Strategies for Promoting Releases in GitOps Environments

Thomas Stadler, BSc
FH Burgenland, Eisenstadt, Österreich

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.

1 INTRODUCTION

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. 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.

2 RELATED WORK

Until now, there have not been any peer-reviewed publications in academic journals or conferences on the specific topic. Some textbooks exist on environment promotion, which do not necessarily incorporate 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).

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 RESEARCH QUESTION

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?

4 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. 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).

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 well as analysing prior written literature. To assist later evaluation, the problem is conceptually broken down into distinct items. 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). 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, 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. 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. 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. 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. 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 In addition, it is communicated to relevant audiences such as the GitOps Working Group of the CNCF.

5 RESULTS

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 evalutation is achieved by means of comparing the qualitative descriptions of the solution objectives against the observed functionality of the prototype in the demonstration. It was shown, that for each solution objective 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.

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, 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. 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. 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. 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 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. 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.

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.

The working professionals discussed alternative approaches for GitOps promotions. Interview partner 1 discussed an alternative approach for handling the promotion process in GitOps environments. 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.

6 CONCLUSION AND FUTURE WORK

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.

The overall goal of the thesis was to provide a solution to the problem of release promotion in GitOps environments. In order to help with recognition and legitimization of the conducted research, the methodology for conducting design science research in information systems was applied. The scientific method of semi-structured interviews was used to support the problem identification and motivation. A software artifact was designed and developed. A prototypical implementation of the *GitOps Promotions Operator* was demonstrated, and its functionality was evaluated against the research objectives.

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.

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.

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.

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 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. 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. 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.

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.

References

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

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/

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

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

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/

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.

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.