Bachelor Thesis

Infrahub meets K8s

Semester FS 2025

Version 1.0
June 12, 2025

Students: Simon Linder
Ramon Stutz

Advisors: Urs Baumann
Jan Untersander
Expert: Damien Garros

INS | Institute for Network and Security

OST Eastern Switzerland University of Applied Sciences

Department of Computer Science
OST - Eastern Switzerland University of Applied Sciences

1. Abstract

As part of the **"Infrahub meets K8s"** project, an automated system for configuration and deployment management in Kubernetes environments was designed and implemented. At its core, the system utilizes **Infrahub** as a centralized inventory system and authoritative source of truth, where the desired state of Kubernetes resources is defined. A custom Kubernetes Operator, **VIDRA**, continuously monitors this desired state and ensures its reconciliation by automatically deploying the corresponding resources into the Kubernetes cluster. With this approach, it becomes feasible to manage the entire infrastructure of a company declaratively and consistently through Kubernetes, enabling streamlined operations and improved scalability.

System Components

- **Infrahub:** Central inventory system and source of truth for all Kubernetes configurations.
- **VIDRA Operator:** A custom Kubernetes Operator written in Go (using the Operator SDK), responsible for continuously reconciling the desired state from Infrahub and applying changes to the cluster.
- **Python Transformers:** Modules that convert Infrahub's structured infrastructure data into valid, deployable Kubernetes manifests.
- **Flask Web Interface:** A lightweight, user-friendly frontend that allows self-service infrastructure requests.
- **VIDRA CLI:** A terminal-based command-line interface built with Cobra for interacting with the VIDRA Operator.

Key Features

- **Automated Deployment:** Enables streamlined and hands-off provisioning of infrastructure and configuration.
- **State Reconciliation:** Ensures that the actual state in the Kubernetes cluster always matches the declared state in Infrahub, detecting and resolving configuration drift automatically.
- **Centralized Management:** Offers unified visibility and control over infrastructure resources through a single, authoritative source of truth.

Conclusion

The **"Infrahub meets K8s"** project presents a modern, extensible framework for infrastructure automation in Kubernetes-based environments. By unifying centralized configuration management in Infrahub with continuous deployment and reconciliation via the custom **VIDRA** Operator, the system embodies core GitOps principles.

2. Vision

Vision

The vision of this project is to revolutionize the way we manage and deploy Kubernetes resources by bridging the gap between infrastructure modeling and deployment. By leveraging Infrahub as a centralized inventory system, we aim to automate the generation of Kubernetes manifests in a way that is easy to use for everyone, even those who are not familiar with Kubernetes. Infrahub will serve as the source of truth for all infrastructure configurations, allowing us to model our infrastructure in a way that is both user-friendly and declarative.

Vidra is a continuous deployment Kubernetes Operator created to automatically reconcile the desired state with the cluster environment based on the configuration stored in Infrahub.

Vidra was designed to be a lightweight, efficient, and reliable operator that integrates seamlessly with Infrahub and Kubernetes. We have built it to be as modular and extensible as possible, allowing for easy addition of other tools and systems as sources of truth. That is why we have chosen to open-source Vidra, so that it can be used and extended by the community. We believe that by sharing our work, we can help others build better and more efficient cloud-native GitOps workflows.

Goals

- Centralize infrastructure modeling and configuration management
- Automate the generation and deployment of Kubernetes manifests
- Continuously reconcile the desired state with the cluster environment
- Streamline infrastructure operations and improve efficiency
- Provide full traceability and auditability of deployment actions

Technology stack

- Infrahub – as the inventory system and source of truth
- Vidra, our Operator written in Go on Operator SDK – for continuous reconciliation and deployment
- Kubernetes – as the deployment and runtime platform
- KubeVirt – for managing virtual machines on Kubernetes
- flask - for a self-service web interface
- cobra - for a Vidra CLI interface

3. Management Summary

Initial Situation

In modern cloud-native environments, continuous deploymant and declarative configuration are critical for achieving scalable, consistent, and reliable operations. Popular tools such as ArgoCD and FluxCD automate the deployment of Kubernetes manifests, enabling clusters to continuously reconcile with their defined state. However, these workflows generally require teams (mostly only) to manually create and maintain YAML files for each deployment. As infrastructure grows in scale and complexity, organizations increasingly prefer to define their systems using higher-level inventory systems and domain-specific models that capture relationships, intent, dependencies, configuration details, and the current state of infrastructure components.

This shift introduces a gap between abstract infrastructure modeling and automated deployment. While higher-level, custom schemas reflect inventory, business logic and system relationships, there is often no direct bridge to generate deployable resources dynamically. Without this bridge, organizations face friction between infrastructure design and cluster operations, often relying on brittle, custom-built pipelines that are hard to maintain and do not scale well across environments. Bridging this gap by automatically generating manifests from these infrastructure models would significantly streamline operations, reduce manual technical effort, and improve scalability and ease of use for other teams.

Objective

The goal of the project was to close this gap and enable a fully automated path from infrastructure modeling to Kubernetes deployments.

That is where Infrahub and our custom Kubernetes Operator, Vidra, come into play. Infrahub serves as a centralized inventory system that allows teams to model their infrastructure declaratively in a easy to use user interface, capturing the relationships and intent of their systems in a structured way. It provides a source of truth for infrastructure configuration, enabling teams to define their desired state in a way that is both human-readable for everyone in the team and machine-processable. Vidra, our continuous deployment Kubernetes Operator, then bridges this gap by enabling fully automated deployment from infrastructure modeling to cluster deployment. Vidra continuously monitors the desired state defined in Infrahub—a centralized, version-controlled infrastructure graph—and ensures that the corresponding manifests are generated and applied to the Kubernetes cluster. This approach brings GitOps principles directly into the cluster, providing a seamless, reliable, and traceable deployment workflow without relying on external CD tools. Vidra empowers teams to focus on strategic improvements and innovation by reducing manual intervention and operational friction, by beeing able to just spinn up new infrastructure components like containers or virtual machines with a few clicks in the Infrahub UI.

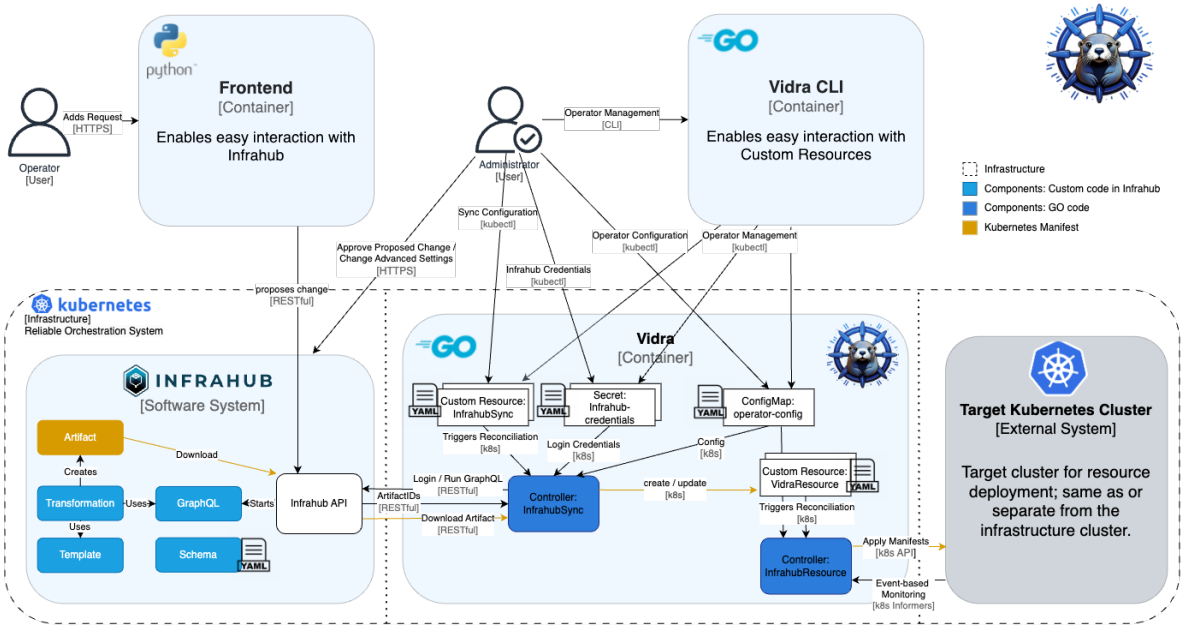

Figure 3.1.: C4 Component Diagram, showing the Infrastructure and Code components

Results

Vidra enables automated deployment of manifests from Infrahub to Kubernetes, ensuring configuration changes are promptly and consistently applied across the cluster. This reduces manual intervention and significantly improves operational reliability. By following GitOps principles, Vidra guarantees continuous reconciliation and full traceability of all deployment actions.

4. Acknowledgments

We sincerely express our deepest gratitude to our advisors, `Urs Baumann` and `Jan Untersander`, for their professional support and expert guidance throughout the course of this project. Their insightful advice and constructive feedback have been valuable and helpfull.

We are also grateful to the OpsMill - Infrahub team for their prompt and helpful support during technical challenges, as well as for the opportunity to present our work at their internal `Otto's Tech Talks`; connecting with people from around the world working on Infrahub and receiving valuable insights and discusscuss our project afterward was a pleasure.

Finally, we thank our friends and family for proofreading and providing valuable feedback.

Contents

1. Abstract i

2. Vision ii

3. Management Summary iii

4. Acknowledgments v

Contents vi

I. Technical Documentation 1

1. Overview 2

2. Requirements 3
2.1. Functional Requirements . . . 3
2.1.1. Persona . . . 3
2.1.2. User Stories . . . 3
2.2. Non-Functional Requirements . . . 5
2.2.1. Approach . . . 5

3. Preliminary Work 7

4. Initial Project Analysis 8
4.1. Flux . . . 8
4.1.1. Lab Setup . . . 8
4.1.2. Result . . . 9
4.1.3. Conclusion . . . 9
4.2. ArgoCD . . . 10
4.2.1. Hypothesis and Testing . . . 10
4.2.2. Community Feedback . . . 10
4.2.3. Solution and Conclusion . . . 11
4.3. Kubernetes Operator . . . 12

5. Architecture 13
5.1. Introduction and Goals . . . 13
5.2. Context Diagram (Level 1) . . . 14
5.3. Container Diagram (Level 2) . . . 15
5.4. Component Diagram (Level 3) . . . 16
5.5. Architectural Decision Records . . . 17

6. Quality Measures 22
6.1. Organizational Measures . . . 22
6.2. Guidelines . . . 22
6.3. Tools Used to Assess Product Quality in CI/CD . . . 22
6.4. Manual Testing . . . 23
6.4.1. User Tests . . . 23

6.5. Code Review . . . 23

7. Infrahub **24**
7.1. Schema . . . 24
7.1.1. UML Diagram . . . 24
7.1.2. Generics - Resource . . . 25
7.1.3. Nodes . . . 26
7.2. Python Transformation . . . 28
7.2.1. Transform Function . . . 29
7.2.2. Helper Functions . . . 30
7.2.3. YAML Templates . . . 33
7.3. Graph QL Queries . . . 34
7.4. .infrahub.yml . . . 35
7.5. Resource Manager . . . 36
7.6. Users, Groups, Roles, and Permissions . . . 37
7.6.1. Users . . . 37
7.6.2. Roles . . . 38
7.6.3. Permissions . . . 38
7.6.4. User Groups . . . 38
7.7. Service Groups . . . 39
7.8. Git Integration . . . 39
7.9. Object Templates . . . 40

8. Self-service Frontend **41**
8.1. Architecture . . . 41
8.2. HTML Interface . . . 42
8.3. Flask . . . 42
8.3.1. Python Libraries . . . 42
8.3.2. index function . . . 43
8.3.3. Python Script . . . 43
8.4. Python - CreateObjects . . . 44
8.4.1. Python Script . . . 44
8.5. Python - HelperFunction . . . 45

9. Technical Issues and Obstacles **47**
9.1. Infrahub . . . 47
9.1.1. Resource Manager via Template . . . 47
9.1.2. Web Browser API Calls . . . 47
9.1.3. General GraphQL File . . . 48
9.1.4. Calling YAML File . . . 48
9.2. Vidra Operator . . . 49
9.3. General Issues . . . 49

II. Project Documentation **50**

1. Results **51**

2. Conclusion **52**

3. Project Planning **53**
3.1. Processes . . . 53
3.2. Architectural Roles . . . 53
3.3. Meetings . . . 53
3.4. Phases . . . 53
3.4.1. Time Table . . . 54
3.5. Risk Management . . . 55
3.5.1. Risks . . . 55
3.5.2. Risk Countermeasures . . . 55
3.5.3. Risk Matrix . . . 56
3.5.4. Risk summary . . . 57
3.6. Planning Tools . . . 57
3.6.1. JIRA . . . 57
3.6.2. Clockify . . . 57
3.6.3. Overleaf . . . 57

List of Tables **58**

List of Figures **59**

Acronyms **60**

Glossary **61**

III. Appendix **62**

1. Quality Attribute Scenarios **63**

2. Detailed Architecture Decisions **74**
2.1. What is Infrahub used for? . . . 74
2.2. Do we need Profiles or Templates in Infrahub? . . . 77

3. Detailed HTML and CSS Code **79**
3.1. HTML . . . 79
3.2. CSS Styling . . . 80

Bibliography **82**

Part I.

Technical Documentation

1. Overview

This project is composed of several integrated components designed to streamline and simplify the creation and management of Kubernetes resources. The primary components developed or enhanced with custom automations include:

- **Frontend Application:** A web-based user self-service interface that allows users (who do not need to manage resources) to easily create resources in Infrahub.
- **Infrahub:** An inventory system that we enhanced to manage the lifecycle of Kubernetes resources. It provides a central repository for modeling and managing resources, as well as representing our desired state.
- **Vidra:** A continuous deployment Kubernetes operator that automates the provisioning and management of Kubernetes resources, ensuring they always reflect the desired state defined in Infrahub.
- **Vidra CLI:** A tool that makes it easy to interact with Vidra. It provides a command-line interface for configuring Vidra and managing its resources.

In this document, we focus on the Infrahub side of the project and how it integrates with the other components. The Infrahub side of the project can ultimately be populated with different resources and use cases. We showcase the use cases of creating app deployments and virtual machines in the following chapters.

Since we decided to open-source Vidra, we have chosen to publish the technical documentation of Vidra alongside it on GitHub Pages. This approach ensures broader accessibility, and we hope it encourages community contributions.

Vidra Documentation

2. Requirements

This chapter outlines the requirements of this project, structured into two main categories: functional and non-functional requirements.

At the beginning, we analyzed the stakeholders involved to determine for whom the project is being built. For a detailed discussion of the stakeholder analysis and our decision, please see the resulting architectural decision in section 2.1.

2.1. Functional Requirements

Functional requirements specify the core capabilities and behaviors the system must provide to meet stakeholder needs. These requirements are primarily derived from the perspectives and objectives of two key personas: (1) the Admin (Kubernetes Cluster Administrator / Cloud Engineer), who is responsible for developing, maintaining, and managing Kubernetes-based solutions and templates; and (2) the User (Developer / Operator), who interacts with the system to deploy and manage resources without deep Kubernetes expertise. The functional requirements are articulated as user stories, ensuring that the system design and implementation are closely aligned with real-world usage scenarios and stakeholder expectations. Collectively, these requirements establish a comprehensive foundation for the project and guide the subsequent development process.

2.1.1. Persona

- **Admin (Kubernetes Cluster Administrator / Cloud Engineer):** This persona represents a technically proficient individual responsible for the setup, configuration, and ongoing management of the Kubernetes cluster. The Admin designs, implements, and maintains deployment templates (e.g., for virtual machines), ensures the security and reliability of the cluster, manages resource allocation, and oversees all Kubernetes resources.
- **User (Developer / Operator):** This persona represents an individual with limited or no expertise in Kubernetes who requires the ability to deploy virtual machines or other resources on the cluster. The User seeks a simplified interface and workflow that abstracts away the complexities of Kubernetes, enabling them to accomplish their tasks efficiently without deep technical knowledge of the underlying infrastructure.

2.1.2. User Stories

In this section, we outline the user stories from each persona's perspective. They are categorized into two scopes: (1) In Scope – user stories that will be addressed by this project; (2) Out of Scope – user stories not immediately required but potentially implementable given sufficient time or in future work.

In Scope

- **U-1:** As a User, I want a simple and easy-to-use interface to self-deploy a virtual machine or other Kubernetes resources on the Kubernetes cluster.
- **U-2:** As a User, I want to be able to deploy a virtual machine or other resources on the Kubernetes cluster without having to know Kubernetes.

- **U-3:** As a User, I want to select from pre-configured templates for common deployments to save time and reduce complexity.
- **U-4:** As a User, I want to get access to the virtual machine or other resources I deploy on the Kubernetes cluster.
- **U-5:** As an Admin, I want to define and manage what resources can be deployed by the User on the Kubernetes cluster.
- **U-6:** As an Admin, I want to define templates for common deployments that can be used by the User.
- **U-7:** As an Admin, I want to specify basic resource limits (such as CPU and memory) for my virtual machines without dealing with low-level Kubernetes settings.
- **U-8:** As an Admin, I want to be able to delete the virtual machine or other resources easily via infrahub.
- **U-9:** As an Admin, I want to be able to determine which image (e.g., Docker image) is used for the virtual machine or other resources deployed on the Kubernetes cluster.
- **U-10:** As an Admin, I want the tool to reconcile the state of the cluster with the state of infrahub automatically.
- **U-11:** As an Admin, I want to see errors easily when deployments fail or exceed resource limits, to quickly respond to issues.
- **U-12:** As an Admin, I want to be able to steer which artifact in infrahub should be used for the resources deployed on the Kubernetes cluster.
- **U-13:** As an Admin, I want to be able to roll back deployments to a previous state in case of issues or failures, by syncing to an older artifact.
- **U-14:** As an Admin, I want the tool to follow the GitOps principles[1], ensuring that the state of the Kubernetes cluster is always in sync with the state defined in infrahub. Even if a User changes the resource manually in the Kubernetes cluster, the tool should detect this and overwrite the change again.

Out of Scope

- **U-15:** As a User, I want to see the status of my virtual machine deployment (e.g., pending, running, failed) and receive feedback on its success or failure.
- **U-16:** As a User, I want to be able to deploy multiple virtual machines or other resources at once.
- **U-17:** As an Admin, I want to deploy resources to multiple Kubernetes clusters to support different environments (e.g., development, staging, production).
- **U-18:** As an Admin, I want to enforce resource quotas and limits for different users to prevent resource exhaustion.
- **U-19:** As an Admin, I want to audit and track user activity related to virtual machine deployments for security and compliance.

[1] https://opengitops.dev

- **U-20:** As an Admin, I want the tool to immediately detect and correct configuration drift between the desired state in infrahub and the actual state in the Kubernetes cluster. (Event-based reconciliation)
- **U-21:** As an Admin, I want to automate the cleanup of unused or expired virtual machines to optimize resource utilization.
- **U-22:** As an Admin, I want to roll back faulty deployments quickly to maintain system stability.

2.2. Non-Functional Requirements

2.2.1. Approach

The non-functional requirements were identified through collaborative discussions within the project team and with guidance from our advisor. These requirements are formulated to be SMART (Specific, Measurable, Achievable, Relevant, and Time-bound), ensuring clarity and traceability throughout the project. For systematic classification, we adopt the ISO/IEC 25010 quality model[2], which provides a comprehensive framework for software quality. Furthermore, we specify the non-functional requirements using Quality Attribute Scenarios (QAS) to make them actionable and verifiable.

