a holistic view
Future Work
further research for future work
Conclusion
key points and research results

Figure 1.2: Thesis structure.

2 Related Work

This chapter presents the related work by other researchers on the topic. Until now, there have not been any peer-reviewed publications in academic journals or conferences on the specific topic of modeling multiple deployment environments with GitOps and promoting releases between them. Some textbooks exist on the topic of environment promotion, which do not necessarily incorporate the GitOps approach. On the topic of promotion between multiple environments with the GitOps approach, there are different suggested good practices, as well as tools.

Beetz et al. (2021) recommend to avoid having to do promotions, by only having one environment. However, if a staging process with multiple environments is desired nonetheless, they suggest to use Git branches for stages and merging between the branches to achieve the integration of changes to other environments. This way it is observable which configuration is deployed to which environment or stage (Beetz et al., 2021).

Yuen et al. (2021) suggest that when implementing the GitOps approach, the CI workflow/pipeline additionally patches the new image version into the desired state, after previous stages are completed without errors. This is done with configuration management tools like Kustomize, which can patch a new image version tag in a manifest. This process achieves a promotion, as the GitOps engine notices the new desired state and deploys it. When the desired state is stored in a different Git repository, then it needs to be cloned in addition to the application build repository. They also suggest the implementation of observability metrics for the CI pipeline, in order to detect issues with the building and testing process (Yuen et al., 2021).

Yuen et al. (2021) also describe the rollback process which is needed, when bad releases need to be reverted. With the GitOps approach, changes to the state, as well as reverts to a previous state of the system can be achieved via operations on the Git repository, where the desired state is stored. Once it is updated, the GitOps engine takes care of the deployment. It is recommended to use the *revert* or *reset* functions of Git. These functions can be incorporated into a preconfigured workflow, which can then be triggered on-demand, or even automatically when a new release is observed to contain bugs or undesired behavior. It is suggested to setup creation of pull requests, which allows the configuration of one or many reviewers. For certain compliance standards such as in the payment card industry, new releases to production have the requirement for an approval of a second person. This is true for any release or change to a production system, be it a new release or a rollback (Yuen et al., 2021).

Kapelonis (2021) discusses the idea of modeling different deployment environments by using Git branches. He explains thoroughly why this approach is an anti-pattern and should not be used (Kapelonis, 2021). He also shares a multitude of suggestions and best practices about modeling environments and promoting releases between them. Different environments are modeled by customizing configuration in separate files and folders or Git repositories. For promoting between environments, basic file copy operations are suggested. He highlights that these simple file copy operations can easily be automated by an external system, like a CI/CD system. He suggests four categories of environment configuration. The application version, Kubernetes specific settings, mostly static business settings, and non-static business settings. While the application version and non-static business settings are promoted, Kubernetes specific settings and mostly static business settings are generally not promoted between environments (Kapelonis, 2022).

In the following, software projects and tools which provide similar problem solutions compared to that of this concrete research are presented. These include functionality for promoting releases or versions in GitOps environments. Some incorporate the same approach as proposed by this research, being the operator based approach, some offer command line interface programs or other interfaces to some sort of automation for promoting.

The company Weaveworks offer a solution in their enterprise GitOps offering to deal with multiple deployment environments. This functionality is limited to their closed-source Weave GitOps Enterprise offering. It allows the user to specify an application reference, which is a Flux HelmRelease resource, which can be deployed in a Pipeline like way, through many environments. Once an environment is successfully delivered with the new version, it sends a HTTP webhook to the next environment, or the management cluster, to trigger deployment to the next environment. The user may configure pull requests to be created for the promotion itself, which a human may review and approve. The pipeline allows for the ability to specify, that certain environments need to pass before a consecutive environment can be deployed to. Alternatively to promoting via pull requests, it may be configured to send notifications to an external system - which can then promote the application in whatever way (docs.gitops.weave.works, 2023).

"Kargo is a next-generation continuous delivery (CD) platform for Kubernetes. It builds upon established practices (like GitOps) and existing technology (like Argo CD) to streamline, or even automate, the progressive rollout of changes across multiple environments." (kargo.akuity.io, 2023) Kargo is still in very early development by the company Akuity. The tool is in the form of a Kubernetes custom operator, which provides custom resources for Environment, Promotion, and PromotionPolicy. Kargo allows users to define promotion processes in the form of updates of desired state, which is stored in Git. It supports updating Kubernetes manifests, container images, helm charts, and ArgoCD Applications, all by updating the desired state in a Git repository. It offers health checks for ArgoCD Applications to determine a healthy state of a particular environment. For the promotion process, Kargo commits to Git repositories (kargo.akuity.io, 2023).

"Telefonistka is a Github webhook server/Bot that facilitates change promotion across environments/failure domains in Infrastructure as Code(IaC) GitOps repos." (Wayfair-Tech–Incubator, 2023) It is designed to sync folders in Git repositories from source paths to target paths. When it detects changes that are not yet synced, it will create a pull request against the repository. It supports any directory structure of users GitOps repositories - it is unopinionated. It has drift detection as a feature. It can detect if there are changes in latter environments, which have not been promoted, i.e. are not in previous environments. Currently the tool can be run as a GitHub action, or as a standalone webhook server, preferably as a GitHub Application. In both cases it supports GitHub as the Git provider (Wayfair-Tech–Incubator, 2023). Telefonistka differs from the proposed prototype of thesis, because it is not designed with the asynchronous GitOps deployment in mind.

"gitops-promotion interacts with a Git provider to do automatic propagation of container images across a succession of environments." (XenitAB, 2023) The supported Git providers are GitHub and Azure DevOps. gitops-promotion is a command line interface program. It is best suited to be used as part of a CI pipeline/workflow (XenitAB, 2023). This tool only works with container image versions.

The k8s-promoter command line interface tool can promote Kubernetes manifests, by taking a Git commit range between two or more commits, and applying that to another environment. For the promotion itself it raises a pull request on GitHub (form3tech-oss, 2023).

The suggestion by Beetz et al. (2021) to avoid doing promotions with GitOps, points to the fact that there is a need for a software tool to do this. This thesis brings forward a software prototype which helps with the automation of promotions between GitOps environments.

The suggested practice by Yuen et al. (2021) ignores the fact that GitOps deployments are asynchronous, as opposed to synchronous pipeline processes. Conversely, the proposed prototype in this thesis incorporates asynchronicity in its design.

The rollback process for releases or promotions in GitOps environments, as described by Yuen et al. (2021) can also be done, when the proposed promotions operator of this thesis is part of the setup.

The design of the proposed prototype for this research considers the suggestions mentioned by Kapelonis (2021, 2022). Git branch-based environments are disregarded for the prototype design. Because of the different categories of environment configuration, the prototype design provides a way to promote specific files or directories, meaning possibly any arbitrary resource, while leaving other configuration specific to a certain environment.

Both projects Weave GitOps Pipelines, as well as Kargo were published before the beginning of the research of this thesis. They both follow a similar approach of having Kubernetes custom resources and respective controllers which provide automation. Weave GitOps Pipelines is created to work exclusively with Flux and its respective enterprise version. The enterprise offering Weave GitOps Pipelines offers a pipeline like functionality for GitOps deployment setups. It is not open-sourced. Kargo is strongly centered around ArgoCD. However it follows a similar approach as the proposed prototype of this thesis, in that it is based on the Kubernetes operator pattern. Conversely, the proposed promotions operator prototype of this thesis focuses on a vendor-neutral and standardized approach.

The presented related work shows that there is a need for an automated software solution for doing promotions in GitOps environments. Some researchers recommend avoiding multiple environments, because of insufficient available tooling. Several software projects try to provide a solution to the problem. They present possible approaches for promoting releases in GitOps environments.

Prior research on the concrete problem is mostly focused on doing promotions by using a workflow/pipeline system. After the build and test stages of a continuous integration process, the desired state is updated with the new version information. Conversely, this thesis presents the design and development of an operator that can do promotions between environments in an asynchronous and GitOps-native way. In addition, the prototype focuses on being neutral to vendors and tools, in order for users to use their various tooling together with the promotions operator. It will bring forward abstract models of environments and promotion processes, which are implemented in the proposed prototype operator. The prototype will assess the feasibility of defining deployment environments and promotion processes declaratively, following the GitOps principles.

3 Theoretical Background

In this chapter, the theoretical background which is needed for comprehending the topic, problem, and discussed material within this thesis, is presented. It consists of general definitions of terms and concepts, related theory, as well as various background information. DevOps and GitOps, environment promotion in the context of GitOps deployments, continuous deployment practices, progressive delivery and short-living environments, Kubernetes and its extensible architecture, as well as custom resources, controllers and operators are explained.

3.1 DevOps

Jabbari et al. (2016) define DevOps as a development methodology aimed at bridging the gap between development and operations, emphasizing communication and collaboration, continuous integration, quality assurance and delivery with automated deployment utilizing a set of development practices (Jabbari et al., 2016). DevOps conforms to a principle from the agile manifesto, which is about "Individuals and interactions over processes and tools". (Fowler, Highsmith et al., 2001) Human interactions between people are important and should be supported by using appropriate technologies, in order to decrease friction between people and teams (Verona, 2018).

Before DevOps, developing and running software were more or less two separate jobs performed by two different groups of people. Developers wrote software, which they then handed off to operations colleagues, who ran and supported the software in the production environment (i.e., served real users instead of just running it under test conditions). Like computers that needed their own floor in the building, this separation has its roots in the middle of the last century. Software development was a job for specialists, just like running computers, and there was little overlap between the two. The two departments had rather different goals and incentives, which often conflicted with each other. Developers like to focus on delivering new features quickly, while operations teams are more concerned with making services stable and reliable over the long term. The origins of the DevOps movement lie in attempts to bring these two groups together - to collaborate, create a common understanding, share responsibility for system reliability and software correctness, and improve the scalability of both software systems and the teams that build them (Arundel & Domingus, 2019).

One of the most important principles of DevOps is to allow the developer who brings new code changes which end up in new product releases, to have as much insight into the software development lifecycle as possible. So it is not just bringing developers and operations closer together, but to shift many processes into the developers hands. This is to give developers as much insight as possible, in order to decrease efficiency and productivity to eventually decrease the time-to-market for new product releases, features and bug fixes. Software should strive to run resilient in the production environment. This is essentially needed for organizations to continue to thrive in today's rapidly changing world.

Cloud native technologies have allowed for this movement to happen. An important aspect of cloud technologies is the self-service. When previously it was necessary to have many different teams and departments within a software development organization, each being responsible for individual parts of the software development lifecycle, it is now easier than ever before possible to reduce the amount of teams down to a minimum, in order to reduce friction between people communications and processes. This additionally gives the developers much better insights of what effect their code changes have on the end product or service, which customers consume.

While DevOps is about creating permanent cultural change in people and communications of an organization, GitOps is a specific continuous deployment practice. When DevOps is already adopted and techniques are in place, it becomes easier to also adopt GitOps practices (Beetz et al., 2021).

3.2 GitOps

GitOps is a set of principles and practices for operating and managing software systems. It provides greater visibility into infrastructure state, a single source of truth with built-in audit history, reduced time to recovery and improved security, among other benefits. It is a way of implementing Continuous Deployment and focuses on a developer-centric experience (Beetz et al., 2021). A version control system such as Git, which developers are already familiar with, is used for storing the desired state of a managed system. The state store is not strictly bound to Git. It can be any system for storing immutable versions of desired state declarations, which additionally provides capabilities for access control and auditing for each change in the version history (opengitops.dev, 2023a). The desired state of the managed system is the sum of all configuration data, that is needed to recreate the software system (opengitops.dev, 2023a). The desired state is defined declaratively as code. A declarative description is "a configuration that describes the desired operating state of a system without specifying procedures for how that state will be achieved." (opengitops.dev, 2023a) The desired state is continuously pulled by the GitOps controller and reconciled with the actual/current state. In figure 3.1 a visual representation can be seen of the main GitOps concept.

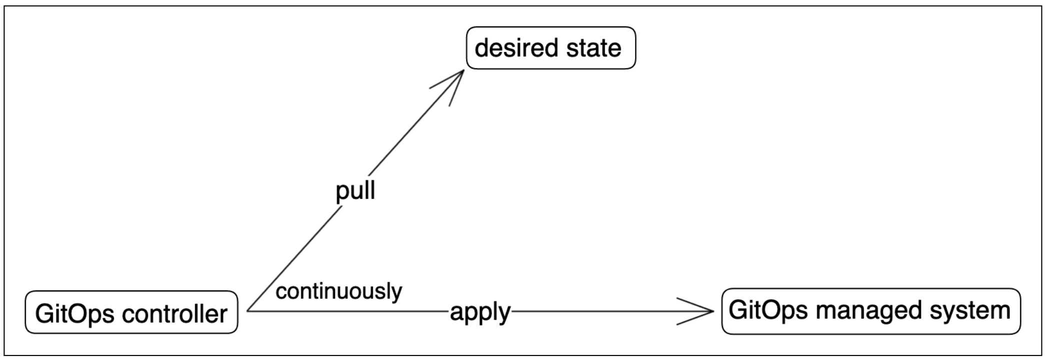


Figure 3.1: GitOps concept.

Reconciliation ensures that the actual state of the managed system is kept up-to-date with its respective desired state. The GitOps controller fixes any divergence upon notice. Divergence of actual state and desired state in the context of GitOps is called drift. (opengitops.dev, 2023a). This is different to traditional deployments without GitOps, where deployments typically only happen once after a change in the Git repository.

Infrastructure-as-Code which is stored in a Git repository, and executed by a workflow system, is not actually the concept behind GitOps. Infrastructure-as-Code is only one aspect of GitOps, namely the declarative one. There are three more principles of GitOps, which describe the desired state of a managed system. The OpenGitOps project defines the following four principles of GitOps: declarative, versioned and immutable, pulled automatically, and continuously reconciled (opengitops.dev, 2023b).

An important component of a GitOps toolchain is the configuration tool for the configuration files, i.e. Kubernetes manifests. Since the Kubernetes manifests are declarative and highly configurable in nature, their configuration and customization can be rather cumbersome. Many definitions are duplicated, so it is desirable to use variables, templates and the like, for making the configuration easier to use and maintain. Popular tools like Helm [1] and Kustomize [2] were created to help with configuration and templating.

"Kustomize introduces a template-free way to customize application configuration." (kustomize.io, 2023) It provides a way to customize base Kubernetes manifests with minimal additional overhead. It is built into kubectl and works purely declarative with raw manifests as input and output.

Helm is the de-facto standard package manager for Kubernetes. It provides a way to package the configuration and easily deploy those packages which can be highly configurable. Most third-party applications offer a helm chart for installation. It is based on using templates with variables and minimal logic, which can be rendered into plain Kubernetes manifests, usually at deployment time, as variables may be input for last-mile configuration (helm.sh, 2023).

The GitOps engine, agent or controller is responsible for the reconciliation of the desired state with the actual state in the target deployment environment. It adheres to the GitOps principles and is the primary tool to achieve the GitOps pattern. The different alternative GitOps engines offer similar functionality, and they have their own advantages and disadvantages. When extending the GitOps toolchain, it should not make a difference which specific tool from which provider is used. GitOps engines should work together with any other tool in the GitOps ecosystem. The most widely adopted GitOps engines and accompanying projects are those from the Argo [3] and Flux [4] projects.

Git providers, or Git servers (e.g. Github [5], Gitlab [6]) are a relevant component of a GitOps setup. For the most important functionalities like branches, commits, history tags, cherry-picking or merges, each Git provider functions the same. However, one of the most important aspects of GitOps is the Git pull request. As pull requests are not part of the open-source core of Git, each Git provider offers a different API for the pull requests. Integrations with the pull request API from Git providers therefore need to be implemented for each Git provider separately. Thus many software tools support only certain Git providers.

[1]https://helm.sh/
[2]https://kustomize.io/
[3]https://argoproj.io/
[4]https://fluxcd.io/
[5]https://github.com/
[6]https://gitlab.com/

3.3 Environments and Promotions

An environment - or GitOps environment - in the context of this thesis is defined as a target deployment environment for a given application; e.g. Development, Testing, or Production. Most of the time this is a Kubernetes cluster or namespace. In the context of the proposed Kubernetes custom resource definition environment, however it represents a folder/directory in a Git repository, which points to a deployment environment or cluster/namespace. Yuen et al. (2021) define the term as an environment, "where code is deployed and executed." (Yuen et al., 2021) This is true for the case of workload resources in a GitOps environment, because typically the compiled code is packaged into a container image, which is then run in a container runtime environment.

Promotion in the context of this thesis is defined as the process of promoting a new application or infrastructure version (release) to another deployment environment. In the context of GitOps and Git repositories, this often means changing declarative definitions of the desired state in Git repositories.

A release in the context of this thesis represents the process of publishing a new version of an application or software component to the users. When following GitOps practices, this usually means pushing a new Git tag to a Git repository, which triggers a workflow/pipeline, which as one of its steps publishes the software artifact of the new version of the application in an artifact registry.

GitOps environments can be modeled using different approaches. The most prominent approach is to have a folder in a Git repository per environment. This is straight-forward and easily compatible with the currently most used configuration and templating tools like Kustomize and Helm. Promotion would be just a file copy operation from one to another file or folder. However, for some promotions it is desired to only update a specific part of a file, with a specific type of information, e.g. patching a container image tag in a Kubernetes deployment resource. For each environment, or only critical ones, there could be a completely separate Git repository. Having a separate repository opens up the possibility to have a more strict separation of concerns, regarding permissions and access rights. Anyone who has access to a Git repository has read access to the entire repository tree. While the write access could potentially be limited by administrators or maintainers to certain folders, the read access will always be open for anyone who has access to the repository.

Another approach is to have a Git branch per environment. Promotion would be a Git merge from one to another branch. When taking this approach, and also having environment-specific configuration, it is possible, when not being careful, that a merge conflict happens, and the promotion would need to be solved manually. However when this approach is purely used for the purpose of staging, meaning environments are identical, but new releases are deployed to some environments first after other ones, it could in theory provide a seamless way of promotion by solely leveraging the built-in merge mechanism in Git. Branches can usually be restricted or limited to certain developers only, which would make it easier to implement the access permissions than the folder-per-environment approach.

This research focuses primarily on the modeling of folder-per-environment. The designed and developed operator prototype is based on the folder-per-environment model. However, if the requirement for branch-per-environment modeling exists in the future, support for it can potentially be added.

Continuous Deployment (CD) is the automatic deployment of successful builds to the production environment. Developers should be able to deploy new versions by either clicking a button, merging a merge request, or pushing a Git release tag (Arundel & Domingus, 2019). With the GitOps approach, Continuous Deployment works differently than with push-based deployments. In the following, push-based and pull-based deployments in the context of GitOps are explained.

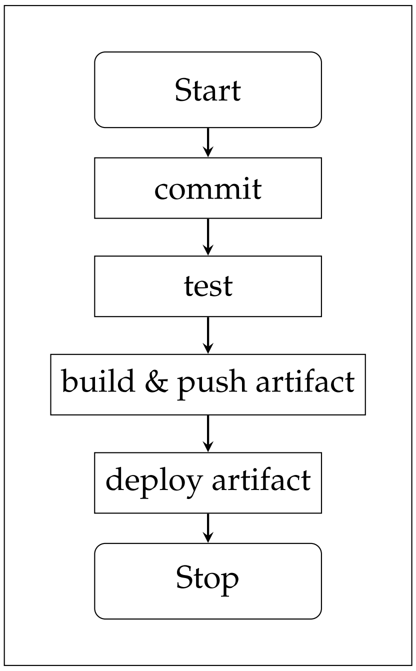


Figure 3.2: Push-based deployment.

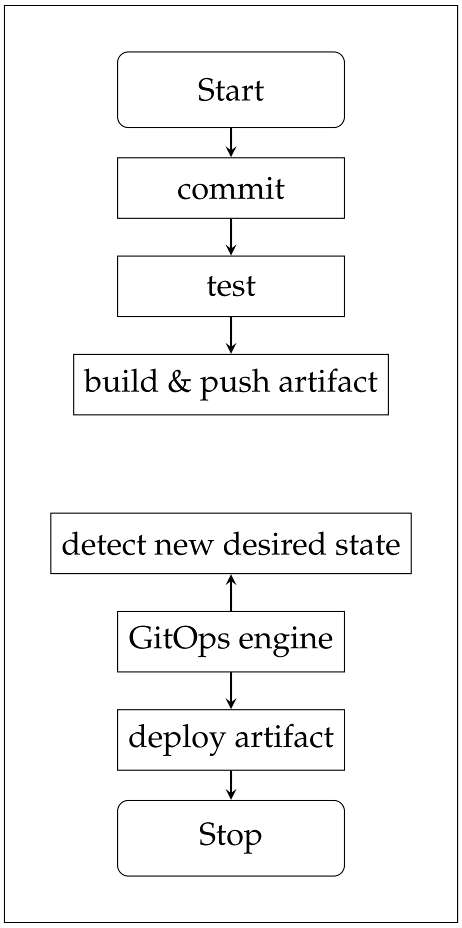


Figure 3.3: Pull-based deployment.

Without GitOps, Continuous Deployment is primarily push-based. This means, that a code change introduced as a commit to a Git repository by a developer, passes through each step in a CI/CD pipeline sequentially. Basically, a single process executes all tasks one after another, and has the knowledge of where the process is at at a given moment, and where it fails, and if it succeeds, it knows the status. These tasks might be automated tests of all sorts, automated builds of artifacts and finally some sort of uploading of the artifact to a registry. When Continuous Deployment is desired with this push-based approach, then a task would be appended to the end of the pipeline, which would deploy the new artifact to the target deployment environment. The workflow/pipeline system has knowledge over the status of the deployment, whether it failed or succeeded.

If multiple environments or stages were desired, another task would be appended to the pipeline in a consecutive manner. If some task fails during a pipeline run, the pipeline would be cancelled. This means that a commit, that fails certain automated tests, whill never be packaged into an artifact. Conversely, an artifact, that fails to deploy to a certain environment, will most likely not be deployed to consecutive environments, because of the stopped pipeline. It is challenging, if an artifact is successfully deployed, and looks like it works to the system, but in reality the user-facing service is actually not working successfully or with lower quality standards. The push-based deployment with CI/CD runs synchronously in one process. This is illustrated in figure 3.2. With current GitOps tooling, such a push-based deployment process can technically be implemented, however it is not advisable, as it violates the "pulled automatically" principle of GitOps.

With the GitOps approach and appropriate tools which implement its principles and pattern, the deployment process works in a pull-based manner. This means, that the GitOps engine, which often lives inside the deployment environment continuously watches the desired state for changes, and if changes occur, the new desired state is reconciled with the actual state, in order for them to match again. Because the deployment process is not done by a task at the end of the pipeline, the pipeline ends with a successful push of the artifact to a registry. At this point, typically a version updating tool like Flux Image Update Automation, which notices, that a new version is available in the artifact registry, patches that new version into the desired state stored in the GitOps repository. Next, the GitOps engine notices the change of the desired state and does the reconciliation, which finally ends up in a deployment. The processes that are responsible for deployment with the GitOps approach run asynchronously. This is illustrated in figure 3.3.

When a Continuous Deployment setup consists of multiple deployment environments, the process is usually to first deploy to the first environment, and if the deployment succeeds, the second environment is deployed to. Environments can also be seen as stages in this context, where previous stages need to pass, before subsequent stages are passed through. The purpose of having multiple environments and promoting releases through them is to detect and correct errors as early as possible in the development cycle (Yuen et al., 2021).

With push-based deployments described in section 3.3, where the host process of a workflow/pipeline typically knows the state of each deployment, it is not that big of a challenge to incorporate more deployment environments. For critical environments, that might be required to have a human approve new releases, the deployment task of the pipeline configuration could be set to manual. Via the workflow/pipeline system's interface, these deployments can be managed.

With the GitOps approach and pull-based deployments described in section 3.3, the process of deploying to multiple environments is more tricky with the currently available tools. While deploying the same new release or version of an application to multiple environments is not a problem, the promotion of a release to other or subsequent environments, depending on if the deployment to the first environment succeeded, is a bit of a challenge.

As previously described, the problem is that the pull-based deployment process with GitOps is asynchronous. Once the workflow/pipeline process uploads the artifact to the registry, the pipeline usually ends at that point. The component responsible for deployment is the GitOps engine, which is watching the desired state, and will pick up and continue the asynchronous deployment process, once it detects a new change in the desired state. There is usually an automated process which updates the desired state with the new artifact's version.

For the popular GitOps tools like Flux or Argo, there are components available with a functionality to update the desired state of a particular container image version with the currently latest available, or some other specification, like a semantic version specification.

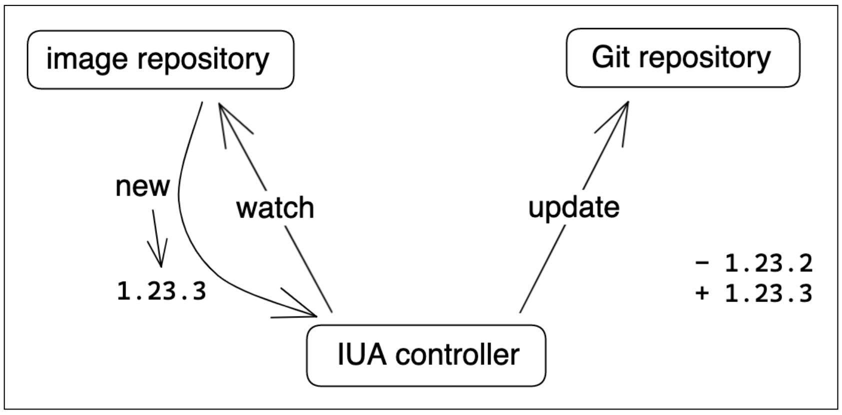


Figure 3.4: Image Update Automation.

It works the following way: A controller continuously watches the repository/registry of the container image for new versions. If the controller detects a new version with the specification it is configured (e.g. stable-*, 1.13.*), it will take that new verion tag and patch it to a declared container image which is specified in one or more YAML files in a Git repository. This tool makes it possible to achieve simple Continuous Deployment, meaning that a commit, a code change, by a developer can automatically be released to production. Ideally tests are run before releasing, in order to ensure quality. When the application is deployed to multiple environments, which is specified by multiple desired state declarations, i.e. folders in the GitOps repository, or separate repositories, such an image update automation can be configured for all environments. However, such a setup has its downsides. Namely, that the new release will be deployed to all environments at once. This is not always desired, and sometimes it is required to have some sort of manual release process between environments.

With the currently available tools in the GitOps ecosystem, it is not straightforward how you would setup such a promotion process, when multiple deployment environments are required. While it is not too difficult to achieve with push-based deployments and a synchronous workflow/pipeline, it is currently a challenge with pull-based deployments and GitOps.

An automated workflow for release promotion between environments, can be achieved in multiple ways, however there is no common or uniform practice for it at the moment. Depending on the used GitOps and workflow/pipeline systems used for a particular project or organization, such a setup for promoting releases with multiple deployment environments can differ by much. Replacing a component in the setup, e.g. the workflow/pipeline system or Git provider, can be quite challenging.

A way to do a GitOps promotion is to have a pipeline task, which can be a shell script, do some particular operation. This pipeline task could be configured to

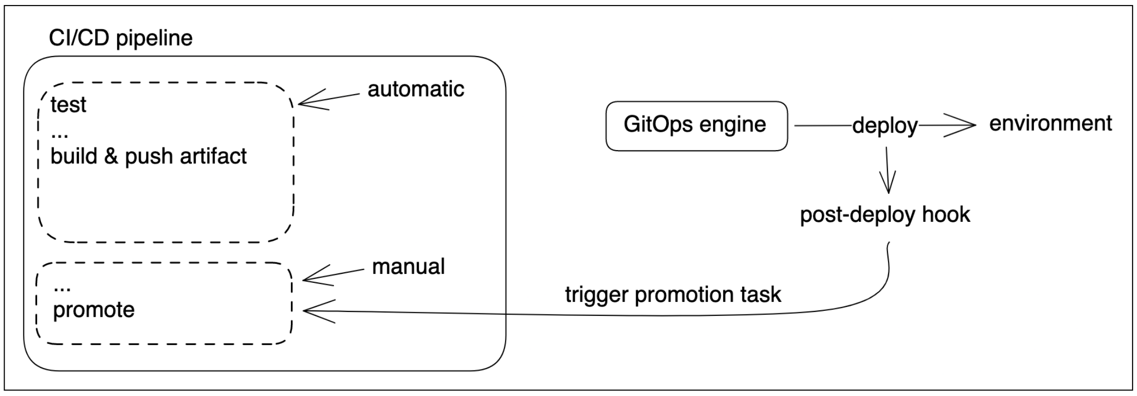
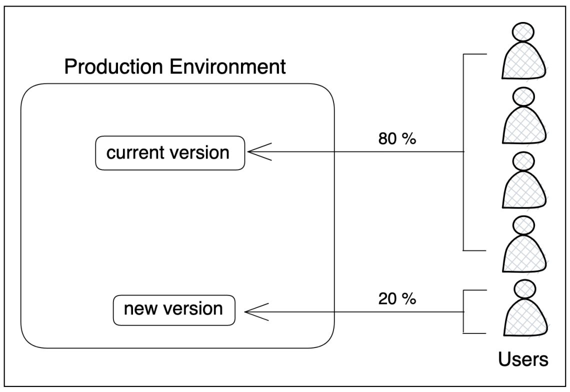


Figure 3.5: Promotion via post-deploy hook and pipeline task.

accept an incoming webhook, upon which it would be executed. The GitOps engine could be configured to execute a post-deployment hook, which it would run after a successful deployment to a particular environment. This hook could send a request to the pre-configured pipeline task, which would then be executed, and do the GitOps promotion.

3.4 Progressive Delivery & Short-Living Environments

With progressive delivery tools like Flagger [7] and Argo Rollouts [8], which offer advanced deployment strategies like canary and blue/green deployments, or A/B testing, it is now easier to ensure a bad release does not impact the end users as drastically. As an example, the named tools make it possible to release new versions to 10 percent of a specific region of end users, then this canary rollout is automatically tested and evaluated against metrics, if certain objectives for metrics fail to be met, the new release can automatically be rolled back.



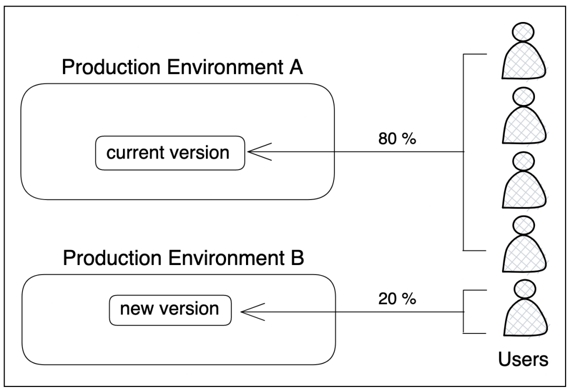


Figure 3.6: Example of gradual version rollout with progressive delivery.

Figure 3.7: Example of version rollout with multiple environments.

Since the progressive delivery tools allow for a more fine-grained segmentation of a single environment based on numerous parameters like client device type, user region, user type (e.g., developer, admin), registered or unregistered users, etc., the requirements to have a multitude of deployment environments have decreased for some organizations. In combination with feature flagging functionality, certain

[7]https://flagger.app/
[8]https://argoproj.github.io/rollouts/

new features can, for example, be released only to specific users. An illustration can be seen in figure 3.6.

The described segmentation of an environment with the progressive delivery tools, however might not be possible for every use case or organization. Financial institutions for example, where the requirement for absolutely zero errors hitting the production environment is top priority and business critical, to say the least, may still want to run multiple environments next to progressive delivery tools, to ensure an even higher quality and level of caution. This is illustrated in figure 3.7. Multiple environments typically mean a big increase in costs; with progressive delivery the need for multiple environments has decreased, and this way costs can be reduced.

Nowadays with the rapid development and releasing of new versions, there has become the concept of dynamic, short-living environments, or preview environments, deployments (Beetz et al., 2021). Hightower et al. (2017) recommend to deploy and test each new application version during development, a new commit or push to the trunk branch (Hightower et al., 2017). For every commit of an individual developer, a deployment environment may be provisioned for previewing the changes in a live environment that resembles the production environment as well as possible. This is to gather feedback as early as possible, and improve code quality, as well as remove bugs quickly. This short-living environment may be deleted after a short specified amount of time has passed.

3.5 Kubernetes and its extensible Architecture

The following section is about the role Kubernetes plays in the current cloud native ecosystem, and the extensible architecture of Kubernetes. "Kubernetes, also known as K8s, is an open-source system for automating deployment, scaling, and management of containerized applications." (kubernetes.io, 2023b) Although this is the primary use case of Kubernetes, and the reason why it was created initially, Kubernetes is increasingly being used as a base cloud native platform, to build other applications and platforms on top of. Kubernetes does many things out of the box, which good system administrators and operators have done manually in the past. Kubernetes does automation of operations, failover handling, centralised logging and monitoring, installation of security patches, backing up data, and more in an automated fashion, so humans can focus on their specific business related problems (Arundel & Domingus, 2019).

The architecture of Kubernetes provides a solid framework and platform, which is easily extensible. Developers may extend its API by specifying custom resources and controllers. There are several advantages when extending the Kubernetes API, in comparison to a plain REST API. Some advantages are hosted built-in API endpoints, state and configuration storage, request and policy validation, support for granular authentication and authorization, support for multiple API domains, versions, conversion between versions and API evolution (Kubebuilder-Authors, 2023).

When developing a Kubernetes-native application, many of the common capabilities which are often required for all applications, are being provided by Kubernetes itself, or otherwise easily consumable and integrated. These may include resource quotas, observability, monitoring, logging and tracing, configuration state storage, declarative APIs, control loops, and event and message queueing.

Kubernetes offers several different extension points and extension patterns. Most extension patterns however share the same basic design and principles. In general, a custom extension is a program which reads and/or writes to the Kubernetes API. By doing that it can provide useful automation. Since Kubernetes is based around a declarative API, where resources are defined as the desired state, and controllers are responsible for continuously reconciling this desired state with the actual state, it has shown to be a good pattern to design custom extensions in the same way (kubernetes.io, 2023c).

3.6 Custom Resources, Controllers and Operators

The concept of a controller and control loop within Kubernetes refers to the meaning from robotics and automation, where "a control loop is a non-terminating loop that regulates the state of a system." (kubernetes.io, 2023d) Controllers in Kubernetes are continuously running in a control loop, in specified intervals and sometimes internal or external triggers, and watch the actual state of the cluster, and make changes to it by interacting with the API, in order to bring the actual state closer to the desired state, like specified in the declarative definition. It is typically a good practice to have one controller be responsible for one resource, in order to help with separation of concerns. Controllers make changes to resources inside the cluster, like pods and deployments, but can also be responsible for resources external to the cluster, like APIs of the infrastructure provider (kubernetes.io, 2023d). A typical controller implementation can be seen in figure 3.8. As an example here, the deployment controller continuously ensures the desired state of three replicas; if it notices the desired state and actual state differ from one another, it does necessary actions to make them match again. When a controller has specific domain knowledge, or does certain tasks, which would usually be done by a human "operator", it is called an operator.

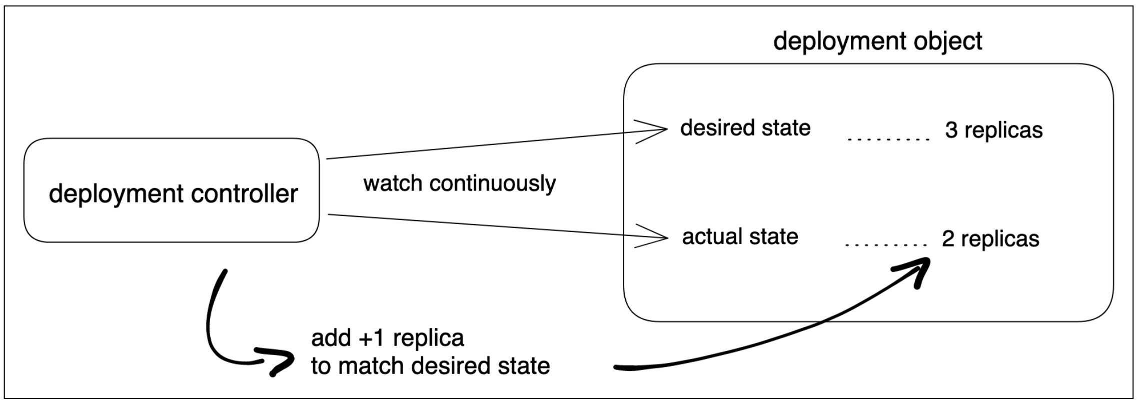


Figure 3.8: Typical controller in Kubernetes.

Operators typically are a set of controllers and custom resources with specific codified domain knowledge. All operational tasks - which would otherwise have to be done by a human operator - are written in code. This code, the controller logic, can then be automated. Examples for such operational tasks are backups and restoring of backups, error remediation, database migrations, etc. (cncf-tag-app-delivery-operator-wg, 2023). In overly simplified terms: An operator is a controller plus domain-specific operational knowledge. (cncf-tag-app-delivery-operator-wg, 2023)

The Operator Design Pattern represents a set of principles about managing complex applications and/or infrastructure resources, using domain-specific knowledge. The goal is to limit any manual work that needs to be done, and try to automate all operational tasks. This is done by capturing domain-specific knowledge in code, defining the desired state of resources and exposing them via a declarative API (cncf-tag-app-delivery-operator-wg, 2023).

To simplify the process of creating and maintaining a Kubernetes-native application in the form of an operator, there exist several operator frameworks. The most prominent framework is the Kubebuilder Framework. The Kubebuilder framework makes the process of extending the Kubernetes API an easy process for developers. An initial project can easily be boostrapped, allowing the developer to focus on implementing the custom resource definitions and controller logic. Any needed Kubernetes primitives, such as service accounts and RBAC permissions are automatically generated. Documentation for the OpenAPI resources are also generated from the code, which is the Go programming language. There exist rich libraries for interfacing with Kubernetes components, since Kubernetes itself is also implemented in the Go language (Kubebuilder-Authors, 2023).

With custom resources, the Kubernetes API can dynamically be extended during runtime, without the need to access its source code or recompile it. While a resource is "an endpoint in the Kubernetes API that stores a collection of API objects of a certain kind" (kubernetes.io, 2023a), a custom resource is "an extension of the Kubernetes API that is not necessarily available in a default Kubernetes installation" (kubernetes.io, 2023a). Custom resources can be dynamically registered and independently updated. Users can interface with its objects as they do with the Kubernetes built-in resources (kubernetes.io, 2023a).