Figure 2.1.: Non-Functional requirements

- **Performance Efficiency**: The system should optimize resource usage on the Kubernetes cluster and in the network. It should be lightweight and efficient, only downloading the necessary resources from infrahub.
- **Compatibility**: The system should be compatible with existing Kubernetes tools and technologies, allowing for seamless integration and interaction with other cloud-native solutions.
- **Usability**: The system should be user-friendly, providing an intuitive interface for Users to deploy resources without needing deep Kubernetes knowledge. It should also provide clear error messages and guidance for troubleshooting.
- **Reliability**:
  - **Maturity**: The system should be stable and reliable, with a low rate of defects and issues. It should be able to handle a high volume of deployments without significant performance degradation.

[2]ISO/IEC 25010 model https://iso25000.com/index.php/en/iso-25000-standards/iso-25010

  - **Fault Tolerance**: The system should be able to recover from failures and continue operating without data loss or significant disruption.
  - **Scalability**: The system should be able to handle an increasing number of deployments and users without significant performance degradation.

- **Maintainability**:
  - **Modularity**: The system should be modular, allowing for future enhancements and modifications without significant rework.
  - **Testability**: The system should be designed to facilitate unit testing, allowing for easy verification of functionality and performance.
  - **Reusability**: The system should allow for the reuse of components and templates, enabling efficient development and deployment of resources.

- **Portability**:
  - **Adaptability**: The system should be adaptable to different Kubernetes environments and configurations, allowing for easy deployment in diverse scenarios.
  - **Installability**: The system should be easy to install and configure, with clear documentation and minimal dependencies.

Out of Scope

- **Security**: Access to the system should be restricted based on user roles and permissions. All other aspects of security are considered out of scope.

The detailed Quality Attribute Scenarios that specify the non-functional requirements are provided in chapter 1 in the appendix. Not all non-functional requirements can be fully validated through comprehensive testing—particularly those related to performance—within the scope of this project. Nevertheless, we will design the system with these requirements in mind, to the point where we can confidently say they are fulfilled. Additionally we are aiming to address as much of them as possible through unit and integration tests. We believe it is preferable to document and consider these performance requirements throughout development, rather than limiting the scope of non-functional requirements and risking their omission.

3. Preliminary Work

Several open-source projects, such as `Infrahub` and `Kubernetes`, are widely used in the industry. Kubernetes operators and controller frameworks, including Kubebuilder, are mature and well-documented, and there are well-known CD operators such as ArgoCD and Flux. However, to date, there has been no direct integration between Infrahub and Kubernetes. Existing approaches, if any, must rely on intermediary tools like Git, which cannot achieve the same level of seamless integration and automation as a native Kubernetes operator.

4. Initial Project Analysis

In this chapter, we look into FluxCD, ArgoCD, and Kubernetes Operators as continuous deployment solutions to retrieve manifests from Infrahub and automatically deploy them to Kubernetes.

4.1. Flux

Flux is a GitOps toolset for Kubernetes that automates the deployment of configuration files (YAML) stored in Git repositories, S3 buckets, or other OCI-compatible registries. It relies on two types of repositories: a source repository, which stores the YAML files and configurations (such as Helm charts and Kustomize files), and a target repository that manages the deployment configurations.

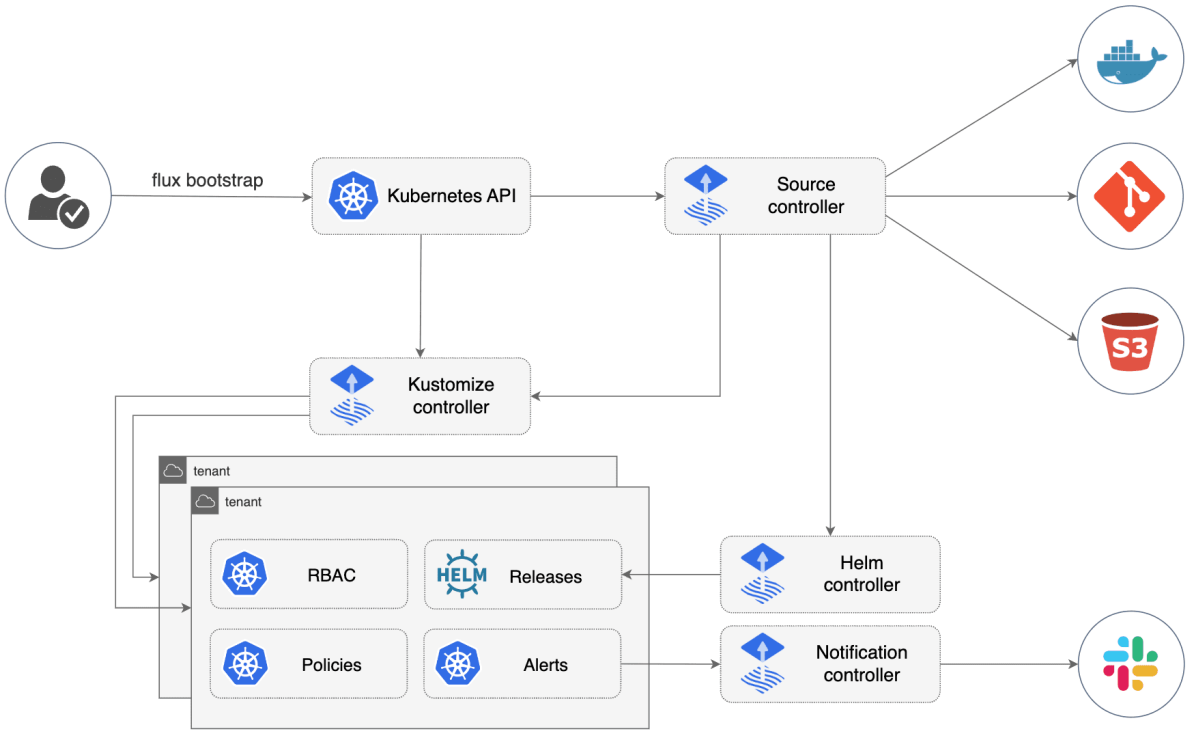

Figure 4.1.: Diagram of the Flux Workflow
[1]

4.1.1. Lab Setup

In our test setup, we utilized one GitLab repository as a bootstrap file and an S3 Bucket from MinIO, which stores the artifacts from Infrahub. Our goal was to test automated deployment from an Infrahub artifact via Flux to the Kubernetes cluster.

[1] Source: FluxCD Documentation, August 24, 2023, https://fluxcd.io/flux/components/

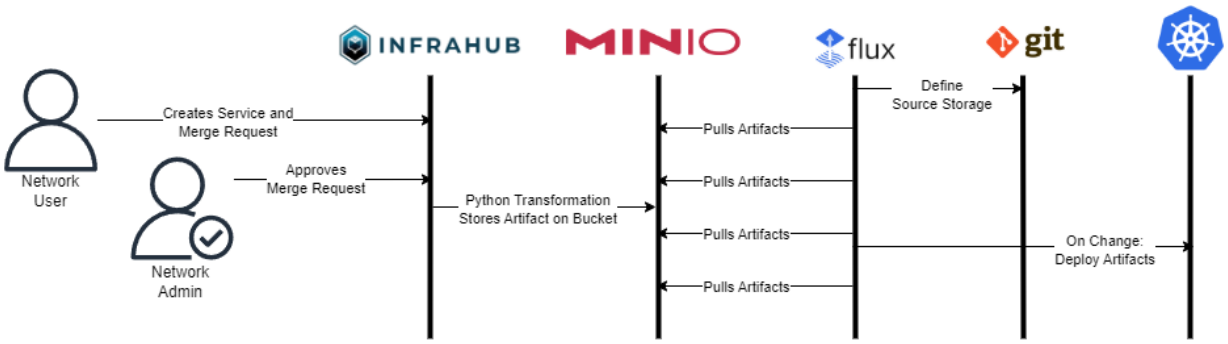

Figure 4.2.: Description Workflow Setup Flux Testing

4.1.2. Result

In our test scenario, we wanted to create a Kubernetes manifest in Infrahub, which would be automatically fetched by Flux and deployed on Kubernetes.

Advantages

- Can use an S3 Bucket as a source.
- It's mature and CNCF graduated.
- There is a large community.

Disadvantages

- Needs its own plugin written in Go to fit our use case.
- Needs a kustomization.yaml or Chart.yaml in the repository, which Infrahub doesn't provide.
- Doesn't recognize YAML files that don't have a .yml extension. Infrahub doesn't put any extensions on its artifacts.

4.1.3. Conclusion

Flux is easy to deploy and use, making it an excellent tool for automating Kubernetes deployments. The integration with Kustomize for managing YAML files is also straightforward and efficient. Using separate repositories can even be an advantage, as it allows an S3 bucket to be used as a source.

However, a major drawback of Flux is its limited flexibility in defining the source storage. In our use case, we need to specify exactly which file should be used, especially since it changes after every merge request. Infrahub, for instance, stores its artifacts using a storage-ID that updates with each deployment, while requiring a Kustomization or HelmChart file named kustomization.yaml or Chart.yaml. To accommodate this, we need a custom script or plugin that dynamically generates and updates this file whenever changes occur. To address this limitation, we would need to develop a custom plugin that extends the source storage definition to meet our specific requirements.

4.2. ArgoCD

ArgoCD is a widely used GitOps tool for automating Kubernetes deployments. Like Flux, it uses a Git repository as the source of truth for configuration files, but distinguishes itself with a user-friendly interface and advanced features for application management and monitoring.

Integration with Infrahub: ArgoCD's capability to integrate with Infrahub was tested using the Config Management Plugin (CMP)[2]. The CMP allows ArgoCD to use custom scripts or tools—written in any language—to generate Kubernetes manifests from external sources. Once configured, the plugin enables ArgoCD to fetch and render manifests from Infrahub or other systems, extending beyond native Git, Helm, or OCI sources.

4.2.1. Hypothesis and Testing

We hypothesized that a custom CMP could directly pull Infrahub artifacts, for example, using a Python script with the Infrahub SDK or API.

In testing, we implemented a simple CMP that generated a static ConfigMap (see Figure 4.3). The CMP successfully rendered Kubernetes resources from any valid input, but it does not trigger ArgoCD syncs—updates only occur when a new commit is detected in the Git repository.

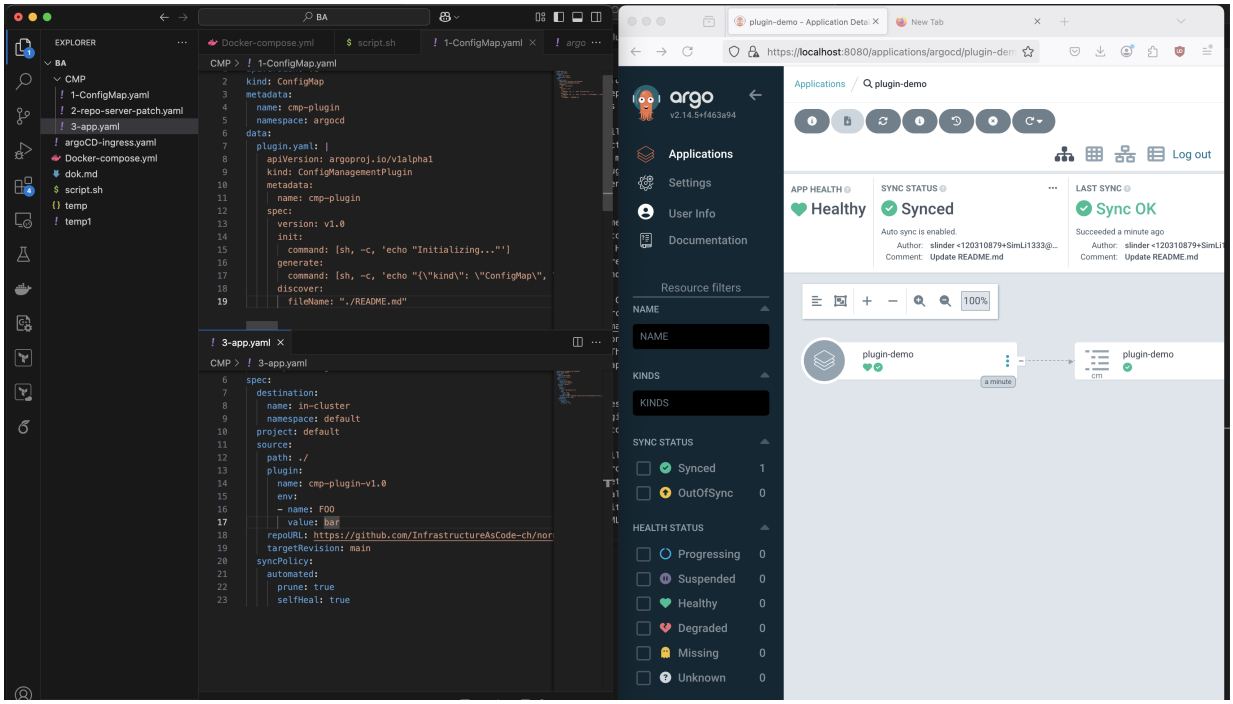

Figure 4.3.: CMP Test: ConfigMap Deployment

4.2.2. Community Feedback

To clarify integration options, we consulted the ArgoCD and Infrahub communities. An ArgoCD developer explained that ArgoCD is designed to sync only with Git repositories to maintain focus: "ArgoCD intentionally avoids supporting a wide variety of sources. We risk doing a lot of stuff poorly and nothing extremely well."[1]

[2] ArgoCD Config Management Plugin: March 21, 2024 https://argo-cd.readthedocs.io/en/stable/operator-manual/config-management-plugins/

[1] Slack.com, “Discussion on ArgoCD Design,” Slack channel: https://app.slack.com/client/T08PSQ7BQ/C01TSERG0KZ, Accessed on March 25, 2024.

Infrahub developers recommended a hybrid approach using ArgoCD, Infrahub, and Git. They suggested pulling artifacts from Git via pipelines or using a custom script/operator to fetch artifacts from the Infrahub API. "Committing the artifact back to a git repository makes sense, it has come up a few times but this is not something Infrahub can do today. Some customers are managing that outside of Infrahub with a GH Action like a script that will pull the artifacts from Infrahub and commit them to git."[2]

[2] Discord.com, “Discussion on Infrahub Integration with Kubernetes,” Discord channel: https://discord.com/channels/1212332642801025064/1301914405176475729, Accessed on March 25, 2024.

4.2.3. Solution and Conclusion

After considering all options, we concluded that a combination of ArgoCD, Infrahub, and Git would be a valid solution in two scenarios. The first scenario could be that the YAML files are generated using a Python-based transformer in Infrahub, and an artifact ID is committed to Git to trigger the CMP. This is illustrated in Figure 4.4.

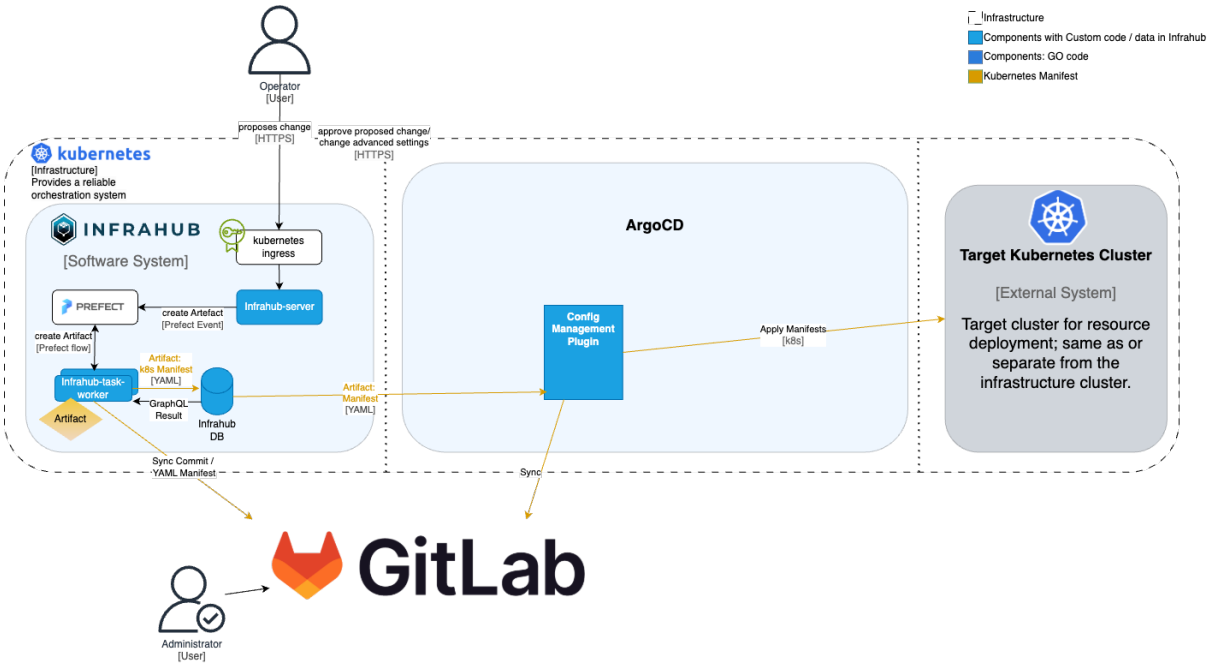

Figure 4.4.: ArgoCD and Infrahub Integration Workflow

Alternatively, the artifact could be pushed or pulled to the Git repository, allowing ArgoCD (or Flux) to sync as usual (second scenario).

Another option is to use Infrahub only to manage key values and generate a Helm ‘values.yaml‘ file for Git.

However, both scenarios present drawbacks, as they require maintaining Infrahub in parallel with Git. This approach effectively turns Infrahub into a graphical interface for managing Helm or YAML values within a traditional GitOps workflow, undermining the objective of having a single, unified source of truth.

4.3. Kubernetes Operator

A custom Operator is the most advanced and customizable solution for integrating Infrahub with Kubernetes. Operators are Kubernetes controllers that automate the management of complex applications and services. They extend the Kubernetes API to create, configure, and manage custom resources. Operators use the Kubernetes client library to interact with the Kubernetes API.

Integration with Infrahub: We could write a custom Operator that pulls the Infrahub artifact and creates the necessary Kubernetes resources. The Operator would watch for changes in the Infrahub artifact and automatically update the Kubernetes resources accordingly.

We would need to identify the Infrahub artifacts that are present in Infrahub and need to be deployed. For that, we could use the Infrahub API to execute a predefined GraphQL query to list all artifact IDs and their hashes of a particular branch and time, as seen in Figure 4.5. With that information, we could pull the artifact from the Infrahub API using `/api/artifact/{artifact_id}`.

Figure 4.5.: Infrahub API - Post GraphQL query

The big advantage of this solution is that we can tailor the Operator to our specific needs. We can define the logic for pulling the Infrahub artifact, parsing the YAML files, and simply applying them to the Kubernetes cluster.

But writing a custom Operator is a complex and time-consuming task. We would need to learn the Go programming language and the Kubernetes client library, as well as the Operator Framework. Additionally, a custom Operator is not as established, proven, and tested as ArgoCD or Flux, and we might need to solve the same problems that ArgoCD and Flux already master perfectly.

5. Architecture

This chapter provides an overview of the high-level architecture for the integrated solution, encompassing all four software containers. For an in-depth, code- and component-level (C4) description of the Vidra and Vidra CLI architecture, refer to the Vidra documentation and Vidra CLI documentation. Component and code-level details about the Infrahub Frontend architecture can be found in Section 8.1.