3.7 Summary

In this chapter, the theoretical background on the topic was presented. General definitions of terms including DevOps and GitOps and releases, promotions, and environments in the context of GitOps were defined. GitOps related tooling and components were presented. It was shown how GitOps changes the architecture and process of Continuous Deployment, and how the promotion of releases is achieved without and with the GitOps approach. Emerging patterns like progressive delivery, as well as the concept behind short-living environments were described. The power of Kubernetes as a cloud native platform and its use cases beyond container orchestration were presented.

4 Methodology

In this chapter, the general methodological approach for the concrete research, as well as the used scientific research methods and their application are presented. The methodology is guided by the applied methodology framework for design science research in information systems.

4.1 Motivation and Objectives

The chapter Methodology starts off with an outline and brief description of each activity of the general approach of this concrete research. The research approach is framed around the design science research methodology by (Peffers et al., 2007), which builds a structure and guideline in which the whole research is based on. The purpose of following this methodological framework is to help with the recognition and legitimization of the conducted research. For the problem identification, the semi-structured interview is used as a scientific method, and for the core of the research, namely the design and development of a software artifact, the prototyping method is applied.

All the used research methods are described, and especially how they are applied within this research. The chapter ends with a summary of what has been presented, as well as the key takeaways, which are needed for comprehension of the following chapters of this thesis.

4.2 General Approach

To achieve the main goal of the thesis and answer the identified research questions, a mix of different scientific methods will be used. In order to help with the recognition and legitimization of the conducted research, a commonly accepted framework, namely the methodology for conducting design science (DS) research in information systems (IS) (Peffers et al., 2007) will be applied. It consists of six activities (fig. 4.1):

- Activity 1: Identify Problem & Motivate
- Activity 2: Define Objectives of a Solution
- Activity 3: Design & Development
- Activity 4: Demonstration
- Activity 5: Evaluation
- Activity 6: Communication

The process is structured in a nominally sequential order. For this concrete research, the problem-centered initiation is chosen as the entry point, thus the process will begin with activity 1. Afterwards the research will proceed sequentially, because the idea of the research resulted from observation of the problem (Peffers et al., 2007).

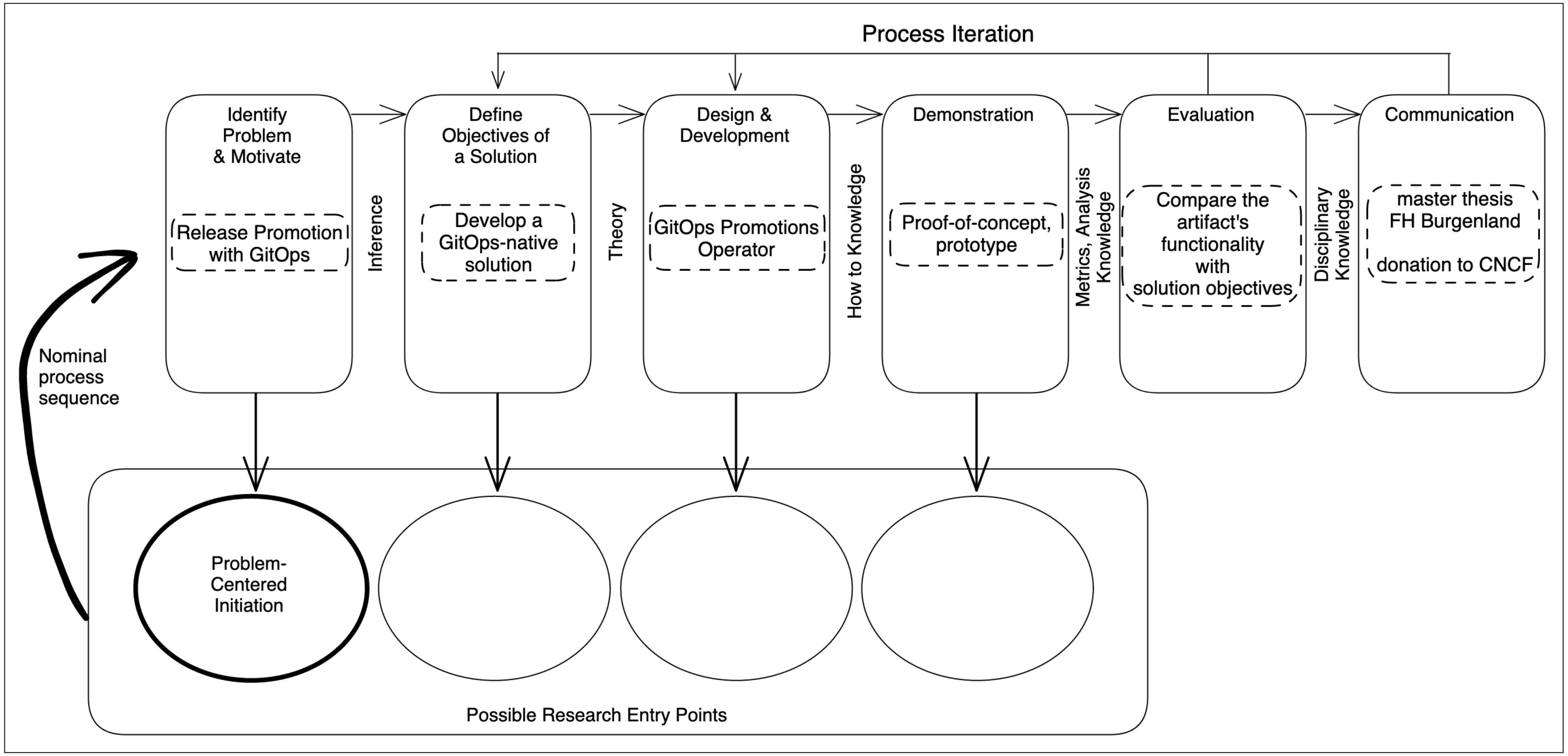

Process Iteration
Identify Problem & Motivate
Release Promotion with GitOps
Inference
Define Objectives of a Solution
Develop a GitOps-native solution
Theory
Design & Development
GitOps Promotions Operator
How to Knowledge
Demonstration
Proof-of-concept, prototype
Metrics, Analysis Knowledge
Evaluation
Compare the artifact's functionality with solution objectives
Disciplinary Knowledge
Communication
master thesis FH Burgenland
donation to CNCF
Nominal process sequence
Problem-Centered Initiation
Possible Research Entry Points

Figure 4.1: DSRM Process for this thesis (adapted from Peffers et al., 2007).

4.2.1 Activity 1: Identify Problem & Motivate

In activity 1, the research problem of release promotion with GitOps is defined. This is accomplished, by seeking knowledge of the state of the problem from practicing professionals. This is done by conducting semi-structured interviews, as described in section 4.3.1, as well as analysing prior written literature. To assist later evaluation, the problem is conceptually broken down into distinct items (fig. 4.2).

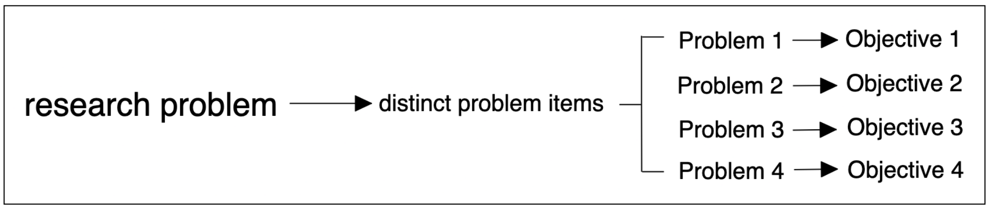


Figure 4.2: Inference of objectives from problems.

The value of a solution is highlighted, in order to help the audience of the research understand the reasoning associated with the researcher's understanding of the problem (Peffers et al., 2007).

4.2.2 Activity 2: Define Objectives of a Solution

In activity 2, research objectives are inferred from the problem definition in activity 1. Each objective maps to a distinct item from the problem specification (illustrated in Figure 4.2), which helps with later evaluation in activity 5. In practice, a research objective is a qualitative description of how a new artifact is expected to support a solution to the problem definition.

4.2.3 Activity 3: Design & Development

In activity 3, solutions for the previously defined objectives are designed and developed by means of producing an artifact. This is achieved by determining the artifact's desired functionality and its architecture, followed by actually creating the artifact (Peffers et al., 2007). In practice this means that: Abstract model definitions are designed; with the specification of the model definitions in place, the *GitOps Promotions Operator* prototype is developed as an artifact.

4.2.4 Activity 4: Demonstration

In activity 4, the in-context use of the artifact is demonstrated in a proof of concept. The prototype operator is used in a practical use case of a promotion in a GitOps environment. A description of the setup, the use and observed functionality is demonstrated.

4.2.5 Activity 5: Evaluation

In activity 5, the implementation of the artifact, and how well it supports a solution to the problem, is evaluated. This is achieved by comparing the objectives of a solution to actual observed results from use of the artifact in the demonstration (Peffers et al., 2007). In practice this means, that the functionality of the artifact implemented in the prototype in activity 4, is compared with the solution objectives from activity 2.

4.2.6 Activity 6: Communication

In activity 6, as a final step, the whole conducted research is communicated by means of disclosing the problem and its importance, the artifact and its utility and novelty, and the demonstration accompanied by the evalution results (Peffers et al., 2007), within the publication of a master thesis at the University of Applied Sciences Burgenland [1]. In addition, it is communicated to relevant audiences such as the GitOps Working Group [2] of the CNCF.

4.3 Research methods

In the following section, the used research methods semi-structured interview by Gläser and Laudel (2010), design science research methodology for information systems by Peffers et al. (2007), and prototyping as described by Riedl (2019) are explained in detail.

4.3.1 Semi-structured Interview

For Activity 1: Identify Problem & Motivate, semi-structured interviews with working professionals are conducted. For this research method, the suggested practices and guidelines from the textbook of Gläser and Laudel (2010) are used. These interviews have the primary goal of aiding with problem identification and motivation, because prior written scientific literature is insufficient on the topic. For conducting the interviews, a semi-structured interview guide (Appendix A) is used.

The interview guide is created on the basis of preliminary considerations. The use of a guide facilitates the comparability of several interviews, but still leaves enough room for spontaneous statements. The advantage of a guide is that the researcher can stick to concrete questions. Follow-up questioning is allowed and the questions are mostly open. The open questioning enables the interviewee to present their point of view (Berger-Grabner, 2016).

Preparing the Interviews: As an initial step, a semi-structured interview guide is developed. It serves as a rough guideline for the structure of the interview process. Additionally it encompasses the content-related interview questions, which are specific to the given research problem.

[1] https://www.fh-burgenland.at/

[2] https://github.com/cncf/tag-app-delivery/tree/main/gitops-wg

The selection of the interview partners is based on the questions mentioned in Gläser and Laudel (2010):

- Who has the information relevant to this work?
- Who can describe it precisely?
- Who is most willing to be interviewed?
- Who is available?

The pool of potential interview partners is mainly oriented around contacts from friends and acquaintances, who are working professionally in the GitOps field. The contact to the interview partners is done via Slack [3] or e-mail. Prior to conducting the interview, the interview guide is sent to the interviewee.

Conducting the interviews: Appointments for the interviews are made with each interview partner in advance. Eventually, the interview is conducted with a web video conferencing tool like Zoom [4]. The interviews are recorded for later transcription. The semi-structured interview guide is used as a guideline for the interview process as a whole. However, the actual structure of the interview may differ for each individual interview and according interview partner.

Transcription and post-processing of the interviews: The output of the interviews are in the form of audio-video recordings. These are transcribed word-for-word into a written form afterwards. The transcriptions can be seen in Appendix B. The content-related answers of the interview partners are taken into consideration for Activity 1: Identify Problem & Motivate, and there further processed.

4.3.2 Design Science Research Methodology

The design science research methodology (DSRM) has already been thoroughly described and applied in the earlier section 4.2 General Approach of this thesis. A concise definition is stated here nontheless. Peffers et al. (2007) describe DSRM as follows:

> "We propose and develop a design science research methodology (DSRM) for the production and presentation of DS research in IS. This effort contributes to IS research by providing a commonly accepted framework for successfully carrying out DS research and a mental model for its presentation. It may also help with the recognition and legitimization of DS research and its objectives, processes, and outputs, and it should help researchers to present research with reference to a commonly understood framework, rather than justifying the research paradigm on an ad hoc basis with each new paper." (Peffers et al., 2007)

"DS is of importance in a discipline oriented to the creation of successful artifacts." (Peffers et al., 2007)

The DSRM framework is used for this research, in order to help with evaluation and legitimization of the conducted design and development of the software

[3]https://slack.com/
[4]https://zoom.us/

artifact. The evaluation of the software prototype is done by comparing the observed functionality in the demonstration with the defined solution objectives, in a purely qualitative and descriptive manner.

4.3.3 Prototyping

For the research method prototyping, which is applied in Activity 3: Design & Development and Activity 4: Demonstration, the literature of Riedl (2019) is used. The terms prototype, prototyping, prototyping cycle and phase scheme are defined in the following (Riedl, 2019):

A prototype is an executable model of the planned product, produced with little effort and easy to modify, which can be tested and evaluated by the future user (Riedl, 2019). Prototyping is the entirety of activities, methods and tools required to produce prototypes (Riedl, 2019). A prototyping cycle is a sequence of steps consisting of using, evaluating and modifying a prototype (Riedl, 2019). A phase scheme is the ideal-typical structure of a project in sections of logically related tasks including the methodology, methods and techniques of task solution. Synonym: phase model (Riedl, 2019).

Types of Prototypes

There are several different types of prototypes, however they each share common traits. A prototype

- can be developed quickly and at a low-cost.
- provides a functional and executable model for evaluation by future users before actual implementation.
- is easy to modify and extend.
- does not necessarily represent the system completely.
- serves as a means of communication between the developers and the users.
- can be evaluated by all stakeholders involved in the planning process (Riedl, 2019).

According to the type of prototype, a distinction is made between complete and incomplete prototypes as well as disposable and reusable prototypes (Riedl, 2019). A full prototype is a prototype that makes all the essential functions of the information system to be created fully available. The experience gained during planning and during use and the prototype itself form the basis for the final system specification (Riedl, 2019). An incomplete prototype is a prototype that allows the usability and/or feasibility of individual components of the information system to be created (e.g., the user interface) to be assessed (Riedl, 2019). A disposable prototype is a prototype that serves only as an executable model; it is not used directly for the information system to be created (Riedl, 2019). A reusable prototype is a prototype that meets all quality requirements and from which essential parts can be adopted in the information system to be created (Riedl, 2019).

Another systematic approach, based primarily on the intended use of the prototype, distinguishes between demonstration prototype, laboratory sample and pilot system (Riedl, 2019). The demonstration prototype supports project initiation or project acquisition; it should convince the potential client that the desired end product can be built or that its handling corresponds to what the future users imagine. A prototype for this purpose therefore has the characteristics of the incomplete prototype and usually also of the disposable prototype (Riedl, 2019). The laboratory sample is primarily used to clarify design-related issues; it is derived from the model of the application task and from an existing specification. The design of the end product should be essentially identical, or at least comparable, to that of the laboratory prototype. This requirement can refer to the architecture and/or to the functionality. (Riedl, 2019). In the pilot system, the strict separation between prototype and end product is eliminated. At a certain level of maturity, the prototype is used productively at individual workplaces and is further developed. It is initially incomplete and reusable; as it matures, it becomes complete (Riedl, 2019).

For the concrete research of this thesis, the type demonstration prototype is used.

Evaluation of Prototypes

Evaluating prototypes requires that there is an evaluation strategy agreed between the developers on the one hand and the users on the other; prototypes must be developed with the evaluation strategy in mind. The evaluation strategy specifies the *what* and the *how* of the evaluation. This includes agreements on the time intervals in which modified versions of the prototype are made available for evaluation (evaluation cycle). From a methodological point of view, short evaluation cycles are preferable in order to quickly stabilize the requirements. This intensifies communication between the partners and promotes user participation (Riedl, 2019).

The *what* of the evaluation makes statements about which properties the prototype should have (especially with regard to functions, services and interfaces). The *what* of the evaluation strategy indicates which properties the final product should have and which properties are specific to the prototype (or to several versions of the prototype) and are therefore only preliminary. The *how* of the evaluation makes statements about which evaluation method should be used to arrive at a result that is accepted by both partners. In general, a procedure according to the model of utility analysis is appropriate, which is adapted to the object of evaluation and coordinated with the evaluation situation. This requires agreements on how different results of the evaluation are to be treated by the developers on the one hand and the users on the other hand. The goal is to achieve agreement (Riedl, 2019).

For the concrete research of this thesis, several disposable prototypes are developed by the researcher. Some parts are reusable for next iterations. Finally a reusable but incomplete prototype is evaluated against the research objectives, together with the interview partners (potential users). A thorough evaluation of the prototype with a high amount of users is not done, due to the time and effort constraints of this thesis.

Types of Prototyping

Types of prototyping means why prototypes are used and fundamentally how to proceed with their usage. A distinction is then made between exploratory, experimental and evolutionary prototyping.

Exploratory prototyping aims to determine the functional requirements of an information system by developing a prototype that allows for testing of different solution alternatives with users. The future users should be able to evaluate the prototype on the basis of real work situations. The focus is on functionality, ease of change, and short development time. The required functions are determined successively. The influence on the phase scheme is minimal as it's primarily used for requirements analysis.

Experimental prototyping aims to complete the specification of system components and prove the suitability of object specifications, architecture models, and solution ideas. Users are not usually involved in this process, and implementation work is integrated ("pulled forward") into analysis and design work, which changes the phase scheme more than exploratory prototyping. Sometimes the use of prototypes for evaluation by users is described as "experimenting with prototypes"; this is concluded as experimental prototyping (Riedl, 2019).

Finally, evolutionary prototyping is an incremental project work approach that involves developing a prototype for easily identifiable requirements, which is then used as the basis for the next planning step. The prototype and information system become indistinguishable as the prototype is successively improved and ultimately used as a productive information system. The change in the phase scheme is most apparent in evolutionary prototyping (Riedl, 2019).

For the concrete research of this thesis, a mix of all three prototyping types are applied, as described in the following section.

Prototyping approach

Prototyping results in an approach that modifies the phase scheme, but in no way substitutes it. Exploratory and experimental prototyping can be used "intermixed". A typical approach is the following: First, an explorative approach is taken, for which a disposable prototype is created (rapid prototyping, also referred to as quick and dirty). The primary goal is to minimize the time in which the first prototype is available. If the assessment shows that the essential requirements have been captured and taken into account, the prototype is only used for comparison purposes (e.g. to be able to check later whether the essential requirements have been updated). After that, an evolutionary approach is taken, for which a reusable prototype is developed. After each assessment, a decision is made as to what will better support the achievement of the planning goals: to modify the existing prototype or to discard it.

If an evolutionary approach is taken and the prototype is reused in any case, then the following work steps can be distinguished (Riedl, 2019):

1. rough specification of requirements
2. creating a prototype
3. using the prototype
4. evaluating the prototype
5. modify the prototype according to the results of the third step; run through the third and fourth steps n times

After the prototype has been run through (n+1) times in the third to fifth steps (prototyping cycle), the final product has been created. A prototyping cycle can serve a specific purpose according to plan (e.g. a first cycle of initiation). Here, the primary purpose is for the participants to familiarize themselves with the project task. In the first prototype, therefore, only a few functions already known to the developers are realized. A second prototyping cycle can serve as orientation. The aim here is to capture all the essential requirements that will be realized in the prototype. Finally, a third purpose is stabilization. This is primarily about refining and adding features and producing the required performance. Late orientation and early stabilization prototypes are close to the level of the final product; they can therefore also be used for user training. If planning and realization of an information system are seen as a layer model, then prototyping can be done either horizontally or vertically. In horizontal prototyping, individual layers of the system are constructed (e.g., the user interface or the functions); in general, horizontal prototyping refers to the construction of the user interface. In vertical prototyping, a selected part of the target system is implemented "in depth". This approach is appropriate when the functionality of the overall system is unknown and its realization possibilities are questionable (Riedl, 2019).

For the concrete research of this thesis, a mix of approaches is used. First, an experimental approach is taken by developing several disposable software prototypes. This is done in order to assess the essential technical requirements and get a feel for the feasibility of the possible features (research objectives). Then, exploratory prototyping is used, by evaluating the thus far developed prototype against the solution objectives together with the interview partners (potential users). Finally, an evolutionary approach is taken, for which a reusable prototype is developed. Parts of the software prototype may be fully reused for future iterations.

Effects of prototyping

Riedl (2019) notes that on the effects of prototyping there is limited empirical evidence available. Field reports suggest that the costs of prototyping can either increase or decrease, depending on the context, type of costs, and type of prototype used. However, some researchers argue that prototyping can reduce development costs by providing early error detection and motivation for users, among other benefits. The most significant effect of prototyping is seen in the improvement of the end product's functionality and usability, resulting from improved cooperation between users and developers. This can lead to better acceptance of the product by users, resulting in a decreased need for maintenance. Additionally, the evaluation of prototypes can support project controlling and help assess whether an IT project should be continued unchanged, rehabilitated with significant changes, or terminated altogether (Riedl, 2019).

4.4 Summary

In this preceding chapter, the scientific research methodology was described. It started with an outline of the motivation and objectives, which gave an overview of the chapter and highlighted its purpose. The general approach was described in detail, explaining what each activity of the design science research methodology (DSRM) consists of. While activities 1 and 2 are conducted with the help of interviews with practicing professionals, the activities 3, 4 and 5 are primarily carried out by applying the prototyping researching method. Furthermore, all the used scientific methods and their application within the thesis were described.

5 Interviews

This chapter discusses the conducted interviews with the working professionals. The interviews within this research have the main purpose of helping with identifying the problem of promotion of releases in GitOps environments, because there is insufficient prior written literature on this topic. Working professionals in the GitOps field are interviewed on the topic. Their personal opinion and practical experience, use cases and concrete problems are brought up for discussion. The importantance of a solution to the problem is highlighted and motivated. With the problem identification in place, several distinct solution objectives are defined from the basis of the discussions from the interviews. These solution objectives are then further elaborated in the following chapter 6, where they primarily serve as the basis for the desired functionality of the developed prototype. In addition, related ideas and approaches that were discussed by the interview partners are presented. The interview transcripts are found in the Appendix B.

5.1 Problem Identification & Motivation

As a first step in the research process, the overarching problem with promoting releases in GitOps environments is elaborated in detail. To help with this process, a series of semi-structured interviews are conducted with practicing professionals in the GitOps field. Furthermore, several distinct problem items are defined, for further use in the next research activity 5.2 Definition of Solution Objectives. The importance of a practical solution to the problem is highlighted.

5.1.1 Problem 1: Promotion is limited to container image

The currently available GitOps toolchains - from e.g. Flux or Argo Projects - provide a way to patch the newest container image version into the desired state defined in Git. This is offered as a plain commit without human interaction, and is mostly used, in order to have a desired state in Git updated with the currently latest version of a container image, that is available in a specific container image registry.

When promoting new releases of applications or infrastructure resources defined in Git as the desired state, it is not only desirable to promote new version tags, but it is also needed to promote arbitrary resources like any files defined in the Git repository, or Kubernetes resources, or even other external resources.

Furthermore, it is desirable to have a human approve the changes, which are done by a machine, e.g. with the help of Git pull requests. The practical experience of interview partner 1 points to the idea that very few companies really do Continuous Deployment, most of them want to have some kind of manual release.

As interview partner 3 mentions, a release can be any combination of a general change to a system. It should not be distinguished between a release, a hotfix, a minor or a major change. Releases can be comprised of many changes that are new to a system, in the Kubernetes context, this could be a new container image, a change in a ConfigMap, or any other change in the desired state of the system. A release may be a change to the application and or to the application specific infrastructure. For promotion there are two important components, the environment infrastructure, and the application itself (Yuen et al., 2021).

While a container image is a specific type of information inside a file in a desired state definition, the ConfigMap stands for a generic resource, typically a separate file inside the GitOps repository. With this statement, the interview partner means, that a release can include any type of information or resource. When following the principles of GitOps this is usually constrained to a plain text file in the GitOps repository, which then may be promoted by copying the file contents from one to another GitOps environment definition.

A solution to this problem could enable users to promote any arbitrary information like a file or folder in the Git repository, or another data from another source (e.g. artifact repository). This is especially valuable, when thinking broadly, that the tool which implements this problem solution, could also be used as a general GitOps tool for interfacing with desired state definitions and moving certain resources, or patching and updating them.

5.1.2 Problem 2: Order of promotion to multiple environments

Since currently there is no general standardized tool for promotion to multiple environments with the GitOps approach, a specific order of promotion through environments can not easily be achieved. For example, if an environment should only be promoted, if the specific release has successfully passed another environment.

The desired specified order of environments could be like interview partner 3 mentioned, where the core banking system had up to twelve environments, which a new application release needed to pass for testing, in a specific order. From the practical experience of interview partner 3, the environment setup for the critical core banking system consisted of many environments, where the production environment with the end users, was only a very small part.

Due to the high criticality in the banking system, it is the highest priority for an application to run in a stable manner throughout its lifetime, and for new version releases and new features to not break anything. The banking system would have a lot of environments/stages just for testing, in order to ensure the very high quality of the software. The criterion for an environment/stage in the context of the banking system would be the quality level of the software. Depending on if the software still had a lot of bugs, or it is closer to acceptance, a new release would be in a certain environment/stage.

Tenant/Customer	A	B	C	D
Stage 1	v1 –> v2	v1	v1	v1
Stage 2	v2	v1 –> v2	v1 –> v2	v1
Stage 3	v2	v2	v2	v1 –> v2
Stage 4	v2	v2	v2	v2

Table 5.1: Rollout to environments in stages

Additionally to the order of promotion through environments, there is the problem of rolling out new releases to all environments at once. When multi-tenancy is not implemented within the application, it can be desirable to not roll out to all environments at the same time. For example, as a platform provider with tenants represented as separate GitOps environments, it can be desirable to split up the rollout of the new release into stages. A new version or release could firstly be rolled out for the unimportant customer, and after quality checks have passed, the release could be promoted also to the important customer. A solution to this problem formulation is especially valuable for use cases with a high amount of regulations like e.g. financial institutions. The rollout to environments in stages is presented in table 5.1.

5.1.3 Problem 3: Dependencies can not be defined

Before promoting a new release to another environment, it may be desirable to have specific quality gates, which define the quality standards the software must meet, in order to be considered for a promotion. These can be integration, acceptance and end-to-end tests for example. Additionally with a monitoring and observability tool, certain metrics can be evaluated with the new release, and if they do not meet a certain value, the release would not be a candidate for promotion. As a minimum, it must be ensured that the new version is deploying correctly. When only given the desired state definiton in Git, it can not be measured, if the system will end up in a running and successful deployment.

Applications may have specific dependencies to other applications or services. To provide an example, in the context of the Kubernetes platform, an application may need an internal service or any other external dependency like a managed database or another infrastructure service. When thinking of release promotion, a dependency might be a Kubernetes object like a job or another custom resource, which - when it is observed with a ready or successful status - can be seen as a test result or in prinicple a quality gate, which qualifies the new application release for promotion.

A solution to this problem statement is especially valuable when having the whole continuous delivery lifecycle in mind. After deployment of a new application release to a certain environment, it should be evaluated for its quality. So, tests and other metrics should be evaluated, in order to ensure high software quality at every stage in the development and also deployment cycle, all this in a fully automated way.

5.1.4 Problem 4: Provider and tool dependency

With the currently available GitOps toolchains, and due to the lack of best practices and standardized solutions, users or platform engineers are inclined to build and setup advanced and complex pipelines that are interdependent on a lot of things, as interview partner 1 highlights. This usually results in tight coupling of the Git provider, the GitOps engine, the CI workflow/pipeline platform, configuration management tools, and other components. Since many tools in the GitOps ecosystem are not very mature in their development and adoption, it is of use that components are loosely coupled and can be exchanged with alternatives in the future. Additionally there need to be permissions and policies granted to many different people and machines on many different systems. In the end, it results in vendor lock-in and tightly coupled toolchains.

A solution to this problem enables users to avoid vendor lock-in as much as possible. This fits well with the Kubernetes ecosystem, which is currently the most prominent cloud native software platform, in that it also tries to be cloud and platform agnostic at every step of the way. With Kubernetes it is always desirable that applications running on the platform can run on any type of infrastructure.

5.2 Definition of Solution Objectives

Now that the problem has been identified in the earlier section 5.1 Problem Identification & Motivation, research objectives of a solution to the problem will be inferred. Each objective provides a solution to a distinct problem item (fig. 5.1).

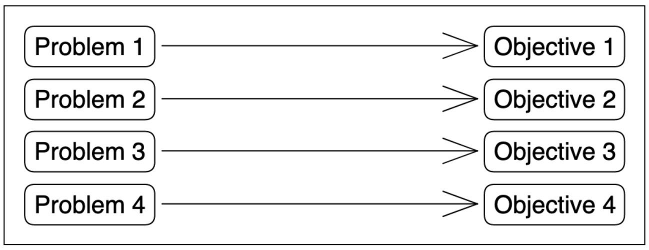


Figure 5.1: Definition of Solution Objectives by inferring from Problem Definitions.

A solution objective is a qualitative description of how the functionality of the developed prototype is expected to support a solution to the problem definition. For later evaluation of the results, the observed functionality can be backtracked to a certain solution objective and its respective problem item.

5.2.1 Objective 1: Arbitrary resources can be promoted

From the problem definition *Problem 1: Promotion is limited to container image*, a solution objective *Objective 1: Arbitrary resources can be promoted*, is inferred. A qualitative description of the solution objective is pointed out in the following.

A promotion subject, which is promoted between GitOps environments can potentially be of many types. A popular type of data, which can be promoted, is the version tag of the container image of a particular application. For some use cases it is not sufficient to promote only the version of the container image. In order to provide a solution to this problem, the solution must provide a way to promote arbitrary types of resources. In the GitOps context, resources are typically constrained to declarative representations in plain text format, which are defined in a Git repository. By providing a way to copy user defined files or directories in the form of filesystem paths inside a Git repository, potentially any type of resource may be a possible subject for promotion. When GitOps environments are stored in multiple, separate Git repositories, there needs to be the functionality to copy between these repositories. This is illustrated in figure 5.2.

Furthermore, for these arbitrary files or directories in GitOps repositories a descriptive name for a certain file or directory should be defined alongside the respective copy operation of the promotion subject. This is useful for identifying the specific promotion subject on a higher abstract level. This way, for example, an application's environment variables, which are defined and actually disguised as a Kustomize overlay in a Kubernetes deployment resource, can be given a friendly

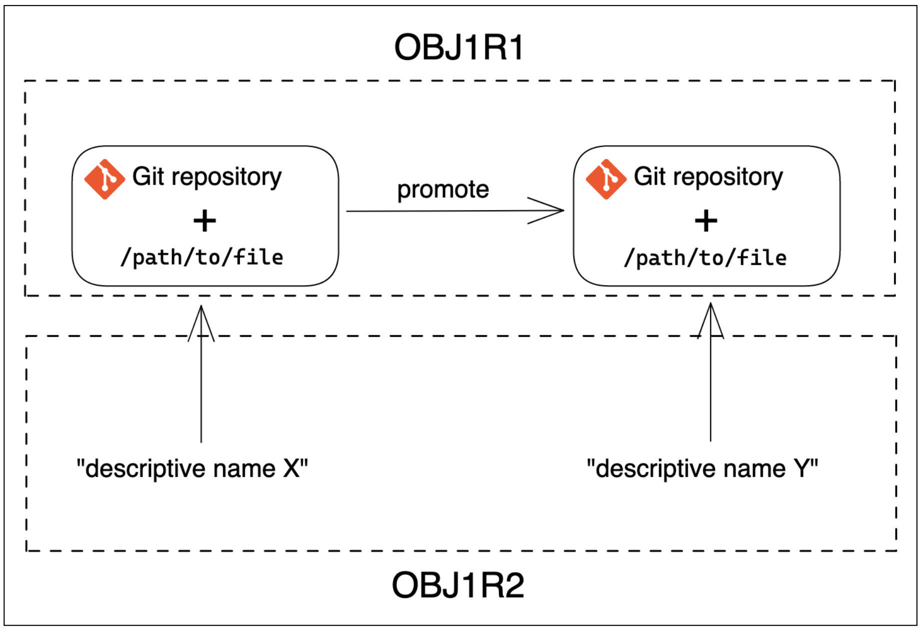


Figure 5.2: Objective 1: Arbitrary resources can be promoted.

name like "Environment variables/Application Configuration". This makes it a lot easier to identify a particular promotion subject, without needing to manually inspect the raw contents of a file or even multiple files or directory trees. This is illustrated in figure 5.2.

A collection of the requirements for the solution objective is shown below Each requirement is represented in the form of a user story (Cohn, 2004), and is labelled with a code, from a combination of the objective's code (e.g. OBJ1) and the requirement's code (e.g. R1).

- OBJ1R1: As a user, I can define any filesystem path inside a Git repository, with a respective target path in a Git repository, as a promotion subject.
- OBJ1R2: As a user, I can define a descriptive name for a promotion subject, which is represented as an arbitrary filesystem path.

These requirements are formulated from the perspective of the user of the developed prototype. Not exclusively the user can evaluate whether the requirement is fulfilled. An observer, like the researcher, can evaluate the fulfillment of a solution objective on the basis of the demonstration (section 6.2).

5.2.2 Objective 2: Strict flow of promotion through environments

From the problem definition *Problem 2: Order of promotion to multiple environments*, a solution objective *Objective 2: Strict flow of promotion through environments*, is inferred. A qualitative description of the solution objective is pointed out in the following.

When promoting a new application release to multiple environments, it may be necessary to define a certain order, in which the promotion proceeds through the environments (fig. 5.3). This can have many reasons, as defined in the problem identification section 5.1.2 earlier. A release may be rolled out to the initial development environment continuously, without any quality gates or other checks, to ensure software code and runtime quality. However, in order for a new version release to proceed to environments like performance test environments, which can produce high resource costs for each test, it may be useful to control the deployment to certain environments and this way, further constrain the deployment to subsequent environments, with a step in between. Another use case for this solution objective is for service providers with a multi-tenant architecture, which is implemented with GitOps environments. These service or platform providers may want to rollout a new release with a certain staging process.

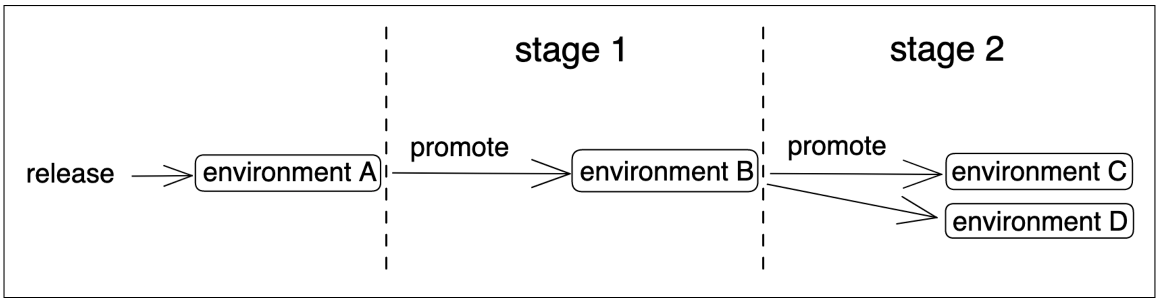


Figure 5.3: Objective 2: Strict flow of promotion through environments.

A collection of the requirements for the solution objective is shown below. Each requirement is represented in the form of a user story, and is labelled with a code, from a combination of the objective's code (e.g. OBJ2) and the requirement's code (e.g. R1).

- OBJ2R1: As a user, I can promote releases through multiple environments in a certain user-defined order.

The initial release to the first environment (environment A in figure 5.3) is not handled by the developed operator prototype. However, the subsequent stages (stages 1 and 2 in figure 5.3) are automated by the promotions operator.

5.2.3 Objective 3: Dependencies of a promotion

From the problem definition *Problem 3: Dependencies can not be defined*, a solution objective *Objective 3: Dependencies of a promotion*, is inferred. A qualitative description of the solution objective is pointed out in the following.

After deploying a new release to a certain environment, the minimum requirement for further proceeding to a subsequent deployment environment, is typically to check if the application was deployed successfully and is running in a healthy state. While this is the minimum that should always be ensured before promotion, other types of processes may be done, in order to reduce the likelihood of delivering bad quality software, which could likely be introduced by a bad release. One of these can be to have dependencies for a promotion. In the context of Kubernetes, this could be another Kubernetes workload, or any other object or custom resource. Kubernetes native testing tools or workflow engines may provide custom resources with Kubernetes API conformant conditions, in order for other programs to check the state of the resource, in particular this may be the ready condition. This solution objective is illustrated in figure 5.4.

A collection of the requirements for the solution objective is shown below Each requirement is represented in the form of a user story, and is labelled with a code,

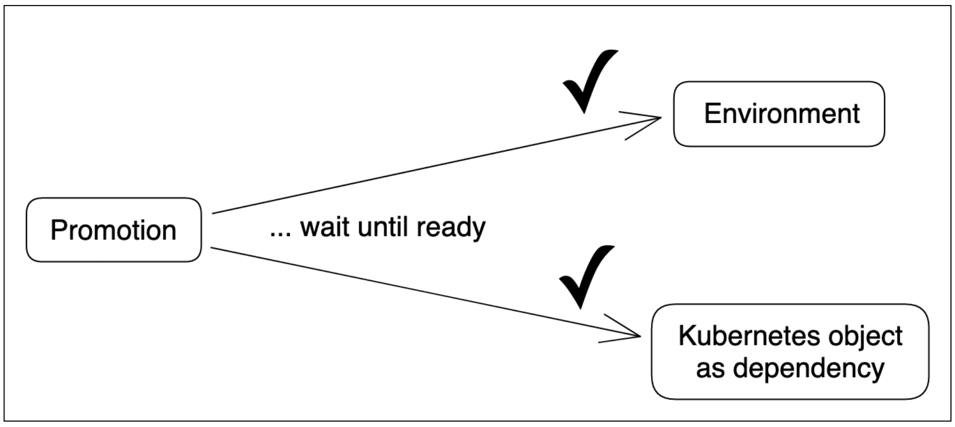


Figure 5.4: Objective 3: Dependencies of a promotion.

from a combination of the objective's code (e.g. OBJ3) and the requirement's code (e.g. R1).

- OBJ3R1: As a user, I can define Kubernetes objects as dependencies for a promotion.

A promotion process can have predefined dependencies, which must be ready or successful before a promotion is triggered. The prototype design sees a Kubernetes object as a dependency. This can be a certain workload like a deployment, a test run, a database, or any other object/resource. The environment (in fig. 5.4) is the aggregation of all resources that are defined in the respective desired state in the GitOps repository.

5.2.4 Objective 4: Vendor-neutral, tool-agnostic

From the problem definition *Problem 4: Provider and tool dependency*, a solution objective *Objective 4: Vendor-neutral, tool-agnostic*, is inferred. A qualitative description of the solution objective is pointed out in the following.