5.1. Introduction and Goals

The main goal is to demonstrate how integrating Infrahub with Kubernetes simplifies and automates application and infrastructure deployment. Infrahub serves as a central management platform, while Vidra, a Kubernetes operator, synchronizes infrastructure artifacts (e.g., manifests, configurations) from Infrahub to Kubernetes.

Continuous Deployment Workflow: Changes in Infrahub are automatically propagated to the Kubernetes cluster, ensuring the actual state matches the desired state.

The architecture aims to:

- Provide a user-friendly interface for Kubernetes application management via Infrahub.
- Automate synchronization of infrastructure definitions from Infrahub to Kubernetes using Vidra.
- Support continuous deployment by applying Infrahub changes directly to the cluster.
- Enable use of reusable templates for streamlined deployments.
- Offer a scalable and adaptable architecture for diverse requirements.

5.2. Context Diagram (Level 1)

The context diagram provides a high-level overview of the system and its interactions with external entities, such as users and other systems.

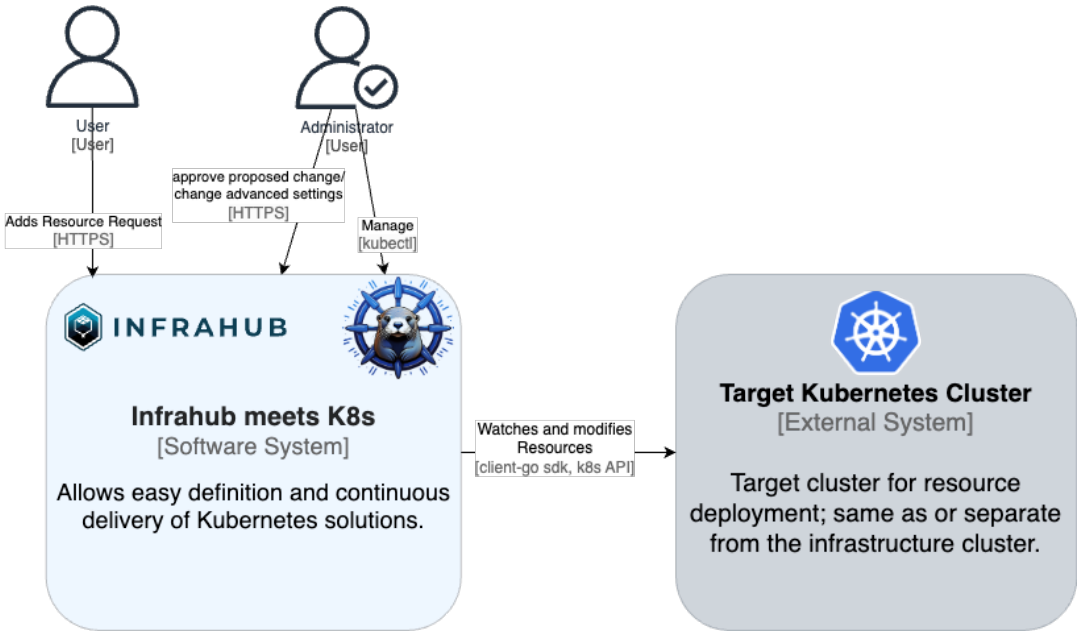

Figure 5.1.: C4 Context Diagram

The diagram illustrates:

- **Infrahub and Vidra:** The core elements of the software system, where Infrahub serves as the management interface and Vidra acts as the Kubernetes operator, working together in harmony to synchronize infrastructure definitions with the real running state.
- **Target Cluster:** The target environment where applications are deployed. It can be the same cluster as the one where Infrahub and Vidra are running, or a separate cluster.
- **Users:** Individuals interacting with Infrahub to manage deployments and configurations. They typically have some understanding of the infrastructure and deployment processes.
- **Administrator:** The individual responsible for managing the Kubernetes cluster, ensuring its health, security, and availability, and providing the necessary resources for application deployment.

5.3. Container Diagram (Level 2)

The container diagram breaks down the system into its main containers, illustrating how each part collaborates to deliver overall functionality.

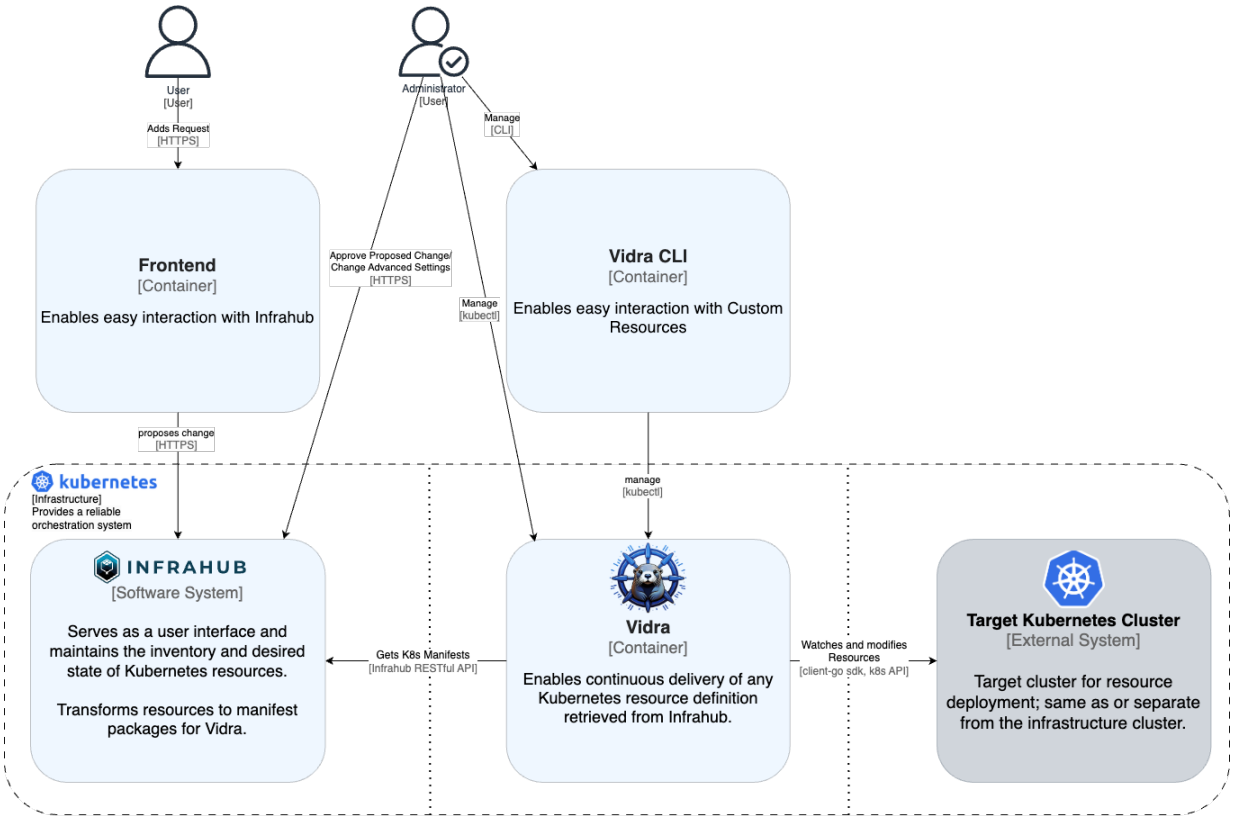

Figure 5.2.: C4 Container Diagram

The diagram illustrates the following containers:

- **Frontend:** The self-service interface for users with no prior knowledge to interact with the system, providing a user-friendly GUI for creating requests for Kubernetes resources.
- **Infrahub:** The inventory software system with a GUI for users and administrators to manage Kubernetes resources, providing a central management platform for an abstract representation of the desired state of the Kubernetes cluster.
- **Vidra Operator:** The Kubernetes operator responsible for synchronizing infrastructure definitions from Infrahub to the target cluster.
- **Vidra CLI:** A command-line interface for interacting with Vidra, allowing administrators to manage Vidra configuration and handle continuous deployment directly from the terminal.

5.4. Component Diagram (Level 3)

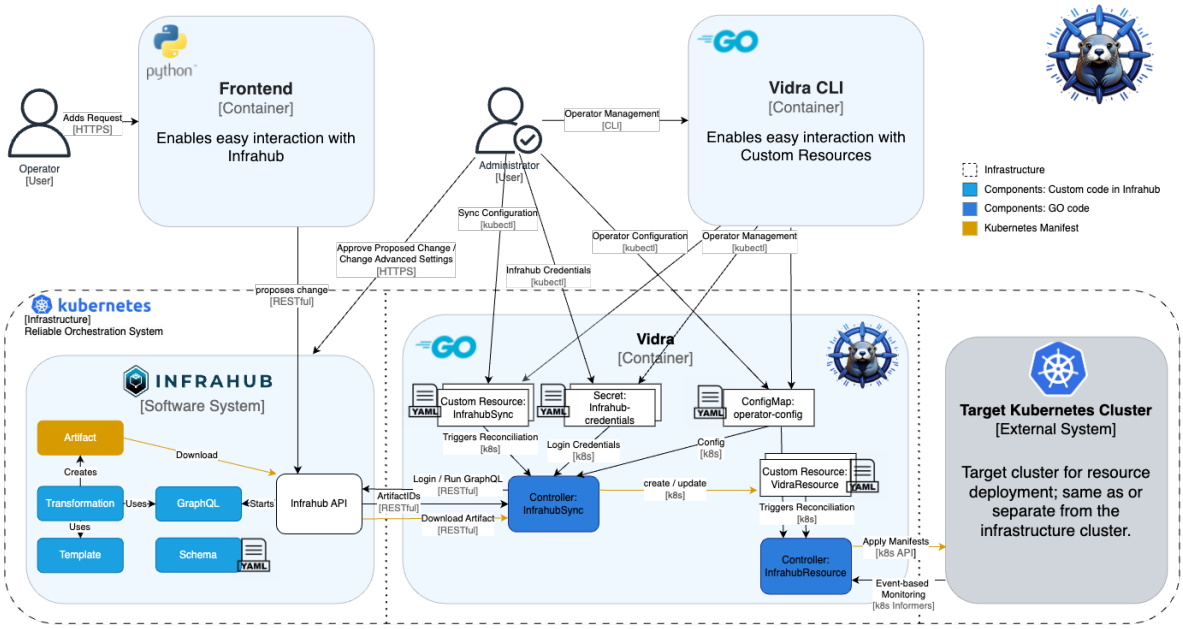

Figure 5.3.: C4 Component Diagram

The component diagram provides a detailed view of the components within the Infrahub and Vidra system, illustrating how they interact to deliver functionality. It includes:

- **Infrahub API:** The API that allows interaction with Infrahub, enabling users to run GraphQL queries and download artifacts containing Kubernetes manifests.
- **GraphQL Queries:** Used to interact with Infrahub's Neo4j database, enabling retrieval, transformation, and configuration of data as required for managing and deploying Kubernetes resources. GraphQL is used because it is the intended way for automations by Infrahub.
- **Transformations:** Python code that generates structured data artifacts from data it gathers using GraphQL queries and, in our case, YAML templates.
- **Artifact:** The structured data artifacts generated by the transformations, which contain Kubernetes manifests.
- **Templates:** YAML templates used to generate Kubernetes manifests, allowing for reusable and customizable configurations.
- **Vidra Operator:** Detailed information about the Vidra Operator can be found in the Vidra Operator Documentation - Section Data Flow and Interaction.

All components that we added to the software system Infrahub reside in a Git repository, which is synced with Infrahub using the *Infrahub Git Sync*[1] feature. This allows us to version control our changes and collaborate effectively on the Infrahub data model.

More detailed information about the supporting components "Frontend" and "Vidra CLI" can be found in Section 8.1 and Vidra CLI Documentation.

[1] Infrahub Documentation Concept: 12.6.2025 https://docs.infrahub.app/getting-started/concepts#integration-with-git

5.5. Architectural Decision Records

During the prototype setup to test components and their interactions, we made key decisions influencing the overall architecture. The decisions influencing Infrahub and its use cases are documented in the following sections. The decisions influencing the Vidra Operator are documented in the Vidra Operator Documentation - Section Decisions.

Choosing a Kubernetes Operator over ArgoCD CMP or Extending Flux

Context and Problem Statement

We needed a solution to automatically deploy Kubernetes manifests generated by Infrahub to a Kubernetes cluster. The main options considered were: (1) using ArgoCD with a Config Management Plugin (CMP), (2) extending Flux with a custom plugin, or (3) developing a custom Kubernetes Operator. As this was a major decision, we conducted a detailed analysis of the options and their implications.

Considered Options

- **ArgoCD with CMP:** Use ArgoCD's Config Management Plugin to fetch and render manifests from Infrahub artifacts.
- **Flux with Custom Plugin:** Extend Flux to support Infrahub artifacts as a source, possibly via S3 or custom scripts.
- **Custom Kubernetes Operator:** Develop a dedicated operator that pulls artifacts directly from Infrahub and applies them to the cluster.

Decision Outcome

Chosen option: Custom Kubernetes Operator. The decision was made to implement a dedicated operator for the following reasons:

- **Direct Integration:** The operator can interact directly with the Infrahub API, eliminating the need for intermediate Git repositories or additional transformation steps.
- **Full Flexibility:** The operator can be tailored to our exact requirements, including artifact selection, authentication, and custom deployment logic.
- **Reduced Complexity:** Avoids maintaining parallel Git repositories or writing/maintaining custom plugins for third-party tools.
- **Single Source of Truth:** Keeps Infrahub as the authoritative source for infrastructure state, rather than duplicating state in Git.

Consequences

- **Good:** The solution is purpose-built for our workflow, providing maximum flexibility and direct integration with Infrahub.
- **Good:** It enables real-time synchronization and reduces operational overhead.
- **Bad:** Developing and maintaining a custom operator requires significant engineering effort and expertise in Go and Kubernetes APIs.

- **Bad:** The operator lacks the maturity, ecosystem, and community support of established tools like ArgoCD and Flux.

A detailed analysis of the considered options can be found in Chapter 4.

Split of Work

Context and Problem Statement

To efficiently progress with the project, we needed to allocate responsibilities based on each teammate's strengths and interests, while ensuring collaborative decision-making and shared code ownership.

Considered Options

- **Divide work by technical expertise and interest:** Each teammate focuses on areas where they have the most experience or motivation, but all major decisions and code reviews are done together.
- **Divide work by feature or component:** Assign specific features or components to each teammate, with less overlap and potentially less shared understanding.
- **Work together on all tasks:** Both teammates work on all aspects together, which may slow progress but maximizes shared knowledge.

Decision Outcome

Chosen option: Divide work by technical expertise and interest. Simon took responsibility for developing the operator, leveraging his experience with Go and interest in the challenge. Ramon focused on adapting Infrahub to fit our use cases, including generating YAML manifests and creating an easy-to-use frontend. Despite this division, all major decisions were made collaboratively, and we conducted code reviews of each other's work to maintain shared understanding and code quality.

Consequences

- **Good:** Efficient use of individual strengths, faster progress, and maintained high code quality through collaboration and reviews.
- **Bad:** Potential for knowledge silos, mitigated by regular code reviews and shared decision-making.

Infrahub User Focus and Resource Modeling

Context and Problem Statement

We needed to decide whether Infrahub should primarily serve administrators (with granular Kubernetes control) or developers/users (with simplified deployment workflows). The goal was to balance user-friendliness with the need for advanced Kubernetes management capabilities. This decision majorly impacts how resources are modeled and presented in Infrahub, and whether Infrahub creates one artifact for each Kubernetes kind or if the whole solution (e.g., Kubernetes manifest) lives in one artifact.

Considered Options

- **Administrator-focused model:** Full Kubernetes kind mapping, high flexibility in Infrahub, as the user can pick and choose the resources they need for their custom solution, but complex as all Kubernetes Kinds need to be present in Infrahub and almost redundant with existing tools like the Kubernetes Dashboard. For this approach, it would make the most sense if an artifact is created for each Kubernetes Kind.
- **Developer/user-focused model:** Simplified interface with predefined templates, reducing complexity for non-experts. For this approach, we would create one artifact for the whole solution, like a webserver manifest, which includes all necessary resources such as Deployments, Services, and Ingresses. This approach focuses on deployment rather than detailed resource management, allowing users to deploy applications with personalized configuration.

Decision Outcome

Chosen option: Developer/user-focused model with predefined templates because:

- It simplifies the user experience, allowing users to deploy applications without deep Kubernetes knowledge.
- It reduces management complexity and data volume in Infrahub, as users interact with a limited set of templates rather than individual Kubernetes resources.
- It avoids duplicating the Kubernetes Dashboard, focusing on deployment rather than detailed resource management.
- It allows for fast, consistent, and safe deployments using templates, which can be easily maintained by administrators.
- It enables users to deploy applications with minimal configuration, using a fixed set of exposed parameters in templates.
- It provides a scalable solution that can adapt to various user needs without overwhelming them with complexity.

Consequences

- **Good:** Users have limited customization; only exposed parameters can be changed, making the interface simpler and less error-prone.
- **Good:** Administrators can maintain and provide templates for common use cases, ensuring consistency and best practices.
- **Bad:** Advanced configurations require administrator intervention, which may slow down complex or custom deployments.
- **Bad:** The Infrahub artifact will represent a deployment template rather than individual Kubernetes resources, potentially resulting in larger artifacts and less granular control.

As it was quite a big decision, a detailed decision document was created, which can be found in Section 2.1.

Use of Templates for Application Deployment

Context and Problem Statement

We needed to determine how to enable users to deploy applications to Kubernetes clusters via Infrahub in a way that balances **flexibility**, **usability**, and **maintainability**. Two options were considered: using *Profiles*, which share attribute values across multiple objects, and using *Templates*, which are reusable blueprints with default values.

Considered Options

- **Profiles[2]:** Share attribute values across multiple objects.
- **Templates[3]:** Reusable object templates that provide predefined defaults, enabling streamlined application deployment.

Decision Outcome

Chosen option: Templates for application deployment, because:

- Templates simplify the deployment process for users by providing sensible defaults and hiding unnecessary complexity.
- They support the definition of every attribute without limitation.
- Templates are easier to maintain and test over time.

Consequences

- **Good:** Users can deploy applications quickly and safely without deep Kubernetes knowledge.
- **Good:** Administrators can curate and update templates to reflect organizational standards.
- **Bad:** Administrators must manually update the default value for each object if changes are needed.

A detailed decision document was created, which can be found in the Appendix Section 2.2.

Creation of a Frontend for Infrahub

Context and Problem Statement

As we wanted to provide a user-friendly interface for requesting new infrastructure to interact with Infrahub, we needed to decide whether to create a custom frontend or build an easy-to-use Infrahub section. There are several steps that need to be completed before a new Kubernetes resource actually creates a correct artifact, such as adding the resource to the correct group and re-generating the artifact. The artifact is automatically created if a proposed change is merged, but then a user needs to know how to create a proposed change in Infrahub and the main branch needs to be protected. This is not user-friendly and requires knowledge about Infrahub, which we wanted to avoid for users with no prior knowledge about the solution.