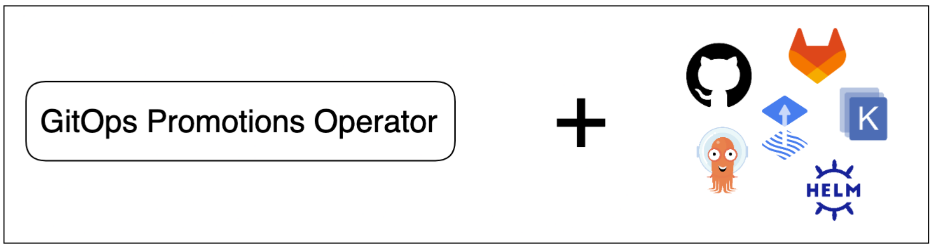


Figure 5.5: Objective 4: Vendor-neutral, tool-agnostic.

When following current good practices and guidelines to build a promotion setup for GitOps environments, users are inclined to build workflows which are constrained to specific Git providers, GitOps engines, CI pipeline and configuration tools. This leads to tightly coupled setups, and vendor lock-in. With the strong open-source foundation of the Kubernetes platform and ecosystem, it is a good practice to build additional tools and platforms on top of Kubernetes, in order to remain cloud and vendor agnostic. In the space of GitOps promotions, there is currently insufficient tooling for this purpose. Therefore, one solution objective of this research (fig. 5.5) is to provide a generic tool, which is vendor-neutral, and agnostic to the Git provider, as well as the configuration/templating tool.

A collection of the requirements for the solution objective is shown below Each requirement is represented in the form of a user story, and is labelled with a code, from a combination of the objective's code (e.g. OBJ4) and the requirement's code (e.g. R1).

- OBJ4R1: As a user, I can use any GitOps engine, Git provider and configuration/templating tool.

The developed prototype operator is compatible with all GitOps engines like Flux or ArgoCD, configuration tools like Helm or Kustomize, Git providers like GitHub or GitLab. The primary purpose of this vendor-neutrality trait is to have a wider pool of potential users who can give early feedback. An integration to certain tools can be built in the future, in order to enhance the user experience and ease of use.

5.3 Insights into Related Ideas and Approaches

The conducted semi-structured interviews gave valuable insights into the unique views on the problem and related topics of each of the working professionals. Some of the related ideas and alternative approaches are discussed in this section.

5.3.1 Rolling Production Environments

Interview partner 1 mentioned a new idea of rolling production environments. With this approach, on major changes a new production environment would be created, and then progressive delivery would be done against the whole environment. This includes the application and the whole infrastructure layer below it. In the Kubernetes context this would be the cluster itself, not only the deployment resource. Since GitOps allows to easily recreate the whole environment infrastructure, this is not difficult to achieve. With tools like the GitOps Terraform Controller [1] such a setup is easier to achieve than before, without the GitOps approach.

Interview partner 1 discussed this idea as an alternative to having different long-living environments, what would have been done in the past. Instead, the idea is to re-create entire copies of the production environment, with the power of GitOps, and do progressive delivery for that copy of the environment as a whole. Not only the application - the running container - but also the platform and infrastructure below can be immutable and versioned. This could be done, in order to further limit the impact of a bad change in a new release of the software version. In addition, when having not just the desired state of the application, but also the entire infrastructure below it, stored in Git, it also increases the immutability and therefore resiliency of the user-facing service.

In the Kubernetes ecosystem, there have emerged a lot of applications, which are providing certain services like the Cert-Manager and TLS certificates, or services which are providing policy functionality. These applications responsible for infrastructure or for supporting the primary application, which is in the end

[1]https://github.com/weaveworks/tf-controller

user-facing, all have their own version and are constantly updated. There is also the possibility that a new release of such a supporting service can break the primary user-facing application. Kubernetes deployments might have many different applications, which are providing supporting services for the actual custom application. All these might be critical dependencies. Every dependent service can change version, Kubernetes itself is upgraded, and some APIs might even be deprecated or removed. There are many variables that could cause an outage of the particular critical user-facing service. Usually there would be a maintenance window for major new version releases, where responsible people are ready, incase anything breaks during a new release. The chance of failures could be decreased with this new approach, where the entire production environment is replaced with all new components with new versions. Interview partner 1 sees this idea of rolling production environments as one of the next steps to move towards with the GitOps approach.

5.3.2 Overview of GitOps Repositories

Interview partner 2 discussed the idea, that currently when using common tooling, it can be quite cumbersome to get a grip of what is inside a GitOps repository/environment, just by looking at the filesystem tree and plain text files. GitOps tools like Argo can visualize the other way around, meaning the ArgoCD dashboard can visualize what is deployed by ArgoCD itself in a specific deployment environment, i.e. cluster or namespace. However, when a human looks at the plain GitOps repository, it can be difficult to understand the setup. So the advantage of GitOps having a single source of truth, where the human operator or developer can look, and understand at one glance what the truth is, is kind of lacking at the moment with the current ecosystem of GitOps tools. The good overview that GitOps is supposed to bring in theory - when thinking of the aspect of the desired state being the single source of truth, with declarative configuration that is easy to read - seems to be a bit insufficient at the moment with the currently available tools.

What goes hand in hand with this topic of the overview over GitOps repositories is the overview of versions that are deployed, and where the versions are deployed. With a GitOps setup, there are a couple of versions, or revisions, for different sources. There is often a version of the GitOps repository, the Helm or Kustomize reference or many of those, and the different container image versions. With all these different versions at different places, it can be confusing. A better overview over the deployed versions of a GitOps repository would be good to have. It might not be desirable to know the revision of the GitOps repository, where the desired state is stored, but better to know the version of the Helm or Kustomize configuration that is stored in the repository.

5.4 Summary

This chapter discussed the problem identification and motivation of the main topic of this thesis, namely the promotion of releases in GitOps environments. With the help of practicing professionals, who are working in the GitOps field, several problems have been identified, and defined along with research objectives which should provide a possible solution.

Problem 1: Promotion is limited to container image, relates to a frequent issue with currently available tooling in the GitOps ecosystem. Often times solely container image version tags are the focus with current tools when promoting new versions or releases. It was discussed, that this is insufficient for some use cases. It is sometimes required to handle all sorts of resources, not just the version tag of a container image. Especially when not using containerization technologies for runtime, this is an important problem to handle. In order to be able to provide a solution to this problem with a comprehensible approach, a solution objective was inferred and its requirements defined.

Objective 1: Arbitrary resources can be promoted, defines a qualitative description of how the respective problem is supposed to be solved by the developed artifact. The main idea is to offer the capability to promote arbitrary resources, meaning any type of resource, instead of solely the container image version. The technical implementation in the proposed prototype foresees the functionality for promoting a list of filesystem paths inside the Git repository of the desired state to other environments. In addition these arbitrary resources should be able to be assigned a descriptive name, in order to identify the promotion subjects more easily.

Problem 2: Order of promotion to multiple environments, states the fact, that it is not a straight-forward process of how the order of promotion through multiple GitOps environments can be setup. When adhering to the principles of GitOps and the asynchronous deployment process (described in chapter 3) there is no streamlined approach or tooling, that automates this.

Objective 2: Strict flow of promotion through environments, defines the requirements for the proposed prototype, in regards to the according problem of having a certain order of promotion through environments or stages. The objective describes the capability for defining a certain order of environments, in which releases traverse through. In addition, this solution objective opens up the possibility to setup promotion in stages, in which certain environments must be deployed to first, before the release can deploy to other specified environments.

Problem 3: Dependencies can not be defined, relates to the problem that when wanting to promote a new release from one environment to another environment, it is not easily achievable with the available tools to specify certain dependencies, like other workloads or microservices in the same or another environment, or altogether dependencies from external sources. This is especially desirable for evaluating test results or other metrics, before triggering the promotion.

Objective 3: Dependencies of a promotion, describes how the respective problem of being able to specify dependencies for a promotion, could be solved in the proposed prototype. While the minimum dependency is the successful deployment of the workload of a release, it may also be desirable to specify other resources or workloads which need to be in a certain state, before triggering a promotion.

Problem 4: Provider and tool dependency, draws attention to the common problem of being dependent on single tools and providers. The more complex the Continuous Delivery is setup for a particular project, the more difficult it is to de-couple or switch providers for certain components. Furthermore, since many tools in the GitOps ecosystem are not very mature in their development and adoption, it is of use that components are loosely coupled and can be exchanged with alternatives in the future.

Objective 4: Vendor-neutral, tool-agnostic, defines the requirements of how a vendor-neutral and tool-agnostic prototype can be implemented. The main components which are desirable to support all alternatives, for being able to switch between them, are the Git providers (e.g. GitHub, GitLab), the GitOps engines (e.g. Argo, Flux), the configuration/templating tools (e.g. Kustomize, Helm).

Additionally, related ideas and approaches were discussed by the interview partners. These points were not directly considered for the conducted design science in the prototype, however they are discussed later in chapter 8.

6 Prototype

This chapter describes the developed prototype, called the *GitOps Promotions Operator*. First, the asynchronous nature of a typical GitOps deployment is described, and where the proposed *GitOps Promotions Operator* prototype finds its place within this architecture. Next, abstract models and their respective custom resources are designed which describe the *Environment* and *Promotion* custom resources; which are then afterwards implemented as mockups of Kubernetes custom resources in the declarative Yaml specification syntax. Next to the mockups implemented in the prototype, additional alternative mockups are suggested for possible alternative implementations. Then, the implementation of the custom resource definitions and respective controller logic is described within the context of the used Kubernetes operator framework Kubebuilder. Once the design and development of the artifact - the *GitOps Promotions Operator* prototype - is successfully done, the operator is demonstrated in a proof of concept, and finally evaluated against the solution objectives from section 5.2.

6.1 Design & Development

The design and development of the prototype is based on the research objectives defined in section 5.2. The overarching goal is to fulfill the objectives with the developed artifact, the *GitOps Promotions Operator* prototype. This is shown by demonstrating the functionality and evaluating against the research objectives.

6.1.1 Asynchronous GitOps Deployments

First, the process of a typical GitOps deployment needs to be addressed, where the proposed *GitOps Promotions Operator* prototype needs to find its place in this asynchronous process.

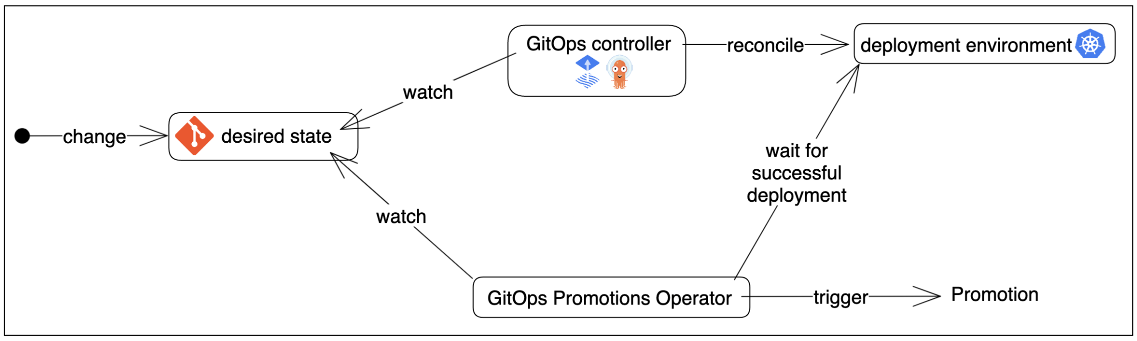


Figure 6.1: Asynchronous GitOps deployment and promotion.

It begins with a change of the declarative state definitions in a Git repository, which is then being reconciled by the GitOps controller. The timeframe from the change of the desired state to the actual state being applied is variable. An external system (i.e. the *GitOps Promotions Operator*), which watches the desired state, has no knowledge of the current state of the deployment environment - when solely given the desired state in the Git repository. An external system also does not know whether the desired state works or not (e.g. when the new version fails to rollout).

The proposed *GitOps Promotions Operator* ideally needs to pick up the asynchronous deployment process after the new release of the desired state has been successfully rolled out to the deployment environment. In order to achieve this, it needs to at least check the availability of the deployed application or workload, before continuing on with the promotion. The described asynchronous GitOps deployment and promotion model can be seen in figure 6.1.

In order for the proposed *GitOps Promotions Operator* prototype to fit into this asynchronous deployment architecture, it needs to have several asynchronous phases in its controller logic. The minimum steps or phases it needs are the following:

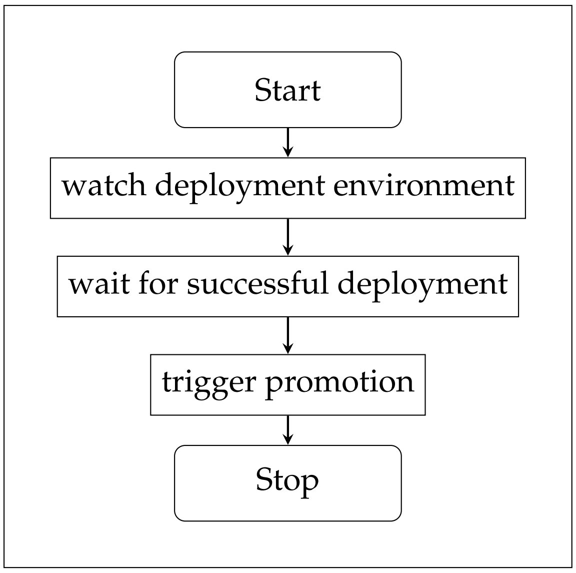
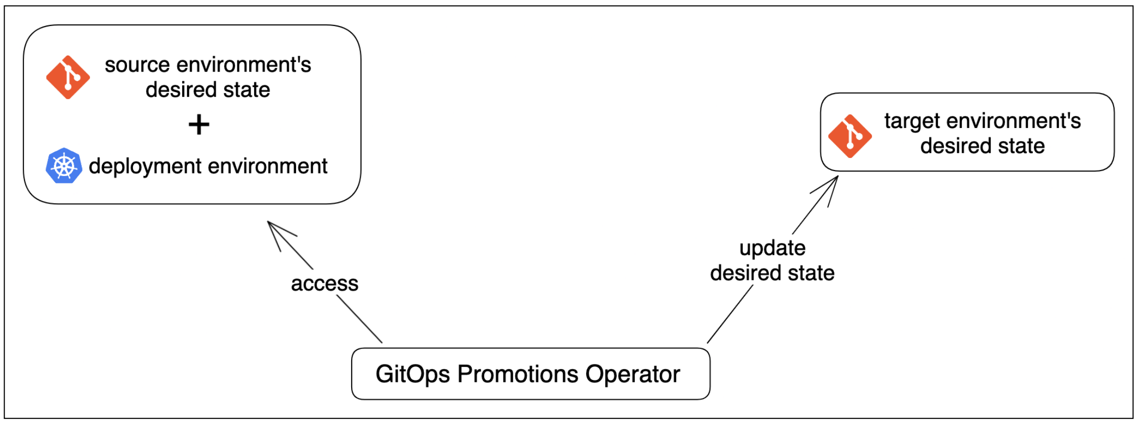


Figure 6.2: Asynchronous Phases of Deployment.

In order to check if a deployment was successful, the *GitOps Promotions Operator* needs access to the deployment environment. This is already given, if the operator is running inside the same deployment environment. For the promotion process, the operator also needs access to the source and the target environment (Git repository). The described architecture can be viewed in figure 6.3.



source environment's desired state
+
deployment environment
target environment's desired state
access
update desired state
GitOps Promotions Operator

Figure 6.3: GitOps Promotions Operator Promotion from source to target environment.

Once the architecture of the asynchronous GitOps deployment process, and where the *GitOps Promotions Operator* fits within this process is clear, the abstract models for the custom resources and their functionality can be designed.

6.1.2 Abstract Models

In this section, abstract models for environments, as well as promotions are designed. The term abstract model stands for a qualitative description of a resource. The abstract model is supposed to function as a high level abstract representation of a resource, that exists in the actual world, either internal or external to the system, or is used purely as an abstraction layer for such an actual resource.

Environment Model

In the context of this software prototype, an environment is a GitOps environment as defined in section 3.2. As a reminder, this is a folder/directory in a Git repository, which essentially points to a deployment environment, typically being a Kubernetes cluster/namespace. Therefore, the model for the environment primarily represents a directory path in a certain Git repository. This can potentially be a specific branch or any commit or refspec [1] in the Git repository.

In addition to the Git repository and the path pointing to the GitOps environment, the abstract model has a definition of dependent resources inside the deployment environment. This is needed in order for the operator to know which resources are deployed and belong to the desired state definition; and furthermore to achieve the possibility to define dependencies, which need to be fulfilled before a promotion is allowed to be triggered. A dependency can be potentially anything, which needs to be in a certain state, before a promotion is triggered. Abstract examples of dependencies could be e.g. HTTP(S) endpoints internal or external to the deployment environment which return a certain response; Kubernetes objects which need to have a certain condition or be in a certain state.

As an alternative to a Git repository, the desired state of the GitOps environment could also be stored in a object store elsewhere like an object or blob store or an artifact registry; or it could be stored in an alternative version control system, other than Git. The desired state could also be a combination of these possible sources. The described other possiblities for the desired state store are not further handled in the proposed prototype, however they must be noted and considered in the base architecture, in order for the possibility to support them in the future.

Promotion Model

In the context of this software prototype, a promotion is a GitOps promotion as defined in section 3.2. As a reminder, this is the process of promoting a new application or infrastructure version to another deployment environment, which typically means a change in the desired state definition in a GitOps environment.

For the proposed and implemented prototype within this concrete research, the process of a GitOps promotion is the change of the desired state definition of a certain GitOps environment, typically mentioned as the target environment of a promotion. For the prototype, the declarative state, which is promoted, i.e. changed in the target environment, is drawn from the source environment's desired state definition. In the easiest case, this means that a certain file or directory is copied from the source to the target environment - or a list of files/directories.

[1] https://git-scm.com/book/en/v2/Git-Internals-The-Refspec

These files or directories which are promoted, are called promotion subjects within the context of this thesis.

Promotion subjects can potentially be of many other types, however other types are not implemented in the prototype. Examples for other promotion subjects could be other representations of resources in the desired state, like object references to Kubernetes built-in resources or custom resources instead of plain text files or directories. Custom resources could be ArgoCD Applications or other high-level abstractions or collections of resources. As alternative sources for promotion subjects, the operator could gather other arbitrary information. For example this could be information like a version tag of an artifact repository. This arbitrary data point could then be patched statically in a certain YAML path of a file. Alternatively, it could be patched into lines, which have been marked as comments in files in advance. The mentioned strategies are used by other software programs, which interface with plain text desired state definitions.

What should also be mentioned, but is not implemented in the proposed prototype, is that a promotion process could be of other types. As an example, this could be the difference between two commits of Git repositories, which is essentially a patch between two states. This difference could be applied to another environment.

The promotion process could possibly have hooks or phases before and afterwards, which could be configurable by the user. Furthermore, such a promotions operator could have the functionality to combine multiple promotions together. This way, the end-to-end observability and manageability would be improved. This is in line with a problem discussed earlier, of the overview over GitOps environments and the deployed applications. On top of that, a functionality could be developed to increase observability into e.g. the current state of a specific release and which environments it passed.

Another alternative possibility of a promotion process or how a promotion could be achieved, is that Git branches are leveraged. This could mean, that an environment would differ from another environment just by the branch of the same Git repository.