[2]Infrahub Documentation Profiles: 8.6.2025 https://docs.infrahub.app/topics/profiles
[3]Infrahub Documentation Templates: 8.6.2025 https://docs.infrahub.app/topics/object-template

Considered Options

- **Custom Frontend:** Develop a custom frontend application tailored to the specific needs of users with no knowledge about the solution (self-service portal). This comes with the benefit of being able to automate the creation of new resources and relationships in Infrahub with one click, but requires additional development and maintenance effort.
- **Infrahub Frontend:** Use the existing Infrahub frontend, which provides a user interface for managing Kubernetes resources, but documentation is needed to explain the steps of creating a request for Kubernetes resources. This option leverages the existing Infrahub infrastructure and provides a consistent user experience, but may not be as tailored to the specific needs.

Decision Outcome

Chosen option: Custom Frontend, because:

- It provides an easy-to-use self-service interface for users to interact with Infrahub.
- It will create all resources and relationships in Infrahub, so users do not need to know how to add new resources to the correct group and create proposed changes.
- It allows the creation of a branch for each request, enabling administrators to review and approve changes before they are applied to the cluster.
- We can control which values are exposed to users, ensuring they can only change parameters that are safe and necessary for their deployments.
- We can create more advanced values in Infrahub so an administrator can change the values in Infrahub and the users will not see them.

Consequences

- **Good:** Users can deploy applications without needing to understand the underlying Kubernetes resources.
- **Good:** Administrators can manage and maintain the resource requests and created resources in Infrahub.
- **Bad:** Users may have limited control over advanced configurations, which could require administrator assistance for complex deployments.
- **Bad:** The frontend may not cover all use cases, requiring additional work or development for specific needs.

6. Quality Measures

Some quality measures, such as coding conventions and continuous integration practices, are documented in the publicly accessible Vidra documentation. These guidelines support open source contributions and ensure transparency. Additional quality measures specific to this bachelor thesis are detailed in this document, providing internal standards.

6.1. Organizational Measures

Merge Requests enforce code review and the Four-Eyes Principle, ensuring collaboration, early issue detection, and main branch stability.

Definition of Done A task is done when:

1. Code is committed and pushed.
2. Code is reviewed and passes CI/CD.
3. Documentation is updated.
4. Acceptance criteria are met.

6.2. Guidelines

Python - PEP8: We follow PEP8 for consistent, readable, and maintainable Python code.

Four-Eyes Principle: All changes require review and approval by another team member to ensure quality and adherence to standards.

6.3. Tools Used to Assess Product Quality in CI/CD

Our continuous integration (CI) processes are powered by GitLab CI/CD, ensuring efficient and reliable product builds, tests, and releases. This automation streamlines our development workflow, maintaining high standards of quality and consistency across all code repositories.

We employ a unified CI/CD pipeline logic for all repositories, utilizing GitLab CI/CD or GitHub Actions. This standardization ensures that every repository adheres to the same rule set. Our Python CI pipeline includes:

A shared before script which installs Poetry and the necessary development dependencies.

Ruff is used to ensure that the code consistently conforms to the PEP8 standard[1]. Ruff is configured in the `pyproject.toml` to enforce coding style guidelines, detect potential issues, and maintain uniformity.

MyPy is utilized as a type checker to ensure type correctness throughout our Python codebase. By enforcing static type checking, MyPy helps detect type-related errors early in the development process, improving code reliability and reducing runtime issues. We incorporate type hints to enhance code readability and comprehension.

[1] PEP8 guidelines: December 11, 2024 https://peps.python.org/pep-0008/

YAMLlint is used to validate the syntax and formatting of YAML files, which are used for creating the Kubernetes manifests. YAMLlint helps catch common errors such as indentation issues, duplicate keys, or malformed syntax, ensuring configuration reliability and preventing deployment failures.

Bandit is employed to identify security vulnerabilities in our Python code. It scans the codebase for common security issues, such as hardcoded passwords, insecure function calls, and potential vulnerabilities. By integrating Bandit into our CI/CD pipeline, we proactively address security concerns and maintain a secure codebase.

6.4. Manual Testing

Manual testing played a crucial role in our project, particularly in validating the correctness of the generated Kubernetes manifests. Since the manifests were created programmatically, it was necessary to manually verify that they could be successfully applied to a Kubernetes cluster and function as intended.

6.4.1. User Tests

We also conducted manual user testing of the webserver interface. The goal was to ensure that the platform is intuitive and accessible, even for users with limited technical background. These tests helped us identify usability issues and refine the user experience to make the creation of Kubernetes resources as straightforward as possible.

6.5. Code Review

We conducted code reviews at critical points in the project to ensure a comprehensive understanding of the entire project and to maintain high code quality. We utilized GitLab Merge Requests for this purpose. Once a feature was completed on a feature branch, a merge request to the protected main branch was created. This merge request was then reviewed by the other team member before being merged.

7. Infrahub

Infrahub defines our Kubernetes infrastructure as structured data and generates deployable manifests. This chapter covers schema definitions, Python-based transformations, template usage, artifact generation, GraphQL Queries and Git integrations.

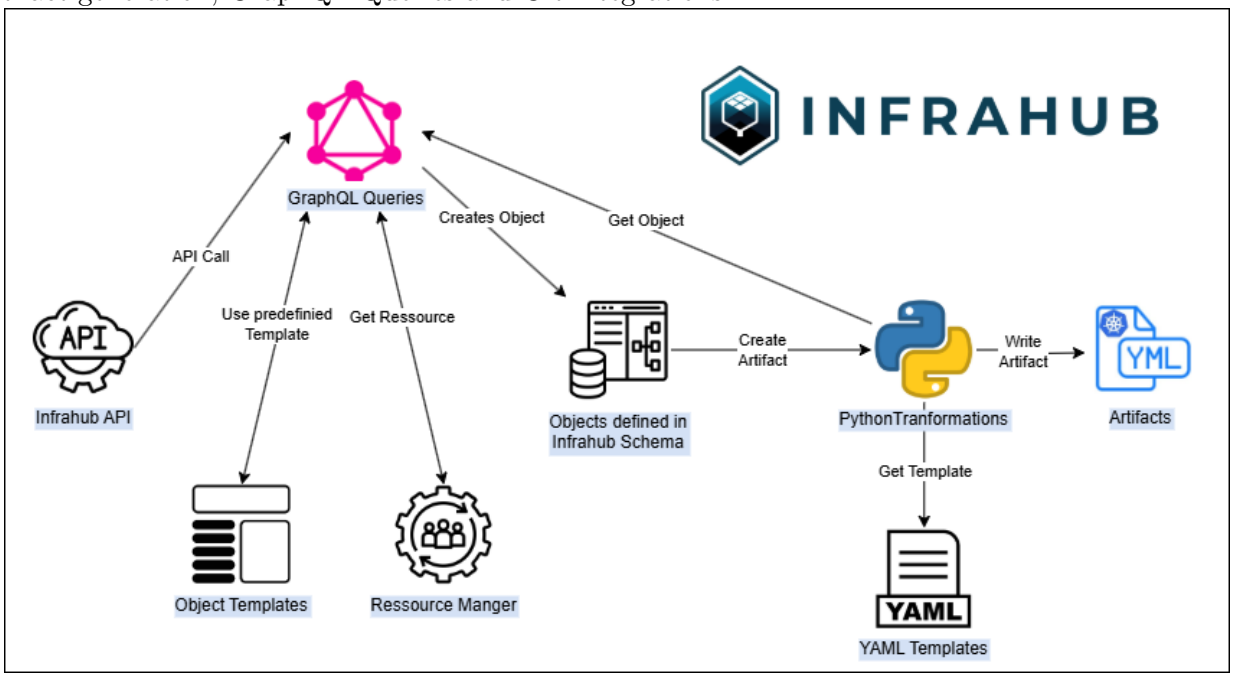

Figure 7.1.: Infrahub Resources

7.1. Schema

In Infrahub, the schema specifies the structure of the infrastructure data, specifying the Kubernetes resources to be deployed.

7.1.1. UML Diagram

The UML diagram illustrates the defined schema structure. The two nodes, **Webserver** and **VirtualMachine**, each have their own unique attributes while inheriting shared attributes from the generic **Resource**. Additionally, they both incorporate relationships defined by the **CoreArtifactTarget**.

Each node maintains a dependency on the **Template** class, as the template relies on these nodes to function meaningfully. Notably, the **VirtualMachine** node includes an attribute called **Port**, defined as a number. However, this attribute is read-only and its value is dynamically assigned by an external Resource Manager.

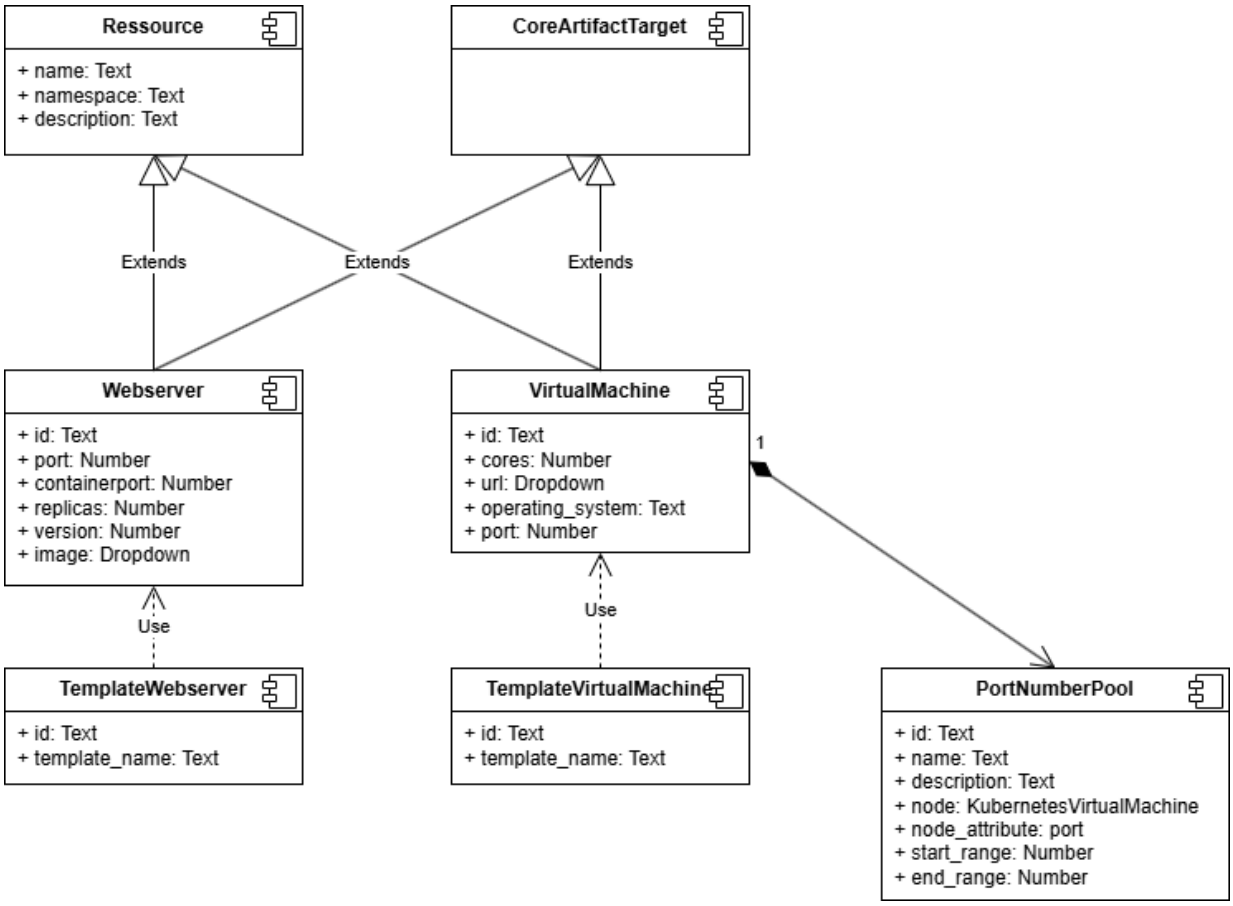

Figure 7.2.: Infrahub Schema UML Diagram

7.1.2. Generics - Resource

Generics represent reusable data structures that can be attached to multiple nodes. Each generic is defined by its own set of attributes, such as `name`, `namespace`, and `description`

```
generics:
  - name: Resource
    namespace: Kubernetes
    description: Generic Device Data
```

We also define the attribute `branch: aware` to ensure that changes made to this object are scoped only to the branch in which they are created. This allows for isolated modifications and safer collaboration across different development or deployment contexts.

```
branch: aware
```

The following attributes specify how the nodes will be presented. The `display_label` attribute determines which property is used as the identifier shown to the user and dictates the value displayed in the GUI.

Meanwhile, the `order_by` attribute defines the sorting order of the node objects. In this example, the nodes are sorted alphabetically based on their `name` attribute.

```
display_labels:
    - name__value
order_by:
    - name__value
```

The unique constraints define the uniqueness of an element. In our case, the combination of `name` and `namespace` must be unique within the entire Infrahub setup. This constraint ensures that no two objects can share the same name within the same namespace.

```
uniqueness_constraints:
    - ["name__value", "namespace__value"]
```

Generics can be treated as nodes, allowing users to see them in the menu. For our use case, we disable this feature so that users only see the nodes we have defined, and not the generic tab.

```
include_in_menu: false
```

In the **Attributes** section, we define which attributes are inherited by the children. This generic definition provides common attributes to all nodes, so the attributes `name`, `namespace`, and `description` will be present in all objects.

All of these attributes are of type `Text` and follow a fixed order. Additionally, `namespace` and `name` are required fields.

```
attributes:
  - name: name
    kind: Text
    description: Name of your Webservice
    order_weight: 1
  - name: namespace
    kind: Text
    description: Namespace name - Default ns-namespace
    order_weight: 2
  - name: description
    kind: Text
    description: Additional Information about the Webservice
    optional: true
    order_weight: 3
```

7.1.3. Nodes

General Attributes

The objects are identified by their name and share the same namespace as the generic definition. The icon represents the Webserver and the node will be visible in the menu.

```
- name: Webserver | VirtualMachine
  namespace: Kubernetes
  icon: mdi:hand-extended | mdi:linux
  include_in_menu: true
```

The node definition includes a directive to generate a template. This setting establishes a relationship with `CoreTemplate` and `CoreProfiles`, which are required for creating a template.

Using `inherit_from`, we generalize the node by linking it to two parent objects, thereby inheriting their attributes. The inclusion of `CoreArtifactTarget` is necessary to enable artifact creation, as it provides the required relationships to the nodes `CoreArtifact` and `CoreGroup`.

```
generate_template: true
inherit_from:
  - KubernetesResource
  - CoreArtifactTarget
```

Webserver Attributes

The attribute `port` and `containerport` are validated using a regular expression to ensure they falls within the range of 1 to 65535.

```
attributes:
  - name: port
    kind: Number
    regex: >
      ^(6553[0-5]|
      655[0-2][0-9]|
      65[0-4][0-9]{2}|
      6[0-4][0-9]{3}|
      [1-9][0-9]{0,3})$
```

The attributes `replicas` and `version` are defined as numbers, with input validated using a regular expression to ensure values are within the range of 1 to 5.

```
  - name: replicas
    kind: Number
    regex: ^[1-5]$
```

The attribute `host` defines the ingress URL where the webserver is reachable. It is a computed attribute based on the `name` and the DNS of our Kubernetes server.

```
- name: host
  kind: Text
  description: URL of the Webserver
    x.cldop-test-0.network.garden
  read_only: true
  optional: false
  computed_attribute:
    kind: Jinja2
    jinja2_template: "{{ name__value
    }}.cldop-test-0.network.garden"
```

The attribute `image` is a `Dropdown` allowing the user to select the desired image. Since this value is sensitive for Kubernetes, we predefined the available options.

```
- name: image
  kind: Dropdown
  choices:
    - name: httpd:latest
      description: Image for the Apache
    Webserver
      color: "#7f7fff"
```

Virtual Machine Attributes

For a Virtual Machine, different attributes are required, such as `cores`, which is simply defined as a number.

```
attributes:
  - name: cores
    kind: Number
```

The user can select a predefined image URL. A `Dropdown` is used for this attribute, as the value is sensitive to Kubernetes requirements and must be accurate.

```
  - name: url
    kind: Dropdown
    choices:
      - name: |
          docker://quay.io/
          containerdisks/ubuntu:24.04
```

The attribute `operating_system` is automatically derived from the attribute `url`. This attribute was added to provide a clearer overview of the operating system in use.

```
  - name: operating_system
    computed_attribute:
      kind: Jinja2
      jinja2_template: "{{
    url__description|lower }}-vm"
```

The last attribute, `port`, defines the SSH service port for the VM. It is a number without regex validation, as the resource manager assigns it. This relationship is set up in the resource manager's creation script and is not shown in the schema.

```
  - name: port
    kind: Number
    description: Port used for ssh
    optional: false
```

7.2. Python Transformation

The Python transformation function generates an artifact for each object based on the artifact definition. We use this transformation to create our Kubernetes manifests. The files are separated into “Transform” and “Helper Function” parts.

Each transform file corresponds to a specific node, such as one for Webserver and another for VirtualMachine. The Helper Function is a separate class used generally by all transform functions. We designed this architecture to keep it as general and reusable as possible.

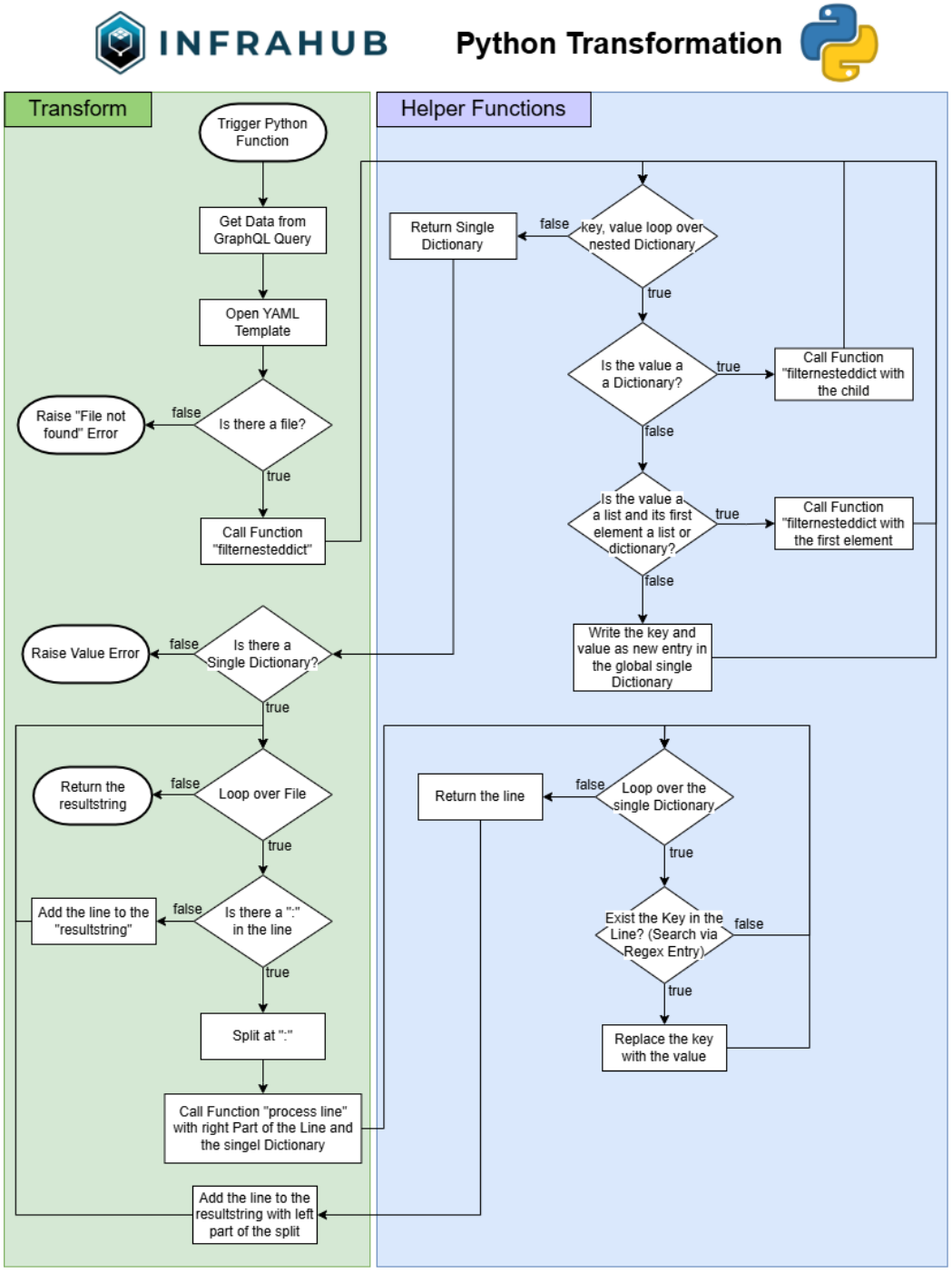

Figure 7.3.: Infrahub Python Transformations

7.2.1. Transform Function

Python Libraries

We use the following Python Libraries in our Code:

Library	Module	Description
`typing`	`Dict, Any`	Used for type annotations and flexible typing
`infrahub_sdk.transforms`	`InfrahubTransform`	Provides the core transformation methods used in the pipeline
`helperfunction`	`helperfunctions`	Custom utility methods developed in-house
`pathlib`	`Path`	Used to handle file paths, especially for YAML files

Table 7.1.: Python libraries used in the transformation scripts

Class and Modul definition

We use the class `TransformWebserver`, which inherits from the interface `InfrahubTransform`. This allows us to implement the asynchronous method `transform` for processing the transformation.