Changing ecosystem

What also needs to be raised for discussion is that the understanding of what a GitOps environment and promotion is, could change in the future, depending on how the GitOps ecosystem changes and in which direction it is going. Currently the GitOps space is strongly centered around Kubernetes, at least for the controller or engine which does the reconciliation between the desired and the actual state.

Now that the abstract models are designed, they need to be implemented. Since the decision for the prototype is to be developed as an extension to the declarative Kubernetes API, and the resources to be implemented as Kubernetes custom resources, the design of the custom resources is described in the next section.

6.1.3 Design of Custom Resources

In order to be able to represent environments and promotions, the requirement is to at least start with two custom resources for each the environment and the promotion.

Environment Resource

The *Environment* represents a GitOps environment, which is a Git repository and path. The Git repository can be a clone URL to the repository, and the path is the relative filesystem path, which points to the environment, inside the repository.

The custom resource for a GitOps environment needs at least the following properties:

- URL of the source Git repository
- Path pointing to the environment inside the repository
- Dependent resources inside the deployment environment

The URL has the format of a HTTP(S) or SSH URL, which links to the Git repository, e.g. `http://localhost:8080/org/repo`, `https://gitprovider.com/org/repo`, `ssh://git@gitprovider.com:org/repo`.

The path has the format of a typical unix style filesystem path. It starts relative from the root of the given Git repository, and points to the directory, which represents the GitOps environment. Examples for a path are the following: `path/to/env`, `/path/to/env`, `./path/to/env`, `./path/to/env/`. Note, that these example paths all represent the same directory, they are just alternative notations.

The dependent resources are the resources (objects) in the deployment environment, which need to be successfully deployed, in order for the promotion to trigger. Examples of such resources can be Kubernetes workloads like deployments, or custom resources like ArgoCD Applications or Flux Kustomizations and Helm Releases.

Promotion Resource

The custom resource for a GitOps promotion needs at least the following properties (explained in detail in the following paragraphs):

- source environment
- target environment
- promotion subjects
- promotion strategy

The source environment defines the environment resource, where a promotion subject is promoted from. The target environment defines the environment resource, where a promotion should promote to.

A promotion subject can be potentially many different things. In the case of this prototype, a promotion subject is a file or directory, which is copied from the source to the target environment. Examples of such files or directories are the following:

`kustomization.yaml`, `./component/cert-manager/kustomization.yaml`, `./helm-values-prod.yaml`. Note, that the relative paths of the promotion subjects, are relative to the paths of the environment, as defined earlier.

As an example - but outside the scope of the implemented prototype - a promotion subject could also be fetched from another source, like an artifact registry, or be any other type of data and updated/promoted in the target environment, e.g. a version tag, helm values, etc.

With the proposed and implemented prototype, the promotion strategy is to raise a pull request at the Git provider (fig. 6.4), with the changes proposed by the promotion. A human can then review the changes and optionally approve and merge the pull request. After merging, the promotion will have taken place.

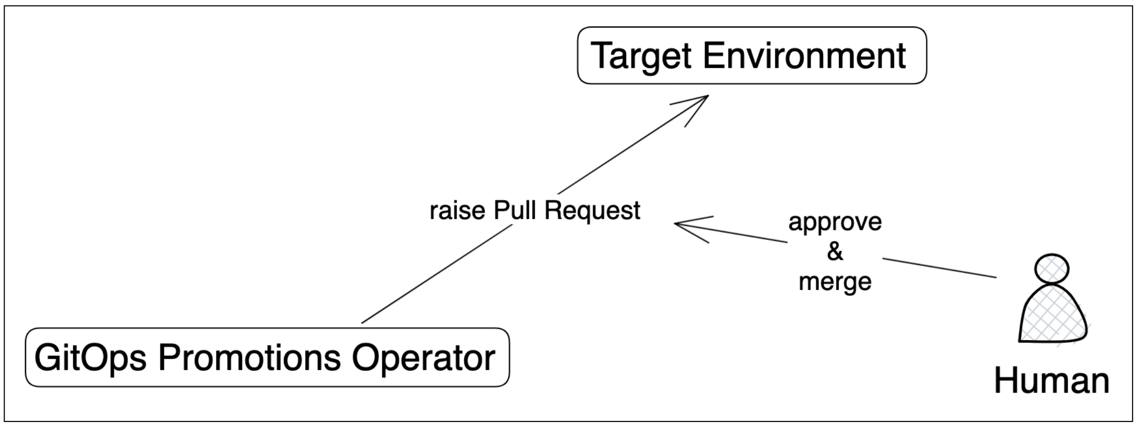


Figure 6.4: Pull Request at target environment.

Alternatively to a pull request, the changes could be directly commited and pushed to the target environment, without human interaction. This strategy should require different means of automated or otherwise external or additional checks, in order to ensure a safe promotion. This strategy is not implemented in the developed prototype.

Now that the custom resources are designed, they need to be implemented. Since the decision for the prototype is to be developed with the Kubebuilder framework and follow its style, mockups of the custom resources in Yaml format will be created as a next step.

6.1.4 Mockups of Custom Resources

When the custom resources have been specified, they can be actually implemented with the framework as Kubernetes custom resource definitions.

Users will mainly be dealing with the custom resources in a Yaml format. Yaml keys should be intuitive and make sense to the user. It also helps if they follow the core Kubernetes definitions regarding naming conventions. An example for a naming convention is the "Ref" suffix for Yaml keys. This suffix is typically appended to keys which represent a reference to another Kubernetes object. For example, "secretRef" says that this field refers to a Kubernetes secret resource. A possible mockup for an *Environment* resource could look like the following.

```yaml
apiVersion: promotions.gitopsprom.io/v1alpha1
kind: Environment
metadata:
  name: my-env
spec:
  path: ./path/to/env
  source:
    url: https://gitprovider.com/org/repo
    ref:
      branch: main
  dependentObjects:
    workloadRef:
    - kind: Deployment
      name: my-deployment
```

In this mockup the `.spec.dependentObjects.workloadRef` represents a list of Kubernetes objects in the cluster, which need to be successfully deployed, before a promotion is triggered. Additionally the git reference branch main is also specified in the `.spec.source.ref.branch` field.

A possible mockup for a GitOps promotion could look like the following.

```yaml
apiVersion: promotions.gitopsprom.io/v1alpha1
kind: Promotion
metadata:
  name: my-promotion
spec:
  sourceEnvironmentRef:
    name: my-source-env
  targetEnvironmentRef:
    name: my-target-env
  copy:
  - name: "Application Version"
    source: app-version
    target: app-version
  - name: "Kustomization File"
    source: ./app-version/kustomization.yaml
    target: ./app-version/
  strategy: pull-request
```

In the promotion mockup definition, there are four fields in the `.spec`. These represent the minimum properties of the previously defined abstract definition. It is to note, that the `.spec.copy` field represents the promotion subjects. It is a list of items, where each item contains a `name`, `source` and `target`. The `name` defines a custom name. The `source` and `target` fields together define a file copy operation, where the `source` is the relative path from the source environment, and the `target` is a relative path from the target environment.

6.1.5 Alternative Mockups

The following alternative mockups, for the custom resource definitions, i.e. the design of the declarative API, are suggested, but not implemented in the current prototype.

Alternatively, the promotion subjects could also be specified in the environment resource like the following:

```yaml
spec:
  promotionSubjects:
    copy:
    - name: "Application Version"
      path: app-version
    - name: "Kustomization File"
      path: ./app-version/kustomization.yaml
```

In the promotion definition, it would then suffice to specify a list of promotion subjects.

```yaml
spec:
  promotionSubjects:
  - "Application Version"
  - "Kustomization File"
```

This alternative design allows that each environment could have a unique path of a specific promotion subject defined. Now if a promotion is spanning over multiple environments, they could each specify their own unique path to a promotion subject. The promotion subject is declared in the promotion, but the actual path is defined per each environment.

6.1.6 Translation to Go types

Once the mockups of the custom resources in Yaml format are done, the declarative structure can be translated to custom Go types. The full source code of the Go types are found in Appendix C. The type EnvironmentSpec represents the `.spec` Yaml field. What also needs to be defined is the status subresource. In the status fields, the controller can save the current/actual state of the resource during runtime. While `.spec` defines the desired state, `.status` defines the actual state, as observed by the controller. For the promotion, a status subresource is also defined. In the status - the actual state of the resource as observed by the controller - most importantly the metadata of the currently opened pull request is saved, which the controller will pick up on every consecutive reconciliation of the same promotion object. Once the Go types are defined, the controller logic can be written.

6.1.7 Controller Logic

In this section, the controller logic of each implemented control loop, responsible for the according resource, is described. It runs on each reconciliation of an object. The designed logic is prototypical and can change in the future, depending on future requirements for the *GitOps Promotions Operator*. The full source code of the controllers can be found in Appendix C.

Environment Controller

For the environment API, a controller is written. For this prototype the following logic is implemented. It runs on every reconciliation of an environment object from start to finish. A diagram of the environment controller logic can be seen in figure 6.5.

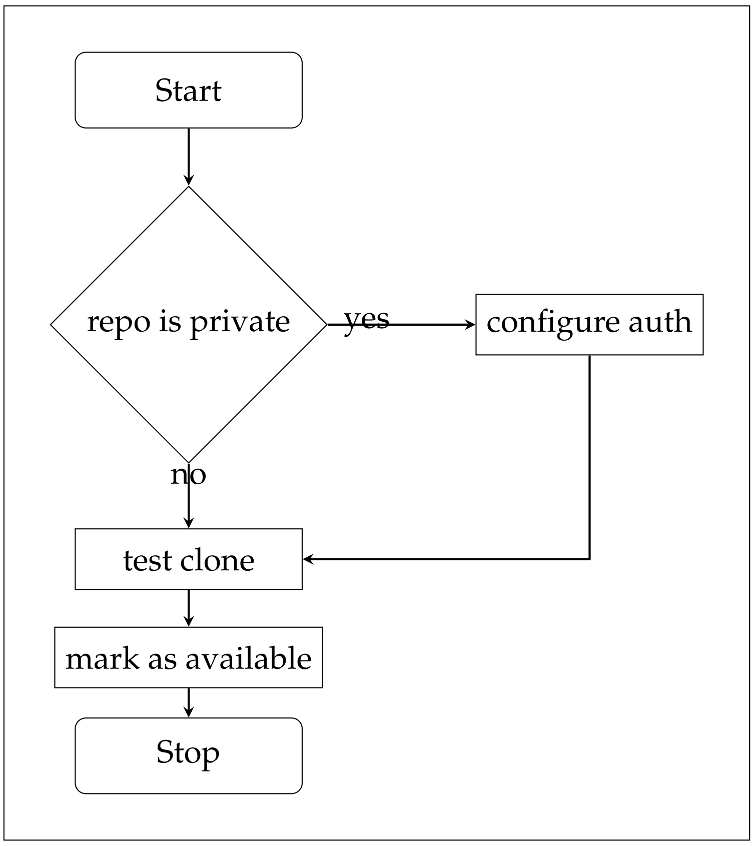


Figure 6.5: Environment Controller Logic.

At first, it is checked if the respective source Git repository of the environment object is private. If it is private, additional authentication options are configured, in order to be able to access the private Git repository. If it is publicly accessible, there is no need to configure authentication. Next, a Git clone process is tested for the environment. It is cloned locally, but afterwards disregarded. When it succeeds, the environment object is marked as available. This is achieved by updating the object's status. The status is observable by other controllers, like the promotion controller. This prototypical controller logic for the environment controller may be extended for additional functionality in the future.

Promotion Controller

For the promotion API, the following controller logic is implemented. It runs on every reconciliation of an environment object from start to finish. A diagram of the promotion controller logic can be seen in figure 6.6.

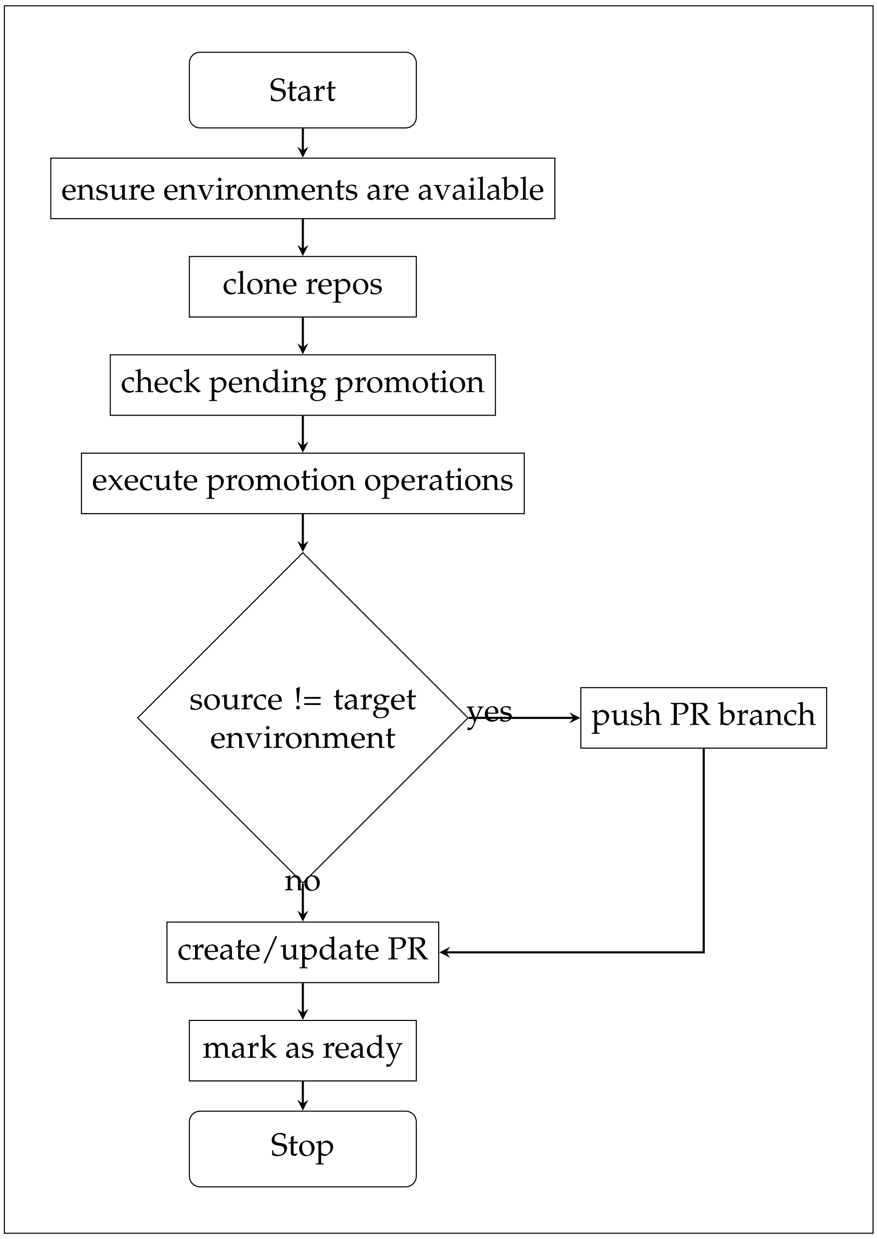


Figure 6.6: Promotion Controller Logic.

The controller checks if the source and target environments are ready, this includes the check of their defined dependent objects. If they are not yet ready, the controller cancels the reconciliation immediately. Then the source and target environments are cloned. Next the controller checks if there is a pending/open pull request, this information is retrieved from the object's status, and then checked if still up to date via the Git provider's API. Afterwards the controller executes the promotion operations. If there were changes since the last reconciliation, the new commits are pushed to the pull request branch. Then a pull request will be raised, if not yet done during a previous reconciliation. Lastly, the promotion is marked as ready in its status.

6.2 Demonstration

The following section demonstrates the in-context usage of the developed prototype - the *GitOps Promotions Operator* - in a proof of concept. The use case described as follows is created for the purpose of this demonstration.

This use case deals with a setup with multiple deployment environments. There are two non-critical environments `dev` and `qa`, and two production environments `prod-1` and `prod-2`. The GitOps definitions of the non-critical environments are living inside the same Git repository `mtpoc-infra-1`, and each production environment lives in its own separate Git repository `mtpoc-infra-2` for `prod-1`, and `mtpoc-infra-3` for `prod-2`. In general, the application version shall be promoted with a strict flow through the environments, one after the other. An overview of the given setup can be seen in the table 6.1.

Order	Environment	Source Repository
1	dev	mtpoc-infra-1
2	qa	mtpoc-infra-1
3	prod-1	mtpoc-infra-2
4	prod-2	mtpoc-infra-3

Table 6.1: PoC Environments Setup

The GitOps environment is centered around the used configuration management tool Kustomize, and generally structured for all environments as below:

```
.
|-- app-version
|   `-- kustomization.yaml
|-- kustomization.yaml
`-- settings
    `-- deployment.yaml
```

However it is independent of the tool Kustomize; any other tools can be used in conjunction with the proposed operator prototype. This structure adheres to the constraints of the currently implemented copy operation promotion type, which can copy files and directories. This means the configuration components which need to be promoted, should be defined in separate files or directories. This is needed, in order to only promote e.g. the application image version, while leaving other configuration untouched, and specific to an environment. With the Kustomize configuration tool, it is possible to split parts of the main `kustomization.yaml` into other separated files, with the components feature.

In this use case, the value of the application's image version lives within the app-version component. This is configured in the main `kustomization.yaml` like this:

```
components:
- app-version
```

The `./app-version` directory contains a `kustomization.yaml` file, with typical Kustomization specification. In this case, the images feature of Kustomize is used for configuring the application's image version tag.

```
apiVersion: kustomize.config.k8s.io/v1alpha1
kind: Component
images:
- name: ghcr.io/stefanprodan/podinfo
newTag: 6.3.4
```

Now the goal is to configure a promotion for the app-version component. To achieve this, an `Environment` resource needs to be created for all environments respectively. Only the `dev` environment is shown here, the other three environment definitions follow the same schema, but are omitted from this demonstration for the sake of brevity.

```
apiVersion: promotions.gitopsprom.io/v1alpha1
kind: Environment
metadata:
  name: dev
  namespace: default
spec:
  path: ./envs/dev
  source:
    url: https://github.com/thomasstxyz/mtpoc-infra-1
    ref:
      branch: main
    secretRef:
      name: mtpoc-infra-1-ssh
  dependentObjects:
    workloadRef:
    - kind: Deployment
      name: dev-podinfo
  apiTokenSecretRef:
    name: github-api-token
  gitProvider: github
```

Now the specified secrets must be created. The API token is required for the creation of pull requests by the promotion controller; it must be stored in a Kubernetes generic secret resource in a key named `token`, and can be created with the following command:

```
kubectl create secret generic github-api-token --from-literal=token="gh..."
```

With the current prototype, also a secret for the SSH connection to push and pull the repository, needs to be created. For this, a ssh key pair needs to be created by the user. Its public key needs to be set as a deploy key at the Git provider, and its private key needs to be stored in a Kubernetes generic secret resource in a key named `private`.

```
kubectl create secret generic github-api-token --from-literal=private="--..."
```

When all the four environments are created, the `Promotion` resources can be defined. In this use case, three promotion resources are needed for the ability to promote between all four environments with a straight flow - promoting from one to the next. Only the `dev-to-qa` promotion is shown here, the other two definitions follow the same schema, but are omitted here for the sake of brevity.

```
apiVersion: promotions.gitopsprom.io/v1alpha1
kind: Promotion
metadata:
  name: dev-to-qa
  namespace: default
spec:
  sourceEnvironmentRef:
    name: dev
  targetEnvironmentRef:
    name: qa
  copy:
  - name: "Application Version"
    source: app-version
    target: app-version
  strategy: pull-request
```

At this point, all which is needed for promoting is configured. When the status of all the environment resources involved in a promotion, have a ready status condition, and the dependent objects are successfully deployed, a promotion will trigger. At this point, the controller logs can be observed.

```
2023-04-16T12:10:46Z INFO    Created new pull request    [...]
```

A new pull request has been created by the controller. The promotion's status will also reflect, that a pull request is open for review.

```
status:
  conditions:
  - lastTransitionTime: "2023-04-16T12:10:45Z"
    message: A pull request is open for review.
    reason: Succeeded
    status: "True"
    type: Ready
  lastPullRequestNumber: 1
  lastPullRequestUrl: https://github.com/thomasstxyz/mtpoc-infra-1/
   pull/1
  observedGeneration: 1
```

The open pull request is ready for review in the Git provider's web interface.