The `query` variable defines the GraphQL query used to retrieve the data stored in Infrahub. The name used in the query must match the one defined in the `.infrahub.yml` file located in the root directory of the Git repository.

```
class TransformWebserver(InfrahubTransform):
    query = "GetWebserver"
    async def transform(self, data: Dict[str, Any]) -> str:
```

Variables

`currentpath` resolves the path of the current Python file, allowing reliable access to YAML files. Since each Infrahub connection uses a uniquely identified Git copy, this ensures path validity despite changing directory names.

`pathfile` is the result of combining `currentpath` with the relative path to a specific YAML file. This value differs for each transformation, as each node (e.g., Webserver or VirtualMachine) requires its own YAML template.

`resultstring` stores the final result of the transformation. We accumulate the output within this variable and return it at the end. It must be initialized outside the loop because we only want to define it once, but append data to it within the loop.

The following example shows how the path to a YAML template is defined. The VirtualMachine templates follow the same pattern.

```
currentpath = Path(__file__).resolve()
pathfile = str(currentpath.parents[1]) + "/YAML_Templates/webserver.yaml"
resultstring = ""
```

Open File and get Single Dictionary Value

We open the YAML file within a `with` block and then call our custom `filternesteddict` function. This function takes a nested dictionary and returns a single dictionary containing the required key-value pair. As a control step, we check whether the returned dictionary is empty; if it is, we raise a `ValueError`.

```
try:
    with open(pathfile, "r") as yamlfile:
        customizedkeyvalue = HelperFunctions.filternesteddict(data)
        if not customizedkeyvalue:
            raise ValueError("No matching keys found in the input data.")
```

Loop over File

We now loop over the YAML lines and process each line. We split the line by the `:` character (if it exists) and call our custom function `process_line` with the right-hand side of the first `:` in the line, along with our single dictionary.

Since we receive a list from the function containing the modified value, we need to convert it into a string by joining all elements without including the brackets (`[]`).

If no `:` exists in the line, we add the line unchanged to the result string.

After the loop finishes, we return the string containing the modified YAML content.

```
for line in yamlfile:
    if ":" in line:
        lineprefix = line.split(":")
        lineresult = HelperFunctions.process_line(
            "".join(str(element) for element in lineprefix[1:]), customizedkeyvalue
        )
        resultstring += lineprefix[0] + ":" + lineresult
    else:
        resultstring += line

return resultstring
```

Error Handling

We handle errors using a `try` and `except` block. Since the code above is inside a `try` block, we catch the `FileNotFoundError` separately from other exceptions to provide more specific error messages.

7.2.2. Helper Functions

Libraries

From the `typing` library, we use the types `Dict`, `Any`, and `cast`. The `re` module is used to interpret and apply regular expressions (regex).

```
from typing import Dict, Any, cast
import re
```

Variables

We only need one class variable, `singledict`, for the `filternesteddict` function. This is necessary because we recursively traverse the dictionary and require the result dictionary to be initialized and consistent throughout the recursion.

```
class HelperFunctions:
    singledict: Dict[str, str] = {}
```

filternesteddict()

The goal of this function is to take a nested dictionary and transform it into a single (flat) dictionary. We need this function because the GraphQL query returns a nested dictionary that mirrors the structure of the query itself, but we require a flat dictionary for further processing.

Example:

```
nesteddict = {
    KubernetesWebserverCreate: {
        edges: [
            node: {
                name: {
                    value: "MyWebsrv"
                },
            }
        ]
    }
}
```

```
singeldict = {
    name= "MyWebsrv",
}
```

We define this function as `@staticmethod` because it does not operate on an instance of a class. The function takes two arguments: the nested dictionary (`nesteddict`) and a `key` of type `str`, which defaults to an empty string. We require the `key` argument because, in order to build the `singledict`, we need both the attribute name (as the key) and its value. Unfortunately, this key-value pair is not located at the same level in the nested dictionary. Therefore, during recursive calls, we pass the `key` as an argument to preserve it across different levels of the structure.

```
@staticmethod
def filternesteddict(nesteddict: Dict[str, Any], key: str = "") -> Dict[str, str]:
```

First, we start at the top level and iterate over the nested dictionary. To determine whether recursion is needed, we check the type of each value. If the value itself is a dictionary, we call the same function recursively, passing the value as the new dictionary and the `key` argument set to the current nested key.

Once the recursive call is complete, we `continue` with the next element at the same level.

```
for nestedkey, value in nesteddict.items():
    if isinstance(value, dict):
        HelperFunctions.filternesteddict(value, nestedkey)
        continue
```

If the value is a list, we need to determine the type of its elements. For our use case, it is sufficient to check only the first element, since our artifacts are defined per object and we always expect a single element in the list.

We check for the presence of logical operators such as `and` and `or`, and verify whether the type of the first element is either a dictionary or another list. If so, we call the same function recursively with the child element and its associated key.

To satisfy the `mypy` linter, we perform an explicit `cast` on the list element before further processing.

```
    if isinstance(value, list) and (
        isinstance(value[0], dict) or isinstance(value[0], list)
    ):
        HelperFunctions.filternesteddict(
            cast(Dict[str, Any], value[0]), nestedkey
        )
        continue
```

If all of the checks are false, we know that the type of the value is either an `int` or a `str`, which indicates that we have reached the lowest level of the nested structure. At this point, we add a new entry to our `singledict` using the provided `key` argument and the corresponding value.

Both the key and the value are converted to lowercase to ensure consistency throughout the dictionary.

As soon as the loop is finished, we return the `singledict`.

```
    HelperFunctions.singledict[key.lower()] = str(value).lower()
return HelperFunctions.singledict
```

process_line()

The function "Process_line" replaces the value in the yaml if required. We divided the functionality into two functions, "process_line" and "match_key_in_line". "Process_line" iterates over the single dictionary and changes the value, "match_key_in_line" checks if the key from the single dictionary is in the given line.

process_line This method takes two arguments: the `line` in which we search for a key, and the `customizedkeyvalue` dictionary (our single dictionary). While we could use a class variable for the dictionary, doing so would increase the risk of unintended side effects. In our use case, the `line` typically represents the right-hand side of a `:` from the YAML input.

```
def process_line(line: str, customizedkeyvalue: Dict[str, Any]) -> str:
```

Next, we iterate over the entries in the single dictionary and use the `match_key_in_line` function to check whether a key is present in the line. If a match is found, we replace the key with its corresponding value. Otherwise, we continue the loop.

At the end of the function, we return the (potentially modified) line.

```
for key, value in customizedkeyvalue.items():
    if HelperFunctions.match_key_in_line(line, key):
        line = line.replace(key, value)
return line
```

match_key_in_line This function takes a given `line` and a `key`, and checks whether the key appears in the line. Since we want to avoid partial matches (e.g., matching `name` inside `namespace`), we use a regular expression for more precise control.

The regular expression ensures that the key is surrounded by non-word characters (such as hyphens, spaces, or punctuation), making sure we only match complete words. The search is case-insensitive to account for variations in capitalization.

The function returns a boolean indicating whether the key exists in the line.

```
def match_key_in_line(line: str, key: str) -> bool:
    pattern = rf"\W{re.escape(key)}\W"
    return bool(re.search(pattern, line, re.IGNORECASE))
```

7.2.3. YAML Templates

Since the focus of this Bachelor thesis is not on the optimal setup of a web server, we do not describe the entire YAML file but only those parts that are essential for the transformation and our deployment.

Kinds

For our **Webserver**, we are creating the following Kind to run.

- Namespace
- Deployment
- Service
- Ingress

For our **Virtual Machine**, we are creating the following Kind to run.

- Namespace
- Kubevirt VirtualMachine
- Persistent Volume
- Service

Naming Convention

In our YAML configuration files, we follow a specific naming convention for Kubernetes kinds. The prefixes are used to easily identify the type of each resource:

Kind	Naming Prefix	Example
Namespace	`ns-`	`ns-test`
Deployment	`dep-`	`dep-test`
Service	`svc-`	`svc-test`
Ingress	`ing-`	`ing-test`
VirtualMachine	`vm-`	`vm-test`
Persistent Volume	`pvc-`	`pvc-test`

Table 7.2.: Naming conventions for Kubernetes kinds

Replacement with Infrahub Values

The transformation process is designed to substitute placeholder keys in the YAML file with corresponding values from a single dictionary. Below is an example of how the replacement logic works:

YAML Before	Dictionary Key	YAML After
`Namespace: ns-namespace`	`namespace: test`	`Namespace: ns-test`
`Name: svc-service`	`service: servicetest`	`Name: svc-servicetest`
`replicas: replicas`	`replicas: 2`	`replicas: 2`

Table 7.3.: Example of key-based value replacement in a YAML file

7.3. Graph QL Queries

Graph QL Queries are used to mutate and get the Objects defined in Infrahub. We define the queries in the Git repository and call it via API or via Transformation. Here's an example of how our GraphQL Query looks:

Create Object

```
mutation CreateWebserver
    ($name:String!,
    $description:String!,
    $namespace:String!, $image:String!) {
  KubernetesWebserverCreate(
    data: {
        object_template: {hfid:
    "tem-webserver"},
        name: {value: $name},
        description: {value:
    $description},
        namespace: {value: $namespace},
        image: {value: $image},
        member_of_groups: {hfid:
    "g_webserver"}
    }
  ) {
    object {
      host {
        value
      }
    }
  }
}
```

Get Object

```
query GetWebserver ($webserver:
    String!) {
  KubernetesWebserver (name__value:
    $webserver) {
    edges {
      node {
        name {
          value
        }
        port {
          value
        }
        containerport {
          value
        }
        replicas {
          value
        }
        image {
          value
        }
        namespace {
          value
        }
        host {
          value
        }
      }
    }
  }
}
```

We use the following GraphQL Queries in our implementations:

Mutation	GraphQL Name	Variables
Create	CreateBranch	branchname: str
Create	CreateProposedChange	sourcebranch: str, name: str, description: str
Create	Webserver	name: str, description: str, namespace: str, image: str
Create	VirtualMachine	name: str, description: str, namespace: str, url: str
Get	Webserver	webserver: str
Get	VirtualMachine	virtualmachine: str
Get	ArtifactIDs	artifactname: list[str]
Upsert	Group	
Upsert	Object Template	
Upsert	Role	
Upsert	ObjectPermission	
Upsert	UserGroup	
Upsert	User	
Upsert	GitIntegration	
Upsert	Password Credentials	
Upsert	Numbers Pool	

Table 7.4.: GraphQL mutations used in Infrahub

7.4. .infrahub.yml

At the root of our Git repository, the `.infrahub.yml` file defines the structure and logic for Infrahub resources. It is essential that this file resides in the root directory, as it specifies where Infrahub should locate the various components.

Queries

GraphQL query files are stored in the `GraphQL` folder of the repository with the `.gql` extension. Each query is defined by its name and the path to the corresponding file:

```
queries:
  - name: GetWebserver
    file_path: "GraphQL/GetWebserver.gql"
```

Schemas

The GraphQL schema is also placed in a dedicated folder. Since our schema is relatively simple, we define the complete structure in a single file rather than splitting it into multiple parts:

```
schemas:
  - "Schema/service-schema.yaml"
```

Python Transformations

For each Python transformation, we specify its internal name, the name of the Python class to execute, and the path to the Python source file:

```
python_transforms:
  - name: TransformWebserver
    class_name: TransformWebserver
    file_path: "python_transform/transform_webserver.py"
```

Artifact Definitions

Artifact definitions are declared directly within the `.infrahub.yml` file rather than in a separate one. Each artifact definition specifies how the artifact should be created, including:

- The parameters used as input variables for the GraphQL query
- The content type of the resulting artifact
- The associated Python transformation
- The target group of objects for which artifacts will be generated

For each object in the target group, Infrahub will generate a corresponding artifact:

```
artifact_definitions:
  - name: "Webserver_Artifact_Definition"
    artifact_name: "Webserver_Manifest"
    parameters:
      webserver: "name__value"
    content_type: "application/yaml"
    targets: "g_webserver"
    transformation: "TransformWebserver"
```

7.5. Resource Manager

The Resource Manager is initialized by our start script. We use it as a dynamic number pool to allocate port numbers in the range of `30000` to `32767`, which are used for the SSH service in Virtual Machine deployments.

The main advantage of the Resource Manager is its ability to automatically assign an available port to each newly created object, ensuring there are no conflicts or manual assignments required.

Our query returns both the `hfid` and `id` of the newly created object, but we currently do not store this output.

While we could use the returned `id` to specify the target resource pool in a subsequent GraphQL mutation for creating a VirtualMachine, this approach would significantly increase the complexity of the deployment. This is because the GraphQL queries are defined in separate files and preloaded into Infrahub, making dynamic value injection across queries more difficult to manage.

```
    mutation {
      CoreNumberPoolCreate(data:{
        name: {value: "portpool"},
        node: {value: "KubernetesVirtuellMaschine"},
        node_attribute: {value: "port"},
        start_range: {value: 30000},
        end_range: {value: 32767}
      })
      {
        ok
        object {
          hfid
          id
        }
      }
    }
```

7.6. Users, Groups, Roles, and Permissions

Permissions in Infrahub define access rights and are assigned to roles. A role can be associated with multiple user groups, and likewise, a user group can have multiple roles. Each group can contain one or more users, and users must authenticate to Infrahub in order to perform actions according to their assigned roles.

7.6.1. Users

Users must authenticate using a username and password and can be assigned to one or more user groups.

In our setup, we use a predefined user named `admin`, who acts as the system administrator with full access rights. In addition, we use a restricted user called `g_createservice`, which is assigned to a group with limited permissions scoped to the `Kubernetes` namespace.

admin

The `admin` user is a predefined system account with full administrative privileges. It has unrestricted access to all Infrahub resources and operations. We use this user in our start script to initialize the environment and perform privileged actions, such as setting up GitLab integration or creating service users.

g_createservice

The `g_createservice` user is designed for controlled access. It is allowed to perform full CRUD operations on objects within the `Kubernetes` namespace, except for those in the main branch. This user is primarily used to create infrastructure objects.

The user is created using the following GraphQL mutation:

```
mutation {
  CoreAccountUpsert(
    data: {
      name: {value: "g_createservice"},
      password: {value: "g_createservice"},
      member_of_groups: {hfid: "g_createkubernetesobjects"}
    }
```

```
  ) {
    ok
  }
}
```

7.6.2. Roles

Roles are abstract definitions of permissions. They are assigned to user groups to grant specific capabilities. The following mutation creates the role used for Kubernetes-related permissions:

```
mutation {
  CoreAccountRoleUpsert(
    data: {
      name: {value: "role_createkubernetes"}
    }
  ) {
    ok
  }
}
```

7.6.3. Permissions

Permissions are linked to roles and define what actions can be taken on which objects. The following example grants the `role_createkubernetes` role the ability to create any object within the `Kubernetes` namespace:

```
mutation {
  CoreObjectPermissionUpsert(
    data: {
      namespace: {value: "Kubernetes"},
      name: {value: "*"},  # All objects in this namespace
      action: {value: "create"},
      decision: {value: 4 },  # 4 is the enum value for "allow_other"
      roles: [{hfid: "role_createkubernetes"}]
    }
  ) {
    ok
  }
}
```

7.6.4. User Groups

Object groups are used to associate roles with users via group membership. In the following mutation, the group `g_createkubernetesobjects` is assigned the `role_createkubernetes` role:

```
mutation {
  CoreAccountGroupUpsert(
    data: {
      name: {value: "g_createkubernetesobjects"},
      roles: [{hfid: "role_createkubernetes"}]
    }
  ) {
    ok
  }
}
```

7.7. Service Groups

Service groups allow logical grouping of objects such as `VirtualMachines` and `Webservers`. These groups are referenced in the artifact definition to determine for which objects Infrahub should generate artifacts.

For each node, we define a service group: `g_webserver` and `g_virtuellmachine`. The association of objects with their respective service groups is handled via GraphQL mutations, where the objects are explicitly added to the correct group.

The following example creates the `g_webserver` group:

```
mutation {
  CoreStandardGroupUpsert(
    data: {
      name: {value: "g_webserver"}
    }
  ) {
    ok
  }
}
```

7.8. Git Integration

The Git integration in Infrahub is used to load schemas, Python transformation scripts, GraphQL queries, and artifact definitions. All related files are stored in our Git server repository, and the Infrahub server clones the entire branch—this includes files not explicitly specified in the `.infrahub.yml` configuration file.

We have chosen to configure the Git repository as a **read-only repository**, targeting a specific branch without automatic updates on new commits. This decision was made to avoid importing test branches into the Infrahub environment and to prevent every single commit from being applied immediately.

Authentication

Authentication is handled using a GitLab **deployment token**, which we created at the project level. This token has sufficient permissions to access the repository and does not expire.

```
name: { value: "gitlab-deployment-token" },
username: { value: "gitlab+deploy-token-infrahub" },
password: { value: "No the Password!" }
```

Read-Only Repository Setup

We configure the read-only repository in Infrahub using the following mutation. The `ref` refers to the branch that should be used (e.g. `Corrected-Pipeline`):

```
CoreReadOnlyRepositoryUpsert(
  data: {
    name: { value: "Gitlab Inventory" },
    location: { value:
    "https://gitlab.ost.ch/ins-stud/sa-ba/ba-fs25-infrahub/infrahubintegration.git" },
    ref: { value: "Corrected-Pipeline" },
    credential: { hfid: "gitlab-deployment-token" }
  }
```

```
) {
  ok
}
```

7.9. Object Templates

Infrahub object templates predefine default values for object attributes, reducing required user input to only essential fields such as `name`, `namespace`, `description`, and `image` (or `url` for Virtual Machines). All other attributes are set by the template or computed automatically.

Webserver Template

Attribute	Value Input
name	User Input
namespace	User Input
description	User Input
port	Template
containerport	Template
replicas	Template
version	Template
host	Computed Attribute
image	User Input

Virtual Machine Template

Attribute	Value Input
name	User Input
namespace	User Input
description	User Input
cores	Template
url	User Input
operating_system	Computed Attribute
port	Resource Manager

Webserver Template Creation

The GraphQL mutation is used to creates a template for a webserver, with predefined values:

```
mutation {
  TemplateKubernetesWebserverUpsert(
    data: {
      template_name: {value: "tem-webserver"},
      port: {value: 80},
      containerport: {value: 80},
      version: {value: 1},
      namespace: {value: "default"},
      replicas: {value: 1}
    }
  ) {
    ok
  }
}
```

8. Self-service Frontend

Our Frontend – also referred to as the "web shop" or "self-service portal" – provides users with a simple interface to request resources. We deliberately chose to implement a frontend because neither letting users create objects directly via generics nor using Infrahub generators met our requirements.

The main goal was to provide a user-friendly way to request infrastructure resources while ensuring that each user can only view and fill in the attributes relevant to them. Furthermore, access control was essential: each user should only be able to view their own virtual machines and related data.

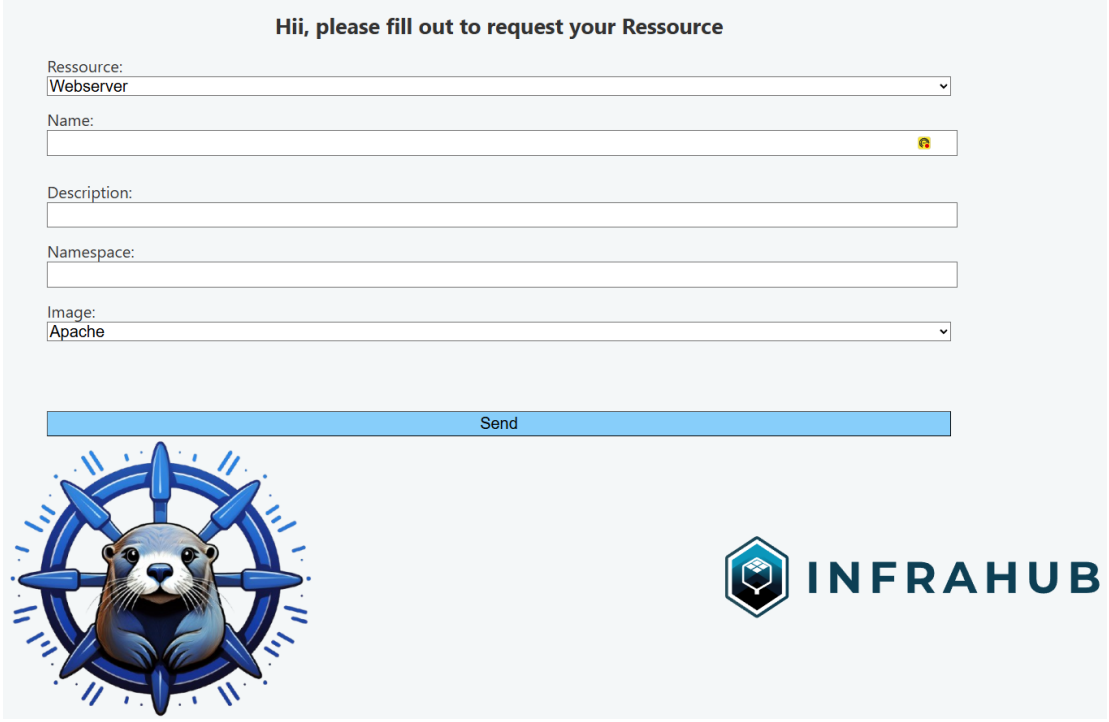

Figure 8.1.: Architecture of the self-service frontend

8.1. Architecture

The architecture of the frontend, along with its components, is illustrated in Figure 8.2. Below is an overview of the main components:

- **Frontend:** Developed using HTML, CSS, and JavaScript to provide an interactive form.
- **Backend:** A Python Flask application that handles requests and implements the logic.
- **Python:** Handles the Infrahub API calls to create the objects.

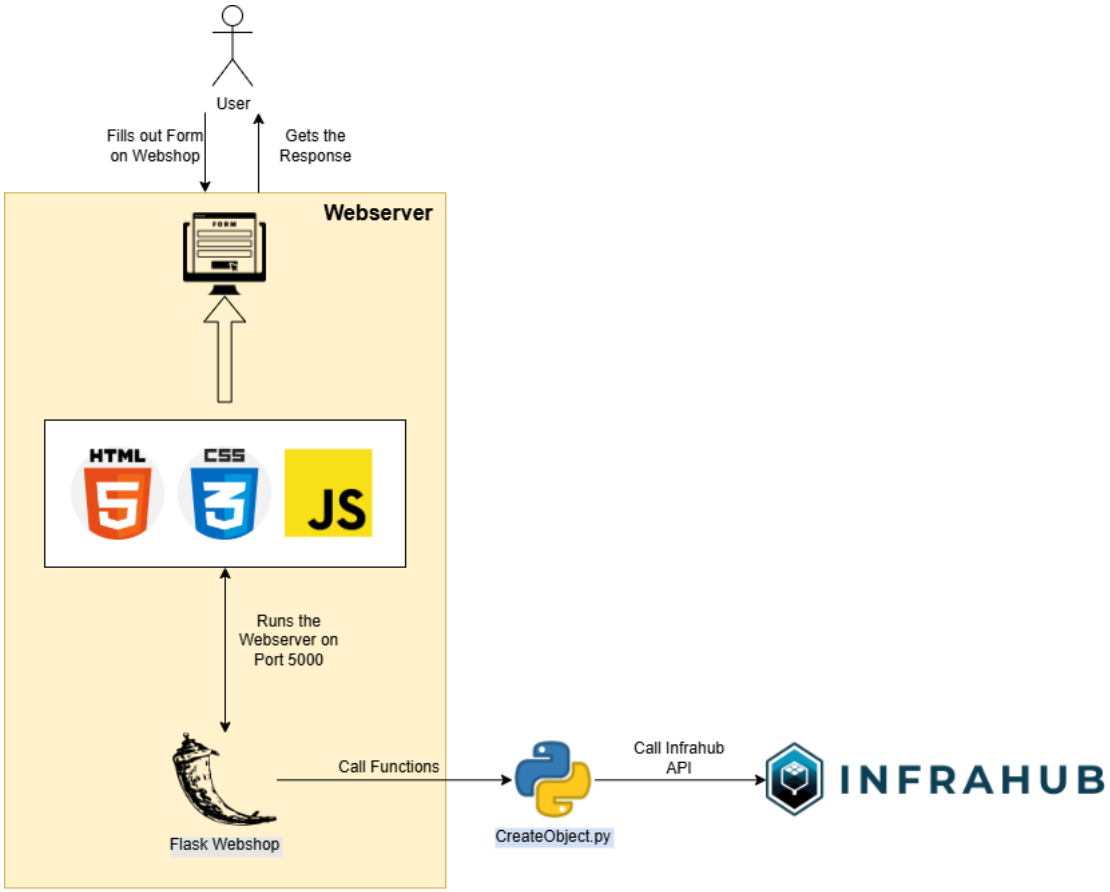

Figure 8.2.: Architecture of the self-service frontend

8.2. HTML Interface

The HTML interface forms the visual component of the self-service frontend. It provides a form where users can request either a webserver or a Virtual Machine (VM), depending on their needs. For the full documentation of the code, please go to Appendix 3.

8.3. Flask

This Flask application serves a web interface that allows users to request the deployment of webservers or virtual machines. Based on form inputs, the app calls backend Python scripts to process the deployment.

8.3.1. Python Libraries

The following code snippet shows the imported Python libraries necessary for the application:

```
from flask import Flask, render_template, request
import sys
import os
from typing import Optional

sys.path.append(os.path.abspath(os.path.join(os.path.dirname(__file__), "..")))
from python_scripts.createobjects import createwebserver, createvirtuellmaschine
```

8.3.2. index function

The `index()` function defines the main route of the web application. It handles both `GET` and `POST` requests. User input is retrieved via form labels defined in the HTML template.

The variables `name` and `namespace` are used to generate a new branch name. Since these branch names are used in URLs for API requests, they must be valid and URL-safe. Therefore, spaces in user inputs are replaced using the `replace()` function, as spaces are not allowed in URLs.

```
@app.route("/", methods=["GET", "POST"])  # type: ignore
def index() -> str:
    if request.method == "POST":
        deployment_type = request.form.get("deployment_type")
        name = request.form.get("name", "").replace(" ", "_")
        description = request.form.get("description", "").replace(" ", "_")
        namespace = request.form.get("namespace", "").replace(" ", "_")
        url = request.form.get("url")
        image = request.form.get("image")

        result = run_python_script(deployment_type, name, description, namespace, url,
    image)
        return render_template("index.html", result=result)  # type: ignore

    return render_template("index.html", result=None)  # type: ignore
```

8.3.3. Python Script

The function `run_python_script()` processes the deployment logic based on the user's form input.

Variables

This snippet shows the function signature with typed parameters, including optional fields:

```
def run_python_script(
    deployment_type: str,
    name: str,
    description: str,
    namespace: str,
    url: Optional[str],
    image: Optional[str],
) -> str:
```

Match Deployment Type

A `match` statement is used to route the request based on the deployment type:

```
    try:
        match deployment_type:
            case "Webserver":
                result = createwebserver(name, description, namespace, image)  # type:
    ignore
            case "VM":
                result = createvirtuellmaschine(name, description, namespace, url)  #
    type: ignore
            case _:
                return "Invalid deployment type."
```

Exception

The function is wrapped in a `try-except` block to catch and handle runtime errors:

```
    except Exception as e:
        print(f"Error while processing the request: {e}")
        return f"""
            An unexpected error occurred. Please try again later.
            If the problem persists, please contact your network administrator.
            Error: {e}
        """
```

8.4. Python - CreateObjects

8.4.1. Python Script

This Python script defines two core functions for creating either a erver or a virtual machine by sending GraphQL requests to an infrastructure backend. It also handles path adjustments and uses helper functionality.

Variables

The script begins by importing necessary libraries and adjusting the system path to ensure local modules like `helperfunctions.py` can be imported. The helper class `_HelperFunctions` is then used to send GraphQL requests.

```
import sys
import os
from typing import Any

sys.path.append(os.path.abspath(os.path.join(os.path.dirname(__file__), ".")))
from helperfunctions import _HelperFunctions
```

Match Deployment Type

The function `createwebserver()` is responsible for creating a webserver. It prepares a GraphQL payload with the required parameters such as `name`, `description`, `namespace`, and `image`, and sends it using a helper method.

The branch name is generated by concatenating `namespace` and `name`, ensuring uniqueness within the system. The response from the server contains the computed attribute `host`, which includes the full URL where the deployed webserver can be accessed. Since the `host` field in Infrahub is a computed attribute, it returns the entire accessible URL directly.

```
def createwebserver(name: str, description: str, namespace: str, image: str) -> Any:
    """Create the Webserver with the given attributes."""
    graphgql = "CreateWebserver"
    payload = {
        "variables": {
            "name": name,
            "description": description,
            "namespace": namespace,
            "image": image,
        }
    }
    json_data = _HelperFunctions._send_graphql(
```

```
        graphgql, payload, branch=namespace + name
    )
    return json_data["data"]["KubernetesWebserverCreate"]["object"]["host"]["value"]
```

Similarly, the `createvirtuellmaschine()` function sends a request to create a virtual machine. After deployment, it returns an SSH command that can be used to access the machine.

```
def createvirtuellmaschine(
    name: str, description: str, namespace: str, url: str
) -> Any:
    """Create the virtual machine with the given attributes."""
    graphgql = "CreateVirtuellMaschine"
    payload = {
        "variables": {
            "name": name,
            "description": description,
            "namespace": namespace,
            "url": url,
        }
    }
    json_data = _HelperFunctions._send_graphql(
        graphgql, payload, branch=namespace + name
    )
    return f"ssh {name}@10.8.36.20 -p
    {json_data['data']['KubernetesVirtuellMaschineCreate']['object']['port']['value']}"
```

8.5. Python - HelperFunction

The `HelperFunctions` class provides static methods to support the webserver and VM creation process. Its primary responsibilities include sending GraphQL requests, creating branches in the version control system, and proposing infrastructure changes. All API interactions are secured with authentication and include error handling for failed operations.

_send_graphql Method

This method handles the full workflow for sending a GraphQL request to the backend. It first ensures the necessary branch exists, sends the main query, and then creates a proposed change to document the infrastructure update. The method throws errors if any API request fails.

```
@staticmethod
def _send_graphql(
    graphql: str, payload: Dict[str, Any], branch: str = "main"
) -> dict[str, Any]:
    responsebranch = _HelperFunctions._createbranch(branch)
    if responsebranch != 200:
        raise ValueError("An error occurred in the creation of the branch.")

    graphqlurl = f"http://localhost:8000/api/query/{graphql}?&branch={branch}"
    headers = {"X-INFRAHUB-KEY": "06438eb2-8019-4776-878c-0941b1f1d1ec"}
    response = requests.post(
        graphqlurl,
        auth=HTTPBasicAuth("g_createservice", "g_createservice"),
        headers=headers,
        data=json.dumps(payload),
    )
```

```
    if response.status_code != 200:
        raise ValueError("An error occurred in the creation of the branch.")

    if "name" in payload["variables"]:
        name = f"New Proposed Change for a Webserver {payload['variables']['name']}"
    description = f"{payload['variables']}"
    responseproposedchange = _HelperFunctions._createproposedchange(
        branch, name, description
    )
    if responseproposedchange != 200:
        raise ValueError("An error occurred in the creation of the proposed change.")

    return response.json()
```

_createbranch Method

This method issues a GraphQL mutation to create a new branch in the infrastructure repository. A branch is required to isolate configuration changes. The request uses admin user credentials because the service account lacks sufficient permissions (e.g., `write:repo`). A successful response returns HTTP status code 200, indicating that the branch was created successfully.

_createproposedchange Method

This method sends a GraphQL mutation to register a proposed infrastructure change. It takes the target branch, a change name, and a description as parameters. Its called at the end, and is used to associate metadata with the configuration change. This enables backend systems to process the change (e.g., validation, approval).

9. Technical Issues and Obstacles

This chapter discusses the technical issues and obstacles we encountered during the development of our project. Each section describes a specific problem, its implications, and the solutions we implemented to overcome it.

9.1. Infrahub

9.1.1. Resource Manager via Template

Description

The Resource Manager assigns a port to virtual machines for SSH access. Our goal was to define this association within the template, so that each time an object is created from the template, it automatically receives a port from the Resource Manager.

Unfortunately, Infrahub does not currently support this feature. Infrahub treats the template as a separate object, so when the Resource Manager assigns a port to the template, this port is then copied to all newly created objects—rather than generating a new, unique port for each one.

Solution

After consulting the community, we confirmed that this feature is not yet implemented. As a workaround, we adjusted our setup so that the Resource Manager assigns a port directly during the object creation process, rather than through the template.

9.1.2. Web Browser API Calls

Description

We encountered an issue when making API calls through a URL such as:

```
http://localhost:8000/api/query/{graphql}?&branch={branch}
```

In this case, the `branch` variable is constructed using user input that includes the name and namespace. However, certain characters—such as spaces, colons (`:`), and other special symbols—are not permitted in URLs and would cause the API call to fail.

Solution

To resolve this, we implemented input restrictions both in the HTML frontend and the Flask backend:

- Minimum length: 4 characters
- Maximum length: 15 characters
- Spaces are automatically replaced with underscores (_)
- Only commonly accepted special characters (e.g., hyphens, underscores) are allowed

This ensures that all user-generated branch names are valid for use in API calls, avoiding malformed URLs and request errors.

9.1.3. General GraphQL File

Description

To maintain a general and reusable structure, our initial goal was to use a single GraphQL query to fetch the desired object. This would simplify the process when introducing new resources such as a `Deployment`, as we would not need to create a new GraphQL query for each resource type.

Issue

However, GraphQL is designed to be explicit in defining the exact data requirements. This means that we must specify every attribute and its value individually—across all nodes in the schema. As a result, a general query is not feasible using standard GraphQL functionality.

Solution

We received a suggestion from the Infrahub developers to use their custom extension to GraphQL, which introduces an `@extend` directive at the node level. This allows retrieval of all attributes and their values without manually listing each one.

Despite this, we were unable to use the `@extend` directive effectively, as the GraphQL query still requires specifying the node name at the top level. Therefore, we ultimately decided to define a full GraphQL query for each node individually.

9.1.4. Calling YAML File

Description

As part of our Python transformation process, we needed to read from a predefined YAML template file. This template is stored within the same Git repository used by Infrahub. The goal was to reference and load this YAML file reliably during execution.

Issue

Infrahub copies the entire Git branch into its internal folder structure when executing the transformation. However, this structure includes dynamically generated folders, specifically one whose name is the Git Integration ID. Since this ID changes with every installation or Git integration, using an absolute path to locate the YAML file is not a viable solution.

Additionally, referencing the file with a relative path does not work as expected within the Infrahub execution environment, due to the altered internal path structure.

Solution

We solved this by using Python's `os` and `pathlib` libraries to dynamically determine the location of the current script, and then derive the path to the YAML file based on that. Since both the Python script and the YAML template are within the same Git repository, this method ensures consistent and correct path resolution.

Below is the relevant code snippet:

```
from pathlib import Path

currentpath = Path(__file__).resolve()
pathfile = str(currentpath.parents[1]) + "/YAML_Templates/virtuellmaschine.yaml"
```

This approach avoids hardcoding paths and allows the code to remain portable and functional across different environments.

9.2. Vidra Operator

There were several technical challenges related to the Vidra Operator. The most important solutions were documented in architectural decision records.

Event-Driven vs. Owns-Based Reconciliation for Managed Resources is an example of a technical issue that resulted in an architectural decision. In this architectural decision record, we discussed the challenges of using an event-based approach for monitoring managed resources in Kubernetes.

Storing Resource State is another example of a technical issue that led to an architectural decision. In this record, we explored the benefits and challenges of storing resource state directly in the Custom Resource (CR).

Other Architectural Decisions related to issues we encountered during the development of the Vidra Operator can be found in Vidra's architectural decision section of the Vidra Operator documentation.

9.3. General Issues

While the Operator and its CLI tool grew in size and complexity, we encountered several Operator SDK-, Go-, or Kubernetes-related issues. These issues were not specific to our project, but rather general challenges. We deemed them not important enough to document all of them.

Solution was mostly to consult the kubebuilder[1], operator sdk[2], kubernetes[3], and go client or k8s package[4] documentation.

[1] Kubebuilder documentation: June 8, 2025 https://book.kubebuilder.io
[2] Operator SDK documentation: June 8, 2025 https://sdk.operatorframework.io/docs/
[3] Kubernetes documentation: June 8, 2025 https://kubernetes.io/docs/home/
[4] Go package documentation: June 8, 2025 https://pkg.go.dev/k8s.io/client-go

Part II.