The changed difference introduced by the commit can be viewed:

```
- newTag: 6.3.3
+ newTag: 6.3.4
```

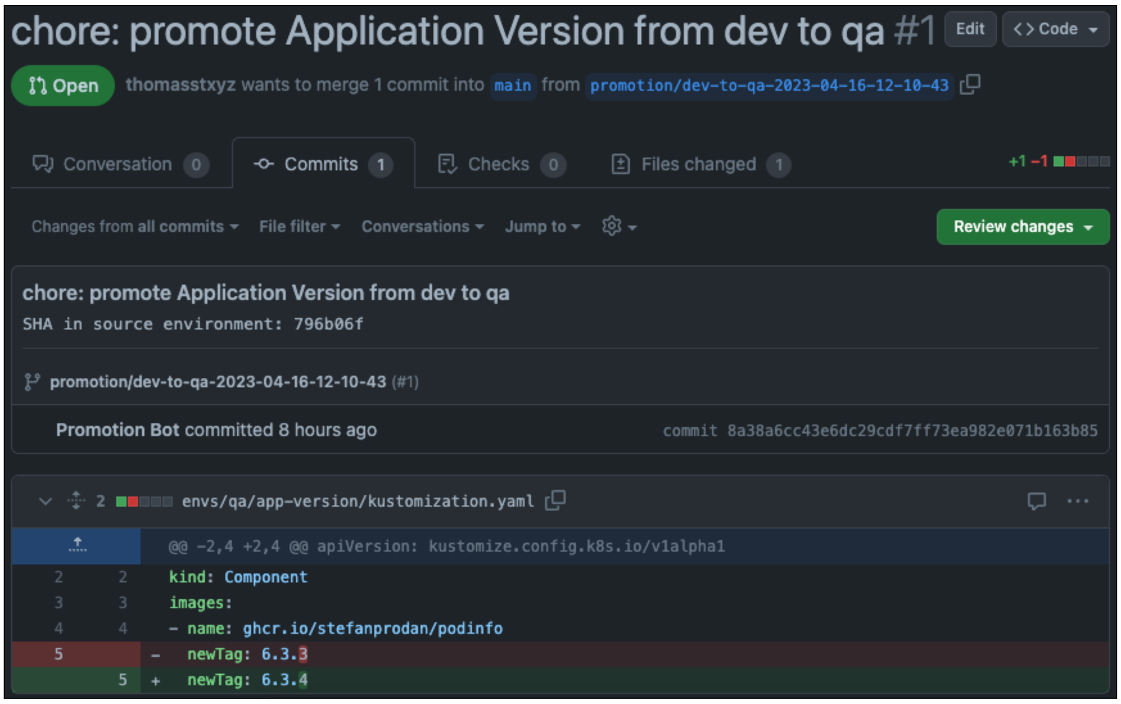
The `dev-to-qa` promotion requested the change of the image version from `6.3.3` to `6.3.4`.

Now, since the `qa` and the `prod-1` environments also differ, a pull request has also been created for this promotion.

The `qa-to-prod-1` promotion requested the change of the image version from `6.3.2` to `6.3.3`.

```
- newTag: 6.3.2
+ newTag: 6.3.3
```

Since the `prod-1` and the `prod-2` environments now also differ, a pull request has also been created for this promotion.


chore: promote Application Version from dev to qa #1

Open thomasstxyz wants to merge 1 commit into main from promotion/dev-to-qa-2023-04-16-12-10-43

chore: promote Application Version from dev to qa
SHA in source environment: 796b06f

promotion/dev-to-qa-2023-04-16-12-10-43 (#1)

Promotion Bot committed 8 hours ago   commit 8a38a6cc43e6dc29cdf7ff73ea982e071b163b85

2 envs/qa/app-version/kustomization.yaml

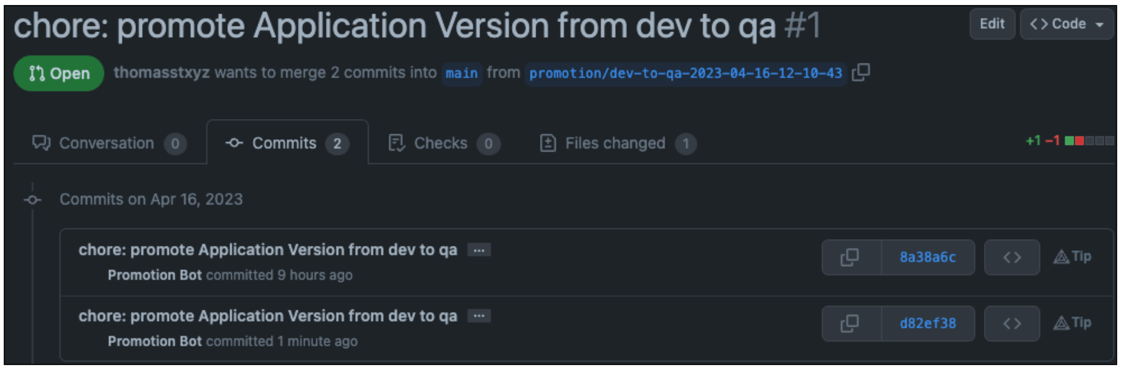
```
@@ -2,4 +2,4 @@ apiVersion: kustomize.config.k8s.io/v1alpha1
 kind: Component
 images:
 - name: ghcr.io/stefanprodan/podinfo
-  newTag: 6.3.3
+  newTag: 6.3.4
```

Figure 6.7: Pull Request for Promotion from dev to qa.

The prod-1-to-prod-2 promotion requested the change of the image version from 6.3.1 to 6.3.2.

```
- newTag: 6.3.1
+ newTag: 6.3.2
```

If the dev environment advances the application image version further, the pull request for the dev-to-qa will be updated with another commit. Note that the previous commit is not overwritten, but the commit history is kept on the pull request branch - now there are two commits on the branch.


chore: promote Application Version from dev to qa #1

Open thomasstxyz wants to merge 2 commits into main from promotion/dev-to-qa-2023-04-16-12-10-43

Commits on Apr 16, 2023

chore: promote Application Version from dev to qa
Promotion Bot committed 9 hours ago   8a38a6c

chore: promote Application Version from dev to qa
Promotion Bot committed 1 minute ago   d82ef38

Figure 6.8: Pull Request updated for Promotion from dev to qa.

The difference for the dev-to-qa promotion is now:

```
- newTag: 6.3.3
+ newTag: 6.3.5
```

Now if the pull request for the dev-to-qa promotion is approved and merged by a human, the promotion will actually take effect. This results in the qa-to-prod-1 promotion pull request being updated by the promotion controller. Version 6.3.5 is now requested for promotion to the prod-1 environment.

The difference for the `qa-to-prod-1` promotion is now:

```
- newTag: 6.3.2
+ newTag: 6.3.5
```

Once reviewed, approved and merged by a human, the promotion of version `6.3.5` to the `prod-1` environment will result in further propagation of version `6.3.5` to the `prod-2` environment.

The difference for the `prod-1-to-prod-2` promotion pull request is now:

```
- newTag: 6.3.1
+ newTag: 6.3.5
```

Once reviewed, approved and merged by a human, the promotion of version `6.3.5` to the `prod-2` environment will take effect.

The described demonstration showed, how an application version can be promoted across multiple environments, while ensuring a strict flow of promotion of e.g. `dev --> qa --> prod-1 --> prod-2`. Alongside the application version or any arbitrary resource, users of the prototype operator may specify multiple other promotion subjects in the copy operations list `.spec.copy`, which shall be promoted. For the sake of brevity, this was not shown in detail in the demonstration.

6.3 Evaluation

Now that the prototype has been demonstrated in a proof of concept, its functionality is compared against the solution objectives, as defined in section 5.2 earlier in this thesis. The evalutation is achieved by means of comparing the qualitative descriptions of the solution objectives against the observed functionality of the prototype in the demonstration in section 6.2.

Objective 1

The requirement 1 of objective 1 (OBJ1R1), formulated as a user story, was the following:

> As a user, I can define any filesystem path inside a Git repository, with a respective target path in a Git repository, as a promotion subject.

As demonstrated in section 6.2 the user of the *GitOps Promotions Operator* can define promotion subjects in the form of file/directory copy operations inside the Promotion custom resource. The source and target may be in separate Git repositories, respective *Environment* custom resources. This creates the possibility to promote arbitrary resources. The demonstration shows a promotion of the "Application Version", which is actually a specified file kustomization.yaml with a Kustomize component, which sets a new image tag. The defined name of the arbitrary filesystem path serves as the demonstration for the requirement 2 (OBJ1R2).

> As a user, I can define a descriptive name for a promotion subject, which is represented as an arbitrary filesystem path.

The definition of such a descriptive name helps to identify promotion subjects more easily, especially when they are arbitrary files or directories. The name can be represented in the commit message or pull request, as demonstrated.

Objective 2

The requirement 1 of objective 2 (OBJ2R1) was the following:

> As a user, I can promote releases through multiple environments in a certain user-defined order.

The prototypical implementation of the functionality of the solution objective is demonstrated in section 6.2. It is shown, how a new application release can be promoted through multiple environments in a specified order. After deployment to the `dev` environment, a promotion is requested for the `qa` environment. Once a human has approved and merged the pull request, the promotion to `qa` takes its effect. Afterwards the promotion from `qa` to `prod-1` environment is requested. Upon successful promotion to `prod-1`, the release is finally promoted from `prod-1` to `prod-2`.

Objective 3

The requirement 1 of objective 3 (OBJ3R1) was the following:

> As a user, I can define Kubernetes objects as dependencies for a promotion.

The functionality of this objective is demonstrated in context in the proof of concept in section 6.2. In the `Environment` custom resource, a field `dependentObjects.workloadRef` may be defined, under which a user can specify a list of Kubernetes workloads of the kind Deployment. The developed prototype supports object types of kind Deployment, however, any Kubernetes native resource, as well as custom resource, can potentially be added as an additional feature for the prototype. For example, a field `dependentObjects.externalHttpRef` could be added, with the logic for calling HTTP/S URIs, and parsing the result.

Objective 4

The requirement 1 of objective 4 (OBJ4R1) was the following:

> As a user, I can use any GitOps engine, Git provider and configuration/templating tool.

The developed prototype *GitOps Promotions Operator* supports the use of GitHub currently, however the custom resource API is designed with a generic specification and therefore allows for adding the support for any other Git provider in the future. The same vendor-neutral approach was chosen for the GitOps engine. The *GitOps Promotions Operator* prototype allows the use of any GitOps engine, because it interfaces only with Kubernetes built-in resources. Furthermore, the prototype is agnostic to the configuration/templating tool which may be used or not. Since the operator provides the ability to promote arbitrary files or directories in Git repositories, it is not needed to specifically integrate with the named tools.

6.4 Summary

In this preceding chapter, the proposed prototype was presented. The asynchronous nature of GitOps deployments, and where the operator prototype fits within this architecture was described. Next, abstract models for the *Environment* and *Promotion* custom resources as well as their prototype design as declarative Kubernetes custom resources was described. The implementation of these custom resources was shown in the form of mockups of Kubernetes custom resources in the Yaml format. Alternative mockup designs were shown as a way to draw attention to the fact that the actual design of the API specification may be changed as desired. The translation of the API specification in Yaml format into Go types was described, and finally the implemented controller reconciliation logic of both the environment, as well as the promotion controller was presented.

In the next step, the developed artifact of the prototype operator was demonstrated in the context of a proof-of-concept use case. The demonstration of the prototype's functionality was then evaluated against the research objectives defined in 5.2.

7 Evaluation and Results

In this chapter, the results of the research will be presented, and evaluated with a holistic view on the research problem of release promotion in GitOps environments. The results primarily stem from the designed and developed prototype, and the learning from the prototyping process. The conducted interviews with the working professionals also gave several interesting insights into the problem statement from their point of view. Next to the presentation of the results, they are also evaluated on how they provide a solution to the problem. Especially the functionality implemented in the prototype is evaluated against the research objectives, which have been defined, and map directly to distinct problem items.

The evaluation of the prototype's functionality against the solution objectives, was already carried out in section 6.3 earlier in the thesis. It was proven, that for each solution objective defined in section 5.2 the respective functionality was implemented in the prototype, and demonstrated in a proof of concept. It was described, how for each solution objective, a solution to the respective problem can be observed in the proof of concept demonstration in section 6.2. The research results are evaluated with a holistic view, with the focus on the research questions defined in section 1.2.

RQ 1.1: How can deployment environments, as well as promotion processes be modeled abstractly?

A possible solution to this research question was presented in chapter 6 by means of describing the design of a prototype of a Kubernetes operator for handling the operations of promotions for GitOps environments. Section 6.1.1 describes the asynchronous nature of GitOps deployments, and where the proposed operator fits within this architectural pattern. Section 6.1.2 presents a qualitative description of how abstract models for environments and promotions in the context of the operator pattern could be designed.

RQ 1.2: How can the abstract models be used to implement a standardized solution for promoting releases?

Described in sections 6.1.3, 6.1.4 and 6.1.5, a possible implementation of the abstract models is presented. It is in the form of declarative custom resources for extending the Kubernetes API. Moreover in section 6.1.6, the translations of the custom resources into Go types, which are used in the Kubebuilder framework, are described. In section 6.1.7, the proposed controller logic for the environment, as well as the promotion controller is presented. The proposed prototype is implemented and demonstrated in a proof of concept in section 6.2. How well the implemented functionality provides a solution to the research objectives, is evaluated in section 6.3.

RQ 1: How can the promotion of releases in GitOps environments be designed?

Chapter 6 proposes the design and development, as well as a possible implementation of a prototype capable of promoting releases in GitOps environments. The basis for the prototype's functionality are the defined solution objectives (section 5.2). The combined evaluations of the sub research questions provide a possible answer to the overarching research question for the thesis. This thesis proposes one possible way of how the promotion of releases in GitOps environments can be designed. This concrete research does not try to propose a definitive answer or solution to the research question. The interview partners discussed insights into related ideas and approaches (section 5.3), which can be used to design alternative ways for promoting releases in GitOps environments.

8 Discussion and Interpretation

In this chapter, the results and evaluations which were presented in the previous chapter 7, are discussed. The meanings behind the specific results are brought forward in more detail. Moreover, interpretations and implications of the results and evaluations are presented by the researcher.

Learnings From The Prototype Implementation

In general, the presented design and development of the prototype showed a possible way on how to provide a solution to the research question. From the actual implementation of the abstract design in a Kubernetes operator, and its in-context use in the proof of concept, certain learnings could be inferred.

User Experience: Due to Objective 4: Vendor-neutral, tool-agnostic, the prototype has been designed to be agnostic to the tooling used, which includes the GitOps engine, the Git provider, and the configuration/templating tool. While it is good that the prototype can work with many other tools, and does not enforce the use of any specific tool for the mentioned components, the user experience can lack, as a result.

Often times users who want to setup a GitOps workflow, implement either e.g. Argo or Flux as their primary GitOps tool which also provides the main engine. When the proposed prototype operator with its according custom resources for environments and promotions should be set up additionally, then some information essentially needs to be specified twice. For example, an environment resource of the prototype operator defines the URL of the Git repository, which also was needed to be defined for the GitOps engine, e.g. Flux GitRepository or ArgoCD Application. The access credentials to the Git repository, i.e. the SSH deploy key, also needs to be setup once for the typically used Flux or Argo GitOps engine, as well as the promotions operator.

A possible solution to this issue could be to directly integrate with Flux or Argo, since these are the most popular GitOps tools, and most likely already installed and used by users of the promotions operator. Platforms like GitLab or AWS EKS have the Flux toolkit already built-in, while for example, OpenShifth as a built-in version of ArgoCD, therefore it makes sense to integrate.

Security Considerations: The designed and implemented prototype operator can generally run in any Kubernetes cluster. It is independent from the deployment environments, so it could either run in the deployment environment itself, or alternatively in a management cluster, which would then be responsible for the promotion of multiple environments. However, when the resource dependency for a promotion is a workload or another resource within the environment, which is to be promoted, it makes sense to have the operator run inside the same deployment environment, i.e. Kubernetes cluster or namespace, as specified in the Environment custom resource.

The goal with the promotion resource is generally to specify two environments, a source and a target environment. While the source environment typically would be the same environment in which the operator runs in, the target environment would merely represent the Git repository of the target cluster/namespace. This raises some security considerations, since the operator can read and write to the target environment's desired state, i.e. Git repository. This means, that the operator running in a certain environment would have read and write access to the specified target environment of the promotion. With this setup, bad actors who have access to the specified environment resources state, could change the desired state in such a way, that potentially harmful applications are deployed to the other environment. Since the operator needs appropriate permissions for the environment's Git repository, in order to raise pull requests and push to pull request branches, the operator could be a potential security issue.

Use at Scale: The use of the promotions operator prototype at scale needs to be addressed. Using the operator at scale could either be to install the operator in a separate management cluster, or to install the operator in each environment. In order for the dependency capability of the promotion controller to be able to check for dependent resources within an environment, which is a source environment for a promotion, the operator ideally should be running inside this same environment, i.e. cluster/namespace.

When installing the promotions operator once in a management cluster/namespace, in order to handle all or many environment promotions, the user would save time on the initial setup of the custom resources, deploy keys and API tokens. However with this approach, it would also mean that the operator running in a completely separated management cluster, typically has no direct access to observe the resources of an environment.

Abstractions: The proposed prototype provides two main abstractions, one for the environment, and one for the promotion. Usually such a custom resource object represents a resource in the real world. Although not part of the proposed design and implementation of the prototype, it would make sense to provide more abstractions. For example, this could be to divide the environment resource into a DeploymentEnvironment resource, which would represent a Kubernetes cluster or namespace, and a GitOpsRepository resource, which would represent the according GitOps repository (the desired state definition). It is generally a good practice to have custom resources represent real tangible resources, as opposed to representing multiple resources.

Modularity: In addition to the topic of abstractions mentioned previously, it could be beneficial to split up the Promotion custom resource. There could be a PromotionPolicy resource, which would define the policy and rules, when the operator should promote and create a Promotion resource. The Promotion resource would then only run once to completion.

Alternative Approaches for Promoting Releases

Interview partner 1 discussed an alternative approach for handling the promotion process in GitOps environments. This was presented in section 5.3. The main idea of this approach is that long-living environments are not necessary, and the resiliency should rather be created by doing progressive delivery, not just for the user-facing application or service, but for the whole infrastructure stack below. This has the purpose of further increasing the immutability and resiliency of a service. Since the amount of supporting and infrastructure services and dependencies are increasing with Kubernetes, and each having a version and constantly new releases, the possibility of breaking the actual service that is user-facing also increases respectively.