Project Documentation

1. Results

Key Achievements

The *Infrahub meets K8s* project successfully delivered a comprehensive and automated system for Kubernetes resource management. The key achievements are as follows:

1. **Automated Deployment of Kubernetes Manifests:** Developed **Vidra**, a custom Kubernetes Operator, to continuously monitor the desired state and fetch manifests for automated deployment.
2. **Multicluster Support:** Vidra supports multi-cluster environments by utilizing kubeconfig contexts (stored in Kubernetes Secrets) to read from and write to multiple clusters. It reconciles resources consistently across environments and maintains unique identity and ownership tracking for each cluster.
3. **Event-Driven Reconciliation:** Vidra employs an event-driven model to trigger reconciliation, responding immediately to resource changes. It dynamically adds Informers only for managed resources, minimizing overhead while supporting any Kubernetes resource type. This approach reduces latency in updates and syncs, and can be enabled per InfrahubSync or globally.
4. **Integration with Infrahub:** Utilized Infrahub as the single source of truth for managing and versioning manifests, ensuring consistency and centralized oversight of infrastructure and system states.
5. **Webserver and Virtual Machine Deployment:** Implemented support for deploying various services, including ervers and virtual machines, through the platform.
6. **Vidra CLI Tool:** Created a CLI tool for Vidra to simplify operations such as creating and updating secrets, managing configuration files, and syncing with Infrahub.
7. **User-Friendly Frontend for Resource Requests:** Developed a frontend interface allowing users to easily create and submit Infrahub objects, minimizing the need for technical expertise.
8. **Automated Workflow:** Leveraged Python and the Infrahub API to automate the resource request process, significantly reducing the manual workload for administrators.
9. **Scalability and Compatibility:** Designed the system to support multiple Kubernetes resource kinds and ensured extensibility of the Infrahub model for future use cases.
10. **Adherence to GitOps Principles:** Architected Vidra to fully comply with GitOps principles, enabling declarative configuration, versioned source control, and automated reconciliation.
11. **Security:** Implemented role-based access by using distinct users to interact with Infrahub, adhering to the Principle of Least Privilege and minimizing the potential attack surface.

2. Conclusion

We have successfully integrated Infrahub with Kubernetes (K8s) and developed the open-source Kubernetes Operator `Vidra` as part of this project. This demonstrates a modern approach to managing cloud-native infrastructure through Infrahub—an infrastructure modeling-driven inventory management system. By leveraging Operator SDK capabilities, we have created a solution that allows for the dynamic lifecycle management of any Kubernetes resource.

Additionally, we created supporting tools such as a CLI for Vidra and a self-service frontend for Infrahub.

Features of Vidra

The overall architecture is designed to be as generic as possible to support a wide range of continuous integration workflows and to be easily extensible for future needs.

- **Advanced CRDs:** Vidra has the potential to manage complex Custom Resources (CRs), such as:
  - *Network configurations* (e.g., Kubenet[1], SDC): Infrahub artifacts could define network policies and configurations.
  - *Cloud-native infrastructure* (e.g., Crossplane): Infrahub artifacts could define Crossplane resources for cloud services.
  - *Other Kubernetes resource types:* Both CRDs and built-in resources can be managed.

These capabilities require further testing and validation before being considered production-ready.

Possible Future Improvements

As we look forward to maintaining and enhancing Vidra, several improvements and features are possible:

Webhook-Based Reconciliation (Planned) Vidra will support webhooks to trigger immediate reconciliation on Infrahub changes, reducing update latency and reliance on periodic resyncs, and enabling faster feedback and state syncing.

Sync to Other Platforms The VidraResource abstraction is independent of Infrahub, enabling future support for Helm (managing resources via Helm charts), Git (enabling -style syncing from repositories), and other platforms that provide Kubernetes manifests.

Enhanced Observability Planned improvements include metrics and logging for operational insights, as well as tracing to follow resource lifecycle events.

User Interface Resource status and dependencies could be visualized in a UI, enabling real-time monitoring and management dashboards.

Overall, this project lays the groundwork for scalable, automated, and cloud-native infrastructure management and opens up new possibilities for further research and development in this area.

[1] Kubenet: 10.06.2025 https://learn.kubenet.dev

3. Project Planning

3.1. Processes

Our project is structured using an agile project management framework. This approach is particularly suitable for our needs, as we work with Epics and have meetings with our advisor every two weeks. This allows us to remain flexible for changes. We have organized our project into Phases (in our Jira called Epics), which help us manage our tasks and maintain clarity throughout the project lifecycle. Each Phase represents a significant component of our work and is further broken down into Tasks and Subtasks. This structured approach enables us to effectively track progress and ensure that all aspects of the project are addressed.

3.2. Architectural Roles

We designated **Architecture Agents** to leverage each team member's expertise in different technologies, fostering knowledge sharing throughout the project. Initially, we held a meeting to discuss the architecture and assign specific responsibilities. A later chapter will detail the reasoning behind our architectural decisions.

1. **Simon Linder:** Implements and manages the Kubernetes Deployment, handling and developing the Kubernetes Operators including the principles.
2. **Ramon Stutz:** Responsible for the Infrahub instance, including schema, Python transformations and API calls. Developing the self-service frontend for requesting a resource.

This distribution leverages each member's strengths while promoting collaborative decision-making within our team. It's essential that everyone contributes their technical expertise without feeling overruled, collectively shaping the project's architecture. Although primary tasks are assigned based on experience (Simon with **Kubernetes**, Ramon with **Infrahub**), we work closely together. This vertical slicing allows efficient domain-specific development while ensuring individual systems integrate seamlessly. We update each other weekly on current statuses and challenges, ensuring mutual support and meaningful contributions, especially in architectural decisions or cross-domain challenges.

3.3. Meetings

We conduct biweekly check-ins with our supervisor to address any challenges, deviations, or uncertainties that may arise. Additionally, our team aims to meet at least once a week to foster collaboration and support one another. If necessary, we can hold more frequent meetings to ensure we stay aligned and address any pressing issues promptly.

3.4. Phases

In our project, we have established a structured approach to planning by defining Phases as overarching goals, complemented by high-level tasks. Each phase is further decomposed into specific tasks and subtasks, facilitating a clear path toward project completion. The timeline allocated for each phase ranges from 1 to 3 weeks, with each phase assigned to a designated team member responsible for its execution.

Phase-ID	Name	Description	Due Date	Assignee
IMK-2	Documentation	Documentation of the project	13.06.2025	Simon Linder
IMK-3	Project Organisation	Keep Jira and the tools up to date	13.06.2025	Ramon Stutz
IMK-13	Project Initialisation	Start our thesis, set up the project tools	06.03.2025	Ramon Stutz
IMK-4	Set up Infrastructure	Set up the tools and our environment	13.03.2025	Simon Linder
IMK-5	Tooling Evaluation and Selection	Analyze the GitOps tools for our project	03.04.2025	Ramon Stutz
IMK-6	Container via Infrahub	Create container via Infrahub	24.04.2025	Simon Linder
IMK-7	VM via Infrahub	Extend our applications to perform deployment of VMs	15.05.2025	Ramon Stutz
IMK-53	Feature and Deployment	Implementation for some features and deployments	31.05.2025	Simon Linder
IMK-34	Absence	Absence of team members	13.06.2025	Simon Linder / Ramon Stutz

Table 3.1.: Project Phases Overview

3.4.1. Time Table

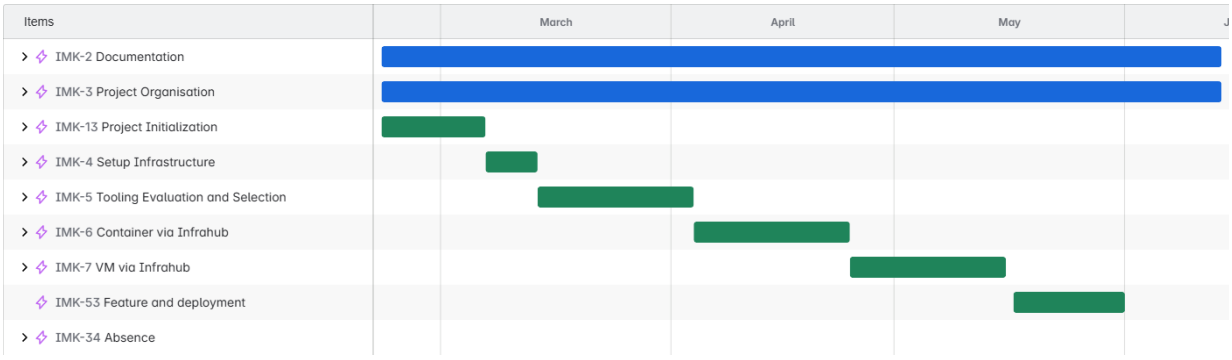

Figure 3.1.: Time Plan of the project

3.5. Risk Management

Like in every other project, risks are always part of it. Our job is to assess, analyze, manage, and if possible, minimize as many risks as possible. Undetected risks could pose a major threat to the success of the project.

3.5.1. Risks

1. **Scope Creep:** Uncontrolled changes or additions to project requirements can lead to scope creep, causing delays or overruns.
2. **Team member falls out:** A team member is absent and isn't capable of working on the project.
3. **Team dynamics:** Poor communication, collaboration issues, or lack of cohesion within the project team can hinder progress and affect project morale.
4. **Quality assurance challenges:** Ineffective testing practices or inadequate quality assurance measures can result in undetected defects, leading to product failures or customer dissatisfaction.
5. **Project management challenges:** Inadequate processes for updating our team project status or updating the timetable can lead to misunderstandings and hinder progress.
6. **Poor Requirements Management:** Inadequate gathering, documentation, or management of project requirements can lead to misunderstandings, rework, and dissatisfaction with the final product.
7. **Technical challenges:** Complex technical requirements, dependencies, or limitations can pose challenges during development, leading to delays or compromised quality.

3.5.2. Risk Countermeasures

Scope Creep

1. Define clear project requirements and objectives from the outset.
2. Regularly review project scope with stakeholders to ensure alignment.

Poor Requirements Management

1. Establish a shared requirements document accessible to all team members.
2. Regularly revisit and refine requirements as the project evolves.
3. Encourage team members to raise ambiguities or uncertainties early for clarification.

Technical challenges

1. Break down complex tasks into smaller, manageable components.
2. Seek input from subject matter experts and consider alternative solutions.
3. Implement a light version of our desired architecture first to ensure functionality between the versions.
4. Allocate sufficient resources and time for addressing technical challenges.

5. Conduct a comprehensive technical feasibility study before project initiation.

Team dynamics

1. Foster open communication and collaboration within the team.
2. Address conflicts and misunderstandings promptly and constructively.
3. Provide opportunities for team-building activities and training.
4. Assign roles and responsibilities clearly to avoid ambiguity.
5. Meet weekly to discuss the tasks and issues.

Quality assurance challenges

1. Develop a comprehensive testing strategy and plan.
2. Don't review your own work; review each other's work.
3. Plan enough time for testing.
4. Do reviews during the project, not all at the end.

Project management challenges

1. Assign a role to a team member who takes time to update the project management tool (Jira).
2. Hold weekly meetings to discuss and assign active and new tasks.

Team member falls out

1. Open conversation and weekly updates on the status of tasks so someone can take over.
2. Properly and up-to-date documentation of the tasks.

3.5.3. Risk Matrix

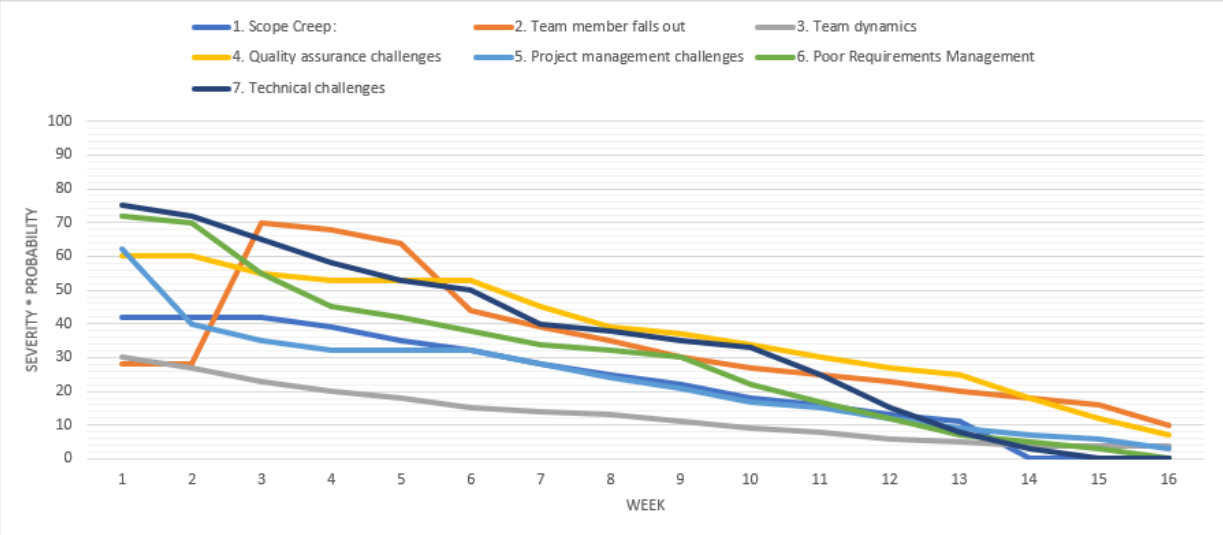

Figure 3.2.: Risk Management

3.5.4. Risk summary

In the first weeks of the project, we identified and assessed potential risks. We took some risk measures to minimize them, but unfortunately one risk occurred in week 3 - team member falls out. In our case, both team members got sick for a week, Ramon for almost 3 weeks. We had to adapt our time plan to finish our project on time.

3.6. Planning Tools

3.6.1. JIRA

We use Jira because all team members are familiar with it, allowing us to manage our projects effectively using Epics, Tasks, and Subtasks. Jira also enables us to create timetables, ensuring clear visibility into project timelines and responsibilities. This familiarity and structured approach enhance collaboration and streamline our project management processes.

3.6.2. Clockify

We use Clockify because it integrates seamlessly with Jira, providing us with easy time tracking on our tasks. Its graphical reports and filter functions allow us to analyze how time is spent on various tasks and promote accountability.

3.6.3. Overleaf

Overleaf is an online collaborative platform that simplifies the process of writing and publishing documents using LaTeX. It allows multiple authors to work simultaneously on a document, making it ideal for academic and technical writing. Overleaf's built-in templates and real-time preview features streamline the formatting process, enabling us to focus on content creation while ensuring high-quality output suitable for publication.

List of Tables

7.1. Python libraries used in the transformation scripts 29
7.2. Naming conventions for Kubernetes kinds 33
7.3. Example of key-based value replacement in a YAML file 33
7.4. GraphQL mutations used in Infrahub 35

3.1. Project Phases Overview 54

1.1. Scenario Refinement Table for Performance Efficiency 63
1.2. Scenario Refinement Table for Compatibility 64
1.3. Scenario Refinement Table for Usability 65
1.4. Scenario Refinement Table for Maturity 66
1.5. Scenario Refinement Table for Fault Tolerance 67
1.6. Scenario Refinement Table for Scalability 68
1.7. Scenario Refinement Table for Testability 69
1.8. Scenario Refinement Table for Reusability 70
1.9. Scenario Refinement Table for Portability 71
1.10. Scenario Refinement Table for Adaptability 72
1.11. Scenario Refinement Table for Installability 73

List of Figures

3.1. C4 Component Diagram, showing the Infrastructure and Code components iv

2.1. Non-Functional requirements 5

4.1. Diagram of the Flux Workflow 8
4.2. Description Workflow Setup Flux Testing 9
4.3. CMP Test: ConfigMap Deployment 10
4.4. ArgoCD and Infrahub Integration Workflow 11
4.5. Infrahub API - Post GraphQL query 12

5.1. C4 Context Diagram 14
5.2. C4 Container Diagram 15
5.3. C4 Component Diagram 16

7.1. Infrahub Resources 24
7.2. Infrahub Schema UML Diagram 25
7.3. Infrahub Python Transformations 28

8.1. Architecture of the self-service frontend 41
8.2. Architecture of the self-service frontend 42

3.1. Time Plan of the project 54
3.2. Risk Management 56

2.1. Description option 1 for Infrahub Schema 74
2.2. Description option 2 for Infrahub Schema 76

Acronyms

API Application Programming Interface. vii, 10, 11, 12, 16, 17, 34, 41, 43, 45, 47, 53

CD Continuous Deployment. iii, 7

CI Continuous Integration. 22

CI/CD continuous integration and continuous deployment. vi, 22, 23, 75

CLI Command Line Interface. 2, 49, 75, 77

CR Custom Resource. 52

CRD Custom Resource Definition. 52

CRUD Create, Read, Update, Delete. 37

CSS Cascading Style Sheets. viii, 80

GUI Graphical User Interface. 15, 25, 75, 76

HTML Hypertext Markup Language. vii, viii, 42, 43, 47, 79, 80

K8s Kubernetes. 52

OCI Open Container Initiative. 8

QAS Quality Attribute Scenarios. 6

SDK Software Development Kit. 10, 49, 52

SSH Secure Shell. 45, 47

UI User Interface. iii, 52

URL Uniform Resource Locator. 43, 44, 47, 79, 80

VM Virtual Machine. 42, 45, 79, 80

YAML YAML Ain't Markup Language / Yet Another Markup Language. vii, 8, 9, 11, 16, 29, 30, 32, 33, 48, 58, 74, 75, 76

Glossary

CMP ArgoCD plugin system to build custom kubernetes manifests with a plugin that can do anything.. 10, 11

DevOps DevOps is a set of practices that combines software development (Dev) and IT operations (Ops). It aims to shorten the systems development lifecycle and provide continuous delivery with high software quality.. iii

GitOps GitOps is a set of practices that uses Git pull requests to manage infrastructure and applications. It is a way to do continuous deployment for cloud-native applications.. ii, iii, 8, 10, 11, 52, 53

infrastructure Infrastructure in our context refers to any Kubernetes native resource or custom resource. These resources can deploy or manage virtual infrastructure like virtual machines, Kubernetes networks, or storage, but also applications and services running on top of that infrastructure. It can even be used to manage physical infrastructure like network devices or servers with projects like kubenet or SDC. The possibilities are huge.. i, ii, iii, 20

PEP8 PEP8 is a style guide for Python code. It is a set of rules that specify how to format Python code for maximum readability.. 22

Part III.

Appendix

1. Quality Attribute Scenarios

This chapter presents the quality attribute scenarios for the project, focusing on ISO/IEC 25000 quality attributes. Each scenario is refined to provide a clear understanding of the system's behavior under specific conditions.

Scenario Refinement for Performance Efficiency	
Scenario(s)	Operator synchronizes resources from Infrahub, filtering only the necessary data based on the current cluster state.
Business Goals	Minimize compute and network overhead to reduce cost and improve scalability.
Relevant Quality Attributes	Performance Efficiency, Scalability, Resource Optimization
Scenario Components	
Stimulus	Synchronization of a large number of Infrahub Artifacts is triggered.
Stimulus Source	Reconciliation loop and a change in only one Infrahub Artifact.
Environment	Normal operational state in a constrained Kubernetes cluster.
Artifact	Vidra operator and Infrahub client logic.
Response	The operator only fetches required artifacts and applies only the diffs.
Response Measure	CPU/memory/network usage per sync stays under threshold (<1% RAM increase, <50MB transferred). Sync completes within 10s.
Questions	
What is the minimum data subset required for a successful reconciliation? How can diffing and filtering be optimized?	
Issues	
Infrahub Artifact might not support fine-grained queries; risk of over-fetching data. However, this can be mitigated in Kubernetes by applying only the changed resources.	