When following this approach, the end goal is to create a complete copy of the production environment, and then do progressive delivery on that. Once the release is marked as good, the old production environment can safely be destroyed. With the GitOps pattern, and the application and infrastructure being stored in Git, this has become increasingly more possible. Tools like the GitOps Terraform Controller have contributed to the enablement of this new approach. While, with this approach, the need for long-living environments decreases, whole copies of production environments will still need to be created. This means there is no guarantee that the cost will decrease. More advanced techniques like auto-scaling will need to be implemented, in order to keep costs low of potentially high numbers of dynamically created environments.

9 Future Work

In this chapter, further suggestions for future research on this topic and the developed prototype are presented. In addition, some interesting ideas that came out of this research are discussed, and how they could be further researched in future work either by the researcher of this thesis or other researchers.

Further Research & Development of the Prototype

The proposed software prototype should be evaluated against its user experience, in order to find out if users have difficulty doing the initial setup, or any other troubles. The use of the prototype could be observed in a case study or tested in a field experiment. The prototype should be adapted in more iterations of the applied methodology process model. The API, the custom resources, which the operator provides, can be changed to fit the users needs. Additional functionality which is required by users, can be added. Since the operator works with any other GitOps tooling the users bring from their individual GitOps setup, it can lack a bit of ease of use. The integration into other GitOps tooling like Argo or Flux should be evaluated, and examined if such an integration is worth it, in order to provide better convenience. The main goal is to extensively adapt for user requirements, in order to fit many possible promotion processes. The prototype should be further developed to enable the use in production by organizations and other projects.

Rolling Production Environments

The term rolling production environments was coined by interview partner 1 during the interview. It was discussed in section 5.3 earlier in this thesis. The term represents a future idea of where promotions and rollouts of new releases could be heading to. This approach steers in a somewhat other direction, than what this thesis researched. Instead of having fixed testing environments or stages, which a new release needs to pass, before being rolled out to the production environment, this approach conversely foresees only the single environment, which is actually used for production.

This approach builds upon the progressive delivery pattern, which can be implemented with tools like Flagger or Argo Rollouts. Instead of doing progressive delivery of an application (i.e. Kubernetes deployment and container image), the progressive delivery would be done on a whole environment (i.e. Kubernetes cluster). This has the advantage of further limiting the chance of supporting services and applications influencing the actual user-facing application in a bad way, i.e. breaking the service, making it unavailable for its users. Each different supporting application constantly changes versions, and all these changes could potentially break some other dependency. Some infrastructure components are sometimes even updated without a version history, eliminating the possibility to roll back to a safe and working state.

One possible future work that could be done for evaluating this new approach, could be to evaluate the costs and feasibility of this approach. According to interview partner 1, tools like the GitOps Terraform Controller have made it a lot easier to achieve the idea of rolling production environments. However since a whole production environment would be dynamically created with each new release, this could introduce higher costs. Especially in a cloud environment, where billing is done on-demand per minute used, and where the costs can be an important factor for deciding between different architectures.

Overview of GitOps Repositories

Discussed by interview partner 2 and presented in section 5.3 is the idea and problem, that when a human is given a GitOps repository, it is often difficult to understand how the setup is exactly structured, what the environments are, what the versions are, and what applications and version are deployed where. The overview that GitOps shall give, with the single place to look and know what is your system's state, is not as good as it is expected, interview partner 2 mentioned.

As possible future research on this idea, this problem statement could be evaluated with a survey research against GitOps users. In addition, especially if the survey's results speak for this statement, a software tool could be developed, which can recognize filesystem structures and plain text configuration files, which represent the desired state. This tool would have have knowledge of the different configuration management tools like Kustomize or Helm. It would probably be beneficial to have a visual representation in the form of a dashboard as well.

Such a visual representation is already provided by ArgoCD for example, however it will only show the configured data in the dashboard. Any files, which are not yet added to the specific ArgoCD instance, can only be viewed by having a look into the Git repository's filesystem manually.

Towards Standardized GitOps Promotions

There is no standard way of doing promotions with the GitOps approach. Sometimes a container image tag is changed in some place or multiple places, other times multiple files are copied over to another place (like the promotion process proposed with the prototype in this thesis), and other times a promotion consists of multiple processes spanning over different domains. This differs for each individual setup for distinctive organizations.

In order to get a better understanding of what the requirements are for different organizations, and how they imagine a solution, and solved their individual use case, it would be of use to survey a wide range and variety of organizations which adopted or want to adopt the GitOps approach. For open-source tooling it is beneficial to strive for functionality that can be used by everyone, instead of providing tailored tooling which may only work for a single setup at a particular organization.

10 Conclusion

In this final chapter, no new information is presented, but what has already been said is summarized again. The most important key points of the thesis are highlighted, in order for the reader to easily consume.

Problem

The increasing adoption of a DevOps culture in organizations to develop applications and services quickly, and reduce friction between people, communications and technical processes, to ultimately decrease the time to market for new product releases, has brought forward a new practice called GitOps.

One of the unresolved problems of the GitOps practice is the process of promoting releases between multiple deployment environments. Current tools in the ecosystem do not provide an integrated solution for this process. Users are therefore inclined to build workflows which are constrained to specific Git providers, GitOps engines, workflow/pipeline systems, and configuration/templating tools. This can lead to tightly coupled setups, and vendor lock-in.

In this research, the given problem was addressed by designing uniform, standardised models for defining GitOps-native deployment environments and promotion processes. These models were implemented in a prototype as custom resources and controllers with the operator pattern, as a Kubernetes extension. This developed software artifact allows users to define abstract representations of their environments, and how they want releases to be promoted between them.

Research Question

The overall goal of the thesis was to provide a solution to the problem of release promotion in GitOps environments. Therefore this overarching research question was defined: RQ 1: How can the promotion of releases in GitOps environments be designed?; with the following sub research questions: RQ 1.1: How can deployment environments, as well as promotion processes be modeled abstractly? RQ 1.2: How can the abstract models be used to implement a standardized solution for promoting releases? The research question is elaborated with a scientific methodology.

Methodology

In order to help with recognition and legitimization of the conducted research, the methodology for conducting design science research in information systems (Peffers et al., 2007) was applied, which consists of six activities. In activity 1, the research problem of release promotion with GitOps was defined. This was done with the help of practicing professionals in the GitOps field, which were interviewed. In activity 2, research objectives were inferred from the problem definition in activity 1. Each objective maps to a distinct item from the problem specification, which helped with later evaluation in activity 5. In activity 3, solutions for the previously defined objectives were designed and developed by means of producing an artifact, namely the *GitOps Promotions Operator* prototype. In activity 4, the in-context use of the artifact was demonstrated in a proof of concept. In activity 5, the implementation of the artifact, and how well it supports a solution to the problem, was evaluated. In activity 6, as a final step, the whole conducted research was communicated by means of publishing it as a master thesis.

Related Work

Prior research on the concrete problem is focused on presenting good practices and suggestions which users need to manually implement themselves. In addition, it is suggested to let external workflow/pipeline systems handle the promotion process, or limit the amount of environments to one, in order to avoid having to do promotions altogether. Conversely, this thesis brought forward abstract models of environments and promotion processes, which are implemented in the proposed prototype operator, as Kubernetes custom resources and controllers, with the operator framework. The prototype assesses the feasibility of defining deployment environments and promotion processes declaratively, following the GitOps principles.

Theoretical Background

The theoretical background on the topic was brought forward to the user, in order to aid comprehension of the material within the thesis. General definitions of terms, fundamentals of DevOps and GitOps along related tooling and components were presented. It was shown how GitOps changes the architecture and process of Continuous Deployment, and how the promotion of releases is achieved without and with the GitOps approach. The modeling approach of GitOps environments was discussed. Emerging patterns like progressive delivery, as well as the concept behind short-living environments were described. The role of Kubernetes as a cloud native platform and its use cases beyond container orchestration were described.

Interviews

For the problem identification and motivation of the main topic of this thesis, interviews with practicing professionals, who are working in the GitOps field, were conducted. Several problems were identified, and defined along with their respective research objectives.

Problem 1: Promotion is limited to container image, relates to a frequent issue with currently available tooling in the GitOps ecosystem. Often times solely container image version tags are the focus with current tools when promoting new versions or releases. Because it is sometimes required to handle all sorts of resources, not just the version tag of a container image, an according research objective was defined. *Objective 1: Arbitrary resources can be promoted*, defines a qualitative description of how the respective problem is supposed to be solved by the developed artifact. The main idea is to offer the capability to promote arbitrary resources, meaning any type of resource, instead of solely the container image version.

Problem 2: Order of promotion to multiple environments, states the fact, that it is not a straight-forward process of how the order of promotion through multiple GitOps environments can be setup. *Objective 2: Strict flow of promotion through environments*, defines the requirements for the proposed prototype, in regards to the according problem of having a certain order of promotion through environments or stages. The objective describes the capability for defining a certain order of environments, in which releases traverse through. In addition, this solution objective opens up the possibility to setup promotion in stages, in which certain environments must be deployed to first, before the release can deploy to other specified environments.

Problem 3: Dependencies can not be defined, relates to the problem that when wanting to promote a new release from one environment to another environment, it is not easily achievable with the available tools to specify certain dependencies, like other workloads or microservices in the same or another environment, or dependencies from external sources. This is especially desirable for evaluating test results or other metrics, before triggering the promotion. *Objective 3: Dependencies of a promotion*, describes how the respective problem of being able to specify dependencies for a promotion, could be solved in the proposed prototype. While the minimum dependency is the successful deployment of the workload of a release, it may also be desirable to specify other resources or workloads which need to be in a certain state, before triggering a promotion.

Problem 4: Provider and tool dependency, draws attention to the common problem of being dependent on particular tools and providers. The more complex the Continuous Delivery is setup for a project, the more difficult it is to de-couple or switch providers for certain components. *Objective 4: Vendor-neutral, tool-agnostic*, defines the requirements of how a vendor-neutral and tool-agnostic prototype can be implemented. The promotions operator supports any GitOps engine, Git provider, and configuration tool.

Additionally, related ideas and approaches were discussed by the interview partners. These points were not directly considered for the conducted design science in the prototype, however they were discussed later in the thesis.

Prototype

The proposed prototype was presented. The asynchronous nature of GitOps deployments, and where the operator prototype fits within this architecture was described. Abstract models for the environment and promotion custom resources as well as their prototype design as declarative Kubernetes custom resources was described. The implementation of these custom resources was shown in the form of mockups of Kubernetes custom resources in the Yaml format. Alternative mockup designs were shown as a way to draw attention to the fact that the actual design of the API specification is not cast in stone. Moreover, the API specification should be tested with users, and should be adapted for usability and ease of use. The translation of the API specification in Yaml format into Go types was described, and finally the implemented controller logic of both the environment, as well as the promotion controller was presented. The developed artifact of the prototype operator was demonstrated in the context of a proof-of-concept use case. The demonstration of the prototype's functionality was then evaluated against the research objectives.

Evaluation & Results

The results of the conducted research were presented, primarily by means of presenting and describing the designed and developed operator prototype, and the learning from the prototyping process. In addition, the interviewed working professionals gave several interesting insights into the problem statement from their point of view. The results of the research were evaluated on how they provide a solution to the research problem. The implemented functionality of the prototype was evaluated against the research objectives. This was done by comparing the qualitative descriptions of the objectives with the actual observed results in the demonstration of the prototype in the proof of concept. For the research question RQ 1.1, a possible solution was presented by means of describing the design of a prototype of a Kubernetes operator for handling GitOps promotions. For research question RQ 1.2, a possible implementation of the abstract models was presented, in the form of declarative custom resources, which extend the Kubernetes API. The overarching research question 1 is the combination of the sub research questions. The thesis proposes one possible way of how the research problem can be addressed, namely the promotion of releases in GitOps environments can be designed. This concrete research does not try to propose a definitive answer or solution to the research question.

Discussion & Interpretation

The results, learnings, and evaluations of the research were discussed and interpreted. The meanings behind the specific results are brought forward in more detail. Moreover, interpretations and implications of the results and evaluations were presented. Learnings from implementing the prototype were presented, namely ideas about the user experience, security considerations, the use at scale, abstractions and modularity. Alternative approaches for promoting releases were presented.

Future Work

Further suggestions and point of references for future research on this topic and the developed prototype were presented. These included further research and development of the proposed prototype, which is about testing its user experience, evaluating integration with other GitOps tools. Generally the aim is to enhance the prototype to make it mature for production use. The idea of rolling production environments by interview partner 1 was discussed. It describes a somewhat different approach for promotions in GitOps, where less environments are needed, but for each new release the production environment is re-created with the new versions, and progressive delivery is done not only on an application level, but on the whole infrastructure stack together with the end user application or service on top. This is to further improve immutability and versioning to increase resiliency. The idea of the problem with the overview of GitOps repositories by interview partner 2 was discussed. It is about the somewhat missing feature of a quick and easy to understand overview over a bare GitOps repository. Depending on the used configuration/templating tool, a setup looks different. Deployment environments can be represented, however it is not possible for the user to know what the target environment is, or where the GitOps definition is deployed to in general. Moreover it was discussed that it would make sense for future work to research a wide range and variety of organizations and do a survey on their requirements, their issues and how they imagine a solution. An initiative towards standardized GitOps promotions should be made, because for open-source tooling the aim should be to strive for functionality that can be used by everyone, instead of providing tailored tooling which may only be beneficial for specific use cases and organizations.

Bibliography

Arundel, J., & Domingus, J. (2019). Cloud Native DevOps mit Kubernetes: Bauen, Deployen und Skalieren moderner Anwendungen in der Cloud. *Deployen und Skalieren moderner Anwendungen in der Cloud. dpunkt. verlag, Heidelberg*.

Beetz, F., Kammer, A., & Harrer, D. S. (2021). *GitOps: Cloud-native Continuous Deployment*. innoQ Deutschland GmbH.

Berger-Grabner, D. (2016). *Wissenschaftliches Arbeiten in den Wirtschafts-und Sozialwissenschaften*. Springer.

cncf.io. (2023). 2022 The year cloud native became the new normal [(Accessed on 05/20/2023)]. https://www.cncf.io/reports/cncf-annual-survey-2022/

cncf-tag-app-delivery-operator-wg. (2023). Operator Whitepaper v1 [(Accessed on 04/13/2023)]. https://github.com/cncf/tag-app-delivery/blob/21b33d1e4d650a081794504a9ae733b53ed8dbf1/operator-whitepaper/v1/Operator-WhitePaper_v1-0.md

Cohn, M. (2004). *User stories applied: For agile software development*. Addison-Wesley Professional.

docs.gitops.weave.works. (2023). Promoting applications through pipeline environments [(Accessed on 04/13/2023)]. https://docs.gitops.weave.works/docs/enterprise/pipelines/promoting-applications/

form3tech-oss. (2023). k8s-promoter [(Accessed on 05/05/2023)]. https://github.com/form3tech-oss/k8s-promoter

Fowler, M., Highsmith, J., et al. (2001). The agile manifesto. *Software development*, *9*(8), 28–35.

Gläser, J., & Laudel, G. (2010). *Experteninterviews und qualitative Inhaltsanalyse*. Springer-Verlag.

helm.sh. (2023). The package manager for Kubernetes [(Accessed on 05/10/2023)]. https://helm.sh/

Hightower, K., Burns, B., & Beda, J. (2017). Kubernetes: Up and Running Dive into the Future of Infrastructure. OReilly Media. *Inc., Sebastopol*.

Jabbari, R., bin Ali, N., Petersen, K., & Tanveer, B. (2016). What is DevOps? A systematic mapping study on definitions and practices. *Proceedings of the Scientific Workshop Proceedings of XP2016*, 1–11.

Kapelonis, K. (2022). How to Model Your Gitops Environments and Promote Releases between Them [(Accessed on 01/01/2023)]. https://codefresh.io/blog/how-to-model-your-gitops-environments-and-promote-releases-between-them/

Kapelonis, K. (2021). Stop Using Branches for Deploying to Different GitOps Environments [(Accessed on 01/01/2023)]. https://codefresh.io/blog/stop-using-branches-deploying-different-gitops-environments/

kargo.akuity.io. (2023). Kargo [(Accessed on 05/04/2023)]. https://kargo.akuity.io/

Kubebuilder-Authors. (2023). Kubebuilder book [(Accessed on 04/13/2023)]. https://book.kubebuilder.io/introduction.html

kubernetes.io. (2023a). Custom Resources [(Accessed on 04/25/2023)]. https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/

kubernetes.io. (2023b). Kubernetes [(Accessed on 04/13/2023)]. https://kubernetes.io/

kubernetes.io. (2023c). Kubernetes [(Accessed on 04/13/2023)]. https://kubernetes.io/docs/concepts/extend-kubernetes/

kubernetes.io. (2023d). Kubernetes [(Accessed on 04/13/2023)]. https://kubernetes.io/docs/concepts/architecture/controller/

kustomize.io. (2023). Kubernetes native configuration management [(Accessed on 01/01/2023)]. https://kustomize.io/

opengitops.dev. (2023a). GitOps Glossary v1.0.0 [(Accessed on 01/01/2023)]. https://github.com/open-gitops/documents/blob/v1.0.0/GLOSSARY.md

opengitops.dev. (2023b). GitOps Principles v1.0.0 [(Accessed on 01/01/2023)]. https://github.com/open-gitops/documents/blob/v1.0.0/PRINCIPLES.md

Peffers, K., Tuunanen, T., Rothenberger, M., & Chatterjee, S. (2007). A design science research methodology for information systems research. *Journal of Management Information Systems*, *24*, 45–77.

Riedl, R. (2019). *Digitale Transformation erfolgreich gestalten*. De Gruyter Oldenbourg. https://doi.org/doi:10.1515/9783110471274

Sánchez-Gordón, M., & Colomo-Palacios, R. (2018). Characterizing DevOps culture: a systematic literature review. *Software Process Improvement and Capability Determination: 18th International Conference, SPICE 2018, Thessaloniki, Greece, October 9–10, 2018, Proceedings 18*, 3–15.

Verona, J. (2018). *Practical DevOps: Implement DevOps in your organization by effectively building, deploying, testing, and monitoring code*. Packt Publishing Ltd.

Wayfair-Tech–Incubator. (2023). Safe and Controlled GitOps Promotion Across Environments/Failure-Domains [(Accessed on 04/15/2023)]. https://github.com/wayfair-incubator/telefonistka

weave.works. (2023). Scaling GitOps in 2023 [(Accessed on 05/20/2023)]. https://www.weave.works/blog/scaling-gitops-in-2023

XenitAB. (2023). GitOps Promotion [(Accessed on 05/05/2023)]. https://github.com/XenitAB/gitops-promotion

Yuen, B., Matyushentsev, A., Ekenstam, T., & Suen, J. (2021). *Continuous Deployment with Argo CD, Jenkins X, and Flux*. Manning Publications Co. LLC New York.

List of Tables

List of Figures

Listings

Acronyms

API	Application Programming Interface
REST	Representational state transfer
YAML	A human-friendly data serialization language

Declaration of Academic Honesty

I hereby declare on my word of honor that I created this thesis at hand independently without the use of unauthorized aids and have not used any sources other than those cited. All passages that were taken literally or analogously from the specified sources are marked as such.

If the head of the degree program intends to use tools (in particular IT and AI-supported), I declare that I have listed these in full in my thesis with the respective product name, the product version and a description of the range of functions used.

I further declare that I have written this thesis in accordance with the applicable examination regulations of the University of Applied Sciences Burgenland and the guidelines of the Austrian Agency for Scientific Integrity for Good Scientific Practice. The thesis has not yet been submitted for review or assessment, either domestically or abroad, and has not been published.

Location, Date Signature