Table 1.1.: Scenario Refinement Table for Performance Efficiency

Scenario Refinement for Compatibility	
Scenario(s)	The system integrates with Kubernetes and Infrahub API, enabling seamless interaction and interoperability.
Business Goals	Ensure seamless integration and interoperability with Kubernetes and Infrahub API.
Relevant Quality Attributes	Compatibility, Interoperability, Extensibility
Scenario Components	
Stimulus	Integration with a new version of Kubernetes or Infrahub API is required.
Stimulus Source	Update or change in Kubernetes or Infrahub API.
Environment	Normal operational state in a Kubernetes cluster using Infrahub API.
Artifact	Vidra operator and Infrahub client logic.
Response	The system continues to interoperate with Kubernetes and Infrahub API without requiring major changes.
Response Measure	Integration is achieved with minimal to no configuration and no disruption to existing functionality.
Questions	
How easily can the system be integrated with new versions of Kubernetes or Infrahub API? Are there any compatibility issues?	
Issues	
Kubernetes and Infrahub API do not change frequently, so compatibility issues are rare. However, monitoring for upstream changes is still necessary.	

Table 1.2.: Scenario Refinement Table for Compatibility

Scenario Refinement for Usability	
Scenario(s)	A user deploys resources using the system's interface without prior Kubernetes expertise and encounters an error.
Business Goals	Lower the barrier to entry for resource deployment and reduce support overhead by making the system intuitive and reporting errors clearly.
Relevant Quality Attributes	Usability
Scenario Components	
Stimulus	A user attempts to deploy a resource and makes a configuration mistake.
Stimulus Source	End user with limited Kubernetes knowledge.
Environment	Normal operational state, accessed via the system's user interface.
Artifact	User interface, error handling in the Kubernetes resource, and feedback mechanisms for admin.
Response	The system displays a clear, actionable error message and logs detailed error information in the Kubernetes resource for the admin, enabling easy administrative troubleshooting.
Response Measure	The admin can easily access detailed logs for troubleshooting.
Questions	
How intuitive is the deployment workflow for new users? Are error messages understandable and actionable?	
Issues	
Users may still require some domain knowledge for complex deployments. Continuous user feedback is needed to improve usability and error guidance.	

Table 1.3.: Scenario Refinement Table for Usability

Scenario Refinement for Maturity	
Scenario(s)	The system processes a high volume of deployment requests over an extended period, with continuous deployment triggered only when actual changes are detected.
Business Goals	Achieve high stability and reliability, minimize defects, and ensure efficient resource usage by avoiding unnecessary deployments.
Relevant Quality Attributes	Maturity, Reliability, Performance Efficiency
Scenario Components	
Stimulus	A large number of deployment requests are received, but only a subset involve actual changes to resources.
Stimulus Source	User-initiated deployment triggers (such as new synchronization to Infrahub with multiple Artifacts).
Environment	Normal and peak operational states in a Kubernetes cluster.
Artifact	Deployment controller and change detection logic.
Response	The system only initiates deployments when changes are detected, maintains stable operation, and logs any errors or failures for review.
Response Measure	Defect rate remains below 0.5% per deployment; unnecessary deployments avoided; no significant performance degradation under load.
Questions	
How does the system detect and avoid redundant deployments? What mechanisms are in place to ensure stability and low defect rates during high-volume operations?	
Issues	
False positives in change detection could trigger unnecessary deployments. Monitoring and robust error handling are required to maintain maturity and reliability.	

Table 1.4.: Scenario Refinement Table for Maturity

Scenario Refinement for Fault Tolerance	
Scenario(s)	The system encounters a failure during synchronization with Infrahub or while applying resources to Kubernetes.
Business Goals	Ensure continuous operation and prevent data loss or significant disruption in the event of transient or persistent failures.
Relevant Quality Attributes	Fault Tolerance, Reliability
Scenario Components	
Stimulus	A network partition, Infrahub API outage, or Kubernetes API error occurs during a sync or deployment operation.
Stimulus Source	External system failures, network interruptions, or transient errors in Infrahub or Kubernetes.
Environment	Normal and degraded operational states in a Kubernetes cluster with Infrahub integration.
Artifact	Vidra operator, Infrahub client, and Kubernetes controllers.
Response	The system automatically retries failed operations using Kubernetes retry logic and exponential backoff for Infrahub API calls, ensuring eventual consistency and recovery without data loss.
Response Measure	No data loss; operations are retried up to a defined limit; system resumes normal operation within 1 minute after transient failure resolution.
Questions	
How does the system detect and recover from failures? Are retry and backoff strategies sufficient to prevent data loss and minimize disruption?	
Issues	
Persistent failures may require manual intervention. Excessive retries could delay recovery or cause resource contention.	

Table 1.5.: Scenario Refinement Table for Fault Tolerance

Scenario Refinement for Scalability	
Scenario(s)	The system experiences a rapid increase in the number of deployments and concurrent users.
Business Goals	Maintain consistent performance and responsiveness as the number of deployments and users grows.
Relevant Quality Attributes	Scalability, Performance Efficiency, Resource Optimization
Scenario Components	
Stimulus	A spike in deployment requests or user activity occurs, such as during a massive synchronization of Infrahub Artifacts or a large number of users accessing the system simultaneously.
Stimulus Source	Multiple users or automated systems initiating deployments simultaneously.
Environment	Normal and peak operational states in a Kubernetes cluster.
Artifact	Deployment controller, resource manager, and supporting infrastructure.
Response	The system dynamically allocates resources and manages workloads to handle increased demand without significant performance degradation.
Response Measure	Throughput and response times remain within acceptable thresholds (e.g., <10% increase in latency, <5% error rate) as load increases.
Questions	
How does the system scale with increased deployments and users? Are there bottlenecks that limit scalability?	
Issues	
Resource contention or architectural bottlenecks may limit scalability. Monitoring and autoscaling strategies are required to ensure consistent performance.	

Table 1.6.: Scenario Refinement Table for Scalability

Scenario Refinement for Testability	
Scenario(s)	Developers write unit tests for the operator's reconciliation logic and API interactions.
Business Goals	Facilitate rapid, reliable verification of functionality and performance to support maintainability and reduce defects.
Relevant Quality Attributes	Testability, Maintainability
Scenario Components	
Stimulus	A new feature or bug fix is implemented, requiring validation through automated tests.
Stimulus Source	Developer or CI/CD pipeline.
Environment	Local development or CI/CD test environment.
Artifact	Operator logic, API client, and test harness.
Response	The system supports isolated, repeatable unit tests with clear pass/fail outcomes.
Response Measure	>80% code coverage; tests execute in <2 minutes; failures are easily diagnosed.
Questions	
How easily can new features be tested? Are tests reliable and fast?	
Issues	
Complex dependencies on Kubernetes may make some logic hard to test in isolation. Mocking, modularization, and envtest are required.	

Table 1.7.: Scenario Refinement Table for Testability

Scenario Refinement for Reusability	
Scenario(s)	Developers create reusable templates or modules for common resource types and deployment patterns.
Business Goals	Accelerate development and reduce duplication by enabling reuse of components and templates.
Relevant Quality Attributes	Reusability, Maintainability, Efficiency
Scenario Components	
Stimulus	A new project or resource is created that can leverage existing templates or modules.
Stimulus Source	Developer or DevOps engineer.
Environment	Development or deployment environment.
Artifact	Templates, modules, or shared libraries.
Response	Existing components are reused with minimal modification, reducing development time and errors.
Response Measure	>50% of new resources use shared templates; time to deploy is reduced by >30%.
Questions	
How often are components reused? How easy is it to adapt templates for new use cases?	
Issues	
Overly generic templates may be hard to maintain; a balance between flexibility and simplicity is needed.	

Table 1.8.: Scenario Refinement Table for Reusability

Scenario Refinement for Portability	
Scenario(s)	The system is deployed to a new Kubernetes cluster or cloud environment with minimal changes.
Business Goals	Enable deployment in diverse environments without significant code changes, supporting multi-cloud and hybrid strategies.
Relevant Quality Attributes	Portability, Flexibility
Scenario Components	
Stimulus	A customer or team requests deployment to a new cluster or cloud provider.
Stimulus Source	Customer, DevOps, or platform engineer.
Environment	New Kubernetes cluster or cloud environment.
Artifact	Deployment manifests, configuration files, and operator code.
Response	The system is deployed successfully with only environment-specific configuration changes.
Response Measure	Deployment to a new environment requires <1 hour of effort and no code changes.
Questions	
How portable is the system across clusters and clouds? What changes are required for deployment in a new environment?	
Issues	
Environment-specific dependencies may hinder portability; use of standard APIs and configuration is recommended.	

Table 1.9.: Scenario Refinement Table for Portability

Scenario Refinement for Adaptability	
Scenario(s)	The system is deployed in a Kubernetes environment with custom configurations.
Business Goals	Support diverse deployment scenarios and customer requirements without code changes.
Relevant Quality Attributes	Adaptability, Flexibility, Maintainability
Scenario Components	
Stimulus	A customer requests deployment in a Kubernetes cluster with non-standard configuration or policies.
Stimulus Source	Customer, DevOps, or platform engineer.
Environment	Kubernetes clusters with varying configurations and policies.
Artifact	Deployment manifests, configuration files, and operator code.
Response	The system is configured and deployed successfully by adjusting only configuration files or manifests, without code changes.
Response Measure	Deployment in a new environment requires <30 minutes of configuration work and no code changes.
Questions	
How easily can the system be adapted to different Kubernetes configurations? Are there any hardcoded dependencies?	
Issues	
Overly rigid configuration or hardcoded values may limit adaptability. Documentation and parameterization are essential.	

Table 1.10.: Scenario Refinement Table for Adaptability

Scenario Refinement for Installability	
Scenario(s)	A new user installs the system in a fresh Kubernetes cluster following the provided documentation.
Business Goals	Minimize installation time and errors, enabling rapid onboarding and adoption.
Relevant Quality Attributes	Installability, Usability, Maintainability
Scenario Components	
Stimulus	A user attempts to install the system using the official documentation and installation scripts.
Stimulus Source	New user, DevOps, or administrator.
Environment	Fresh Kubernetes cluster with standard prerequisites.
Artifact	Installation scripts, manifests, and documentation.
Response	The system is installed and operational with minimal manual steps and no errors.
Response Measure	Installation completes in <10 minutes; user reports no blocking issues; documentation is rated as clear and sufficient.
Questions	
How easy is the installation process for new users? Are all dependencies and steps clearly documented?	
Issues	
Missing or outdated documentation and complex dependencies may hinder installability. Regular updates and user feedback are needed.	

Table 1.11.: Scenario Refinement Table for Installability

2. Detailed Architecture Decisions

2.1. What is Infrahub used for?

We had to decide if Infrahub should be a place for an Administrator to manage the Kubernetes Cluster or if it should be a place for a Developer to deploy their applications. A solution for an Administrator would include many small components (one for each kind in Kubernetes) which could be custom-fitted to the needs of any application. A solution for a Developer would include predefined templates for common applications like a web app, VM, or database with only the most important values to customize a predefined solution.

Option 1 - Infrahub for Administrators In this approach, every Kubernetes Kind, such as Deployments, Services, and Ingress, is represented as a distinct entity within Infrahub, generating a separate YAML file as an artifact for each object. This method offers significant flexibility, particularly for administrators, as they retain full control over the structure and configuration of deployments. It allows for precise customization of each Kubernetes Kind, ensuring that individual requirements can be met with ease.

However, this granular approach comes with notable challenges. The sheer volume of generated data within Infrahub may become substantial, potentially leading to increased complexity in management. Additionally, a dedicated script or plugin would be required to establish clear associations between the various YAML files and their respective manifests, ensuring seamless integration and maintainability.

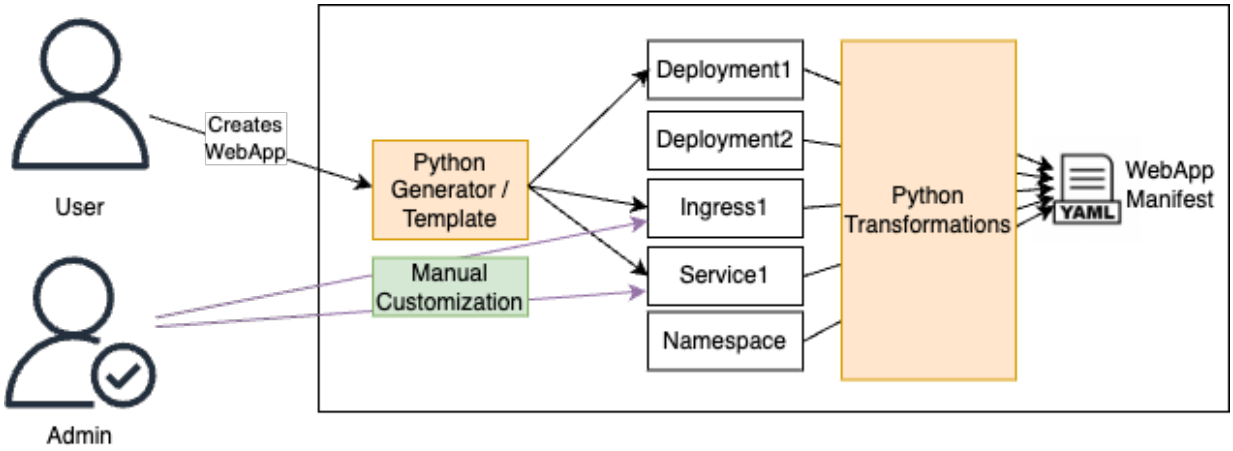

Figure 2.1.: Description option 1 for Infrahub Schema

Pros:

- The Administrator can customize the Cluster to their needs.
- Small components can be used in different solutions, like one component for an nginx Ingress, one for a deployment, and one for a service for a web application.
- The Administrator can use Infrahub to manage Kubernetes and always see the current state of the Cluster and each Kubernetes kind.
- This approach offers granular control, enabling tailored modification of each individual Kubernetes kind as needed.

- It promotes a modular structure, simplifying maintenance and updates for individual components.
- It supports advanced configurations and custom solutions, granting administrators deep insight into the underlying infrastructure.

Cons:

- Each component needs to generate its own YAML artifact and we need a way to stitch them together into one bigger solution later on. This could be done with Helm or Kustomize or by registering the different components together as one application in the CI/CD tool which we utilize (Flux, ArgoCD, or a custom Operator).
- Infrahub would be full of many small components which could be hard to manage, maintain and could lead to a lot of artifact files and a messy artifact section in Infrahub.
- We need to map many or almost all fields of the Kubernetes API to the GUI and the YAML artifact, which could be difficult. One could try to generate pydantic models from the Kubernetes API and try to use them to generate the Infrahub schema, but this could lead to a lot of conflicts and manual work as there is no standardized way to generate the schema from the Kubernetes API or pydantic models yet.
- Infrahub would be more complex and harder to use for a Developer who just wants to deploy their application as they probably do not know what kind of Ingress, Service, Deployment, etc. they need.
- We would basically build a GUI for YAML files, where the Administrator can click together their solution. But the GUI could actually be less powerful and customized than YAML files and the Administrator who already knows Kubernetes would probably prefer to use the CLI or the Kubernetes Dashboard.
- Kubernetes Dashboard is already a good tool for Administrators to manage the Cluster and see the current state of the Cluster and each Kubernetes kind. So Infrahub would be a duplicate of the Kubernetes Dashboard with a GUI for YAML files. And it would be hard to compete with a tool that is already integrated in Kubernetes and is actively developed by the Kubernetes community, which is perfectly adapted and integrated into Kubernetes.

Option 2 - Infrahub for Developers and Users This approach fundamentally shifts the responsibility of defining Kubernetes configurations from Infrahub to the administrator. Instead of storing all deployment details within Infrahub, the administrator creates a set of predefined YAML manifest templates for common services such as databases or web applications. These templates include fixed parameters, such as the number of services, deployments, and other structural elements, that remain unchanged across instances.

However, certain configurable parameters, such as container images or hardware limitations, are stored within Infrahub, allowing for customization where necessary. When generating a final manifest, a Python-based transformation process selects the appropriate YAML template and injects the required dynamic values from Infrahub, ensuring that each deployment is tailored to its specific context.

One key advantage of this method is that Infrahub does not need to store complete manifest information, reducing complexity and data volume. Additionally, the Python transformation process is relatively simple to implement, streamlining deployment automation.

On the downside, this approach limits flexibility, as a new template must be created for every unique service deployment. Furthermore, the rigid structure of predefined manifests means that administrators have less freedom to fine-tune individual Kubernetes resources dynamically.

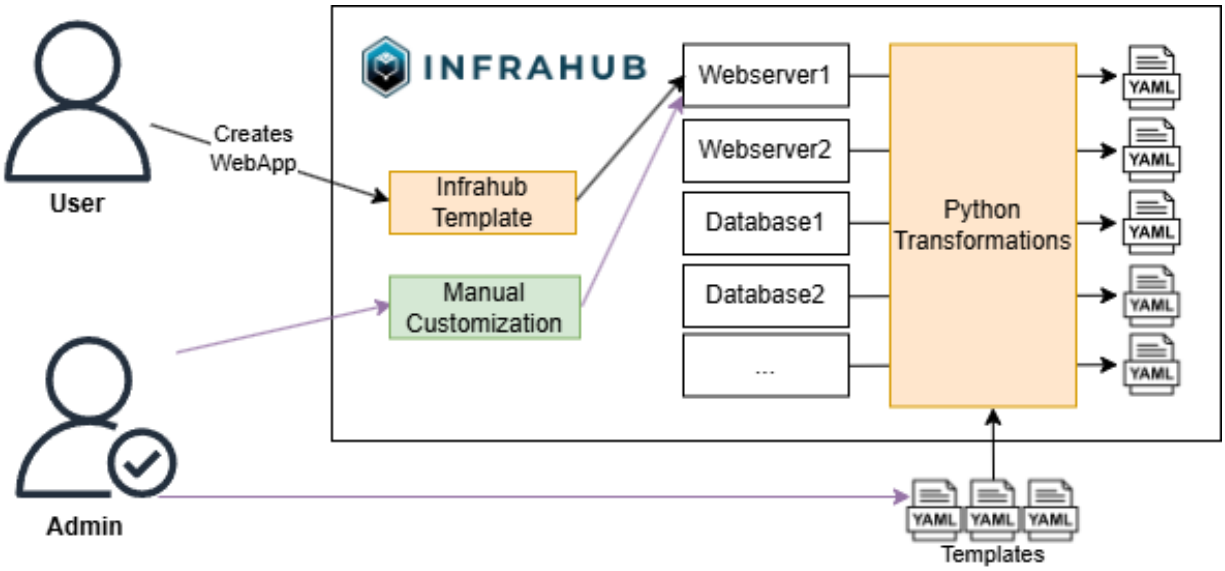

Figure 2.2.: Description option 2 for Infrahub Schema

Pros:

- The Developer / User can deploy their application with only a few clicks and does not need to know Kubernetes or what kind of Ingress and other resources they need for their web application.
- Infrahub would be small and easy to use. The User would only need to select a template, fill in the values, and deploy the application. It minimizes the risk of misconfiguration by restricting editable parameters to safe and proven values.
- Each solution would be tested and work out of the box. The User is only given configuration which would not break the solution (web application, VM, etc.).
- Each solution would generate only one YAML artifact file which would be easy to manage and maintain as everything is in one file.
- It enables faster iteration and prototyping of solutions as standardized templates can be easily replicated.
- It supports consistency by ensuring each deployment follows the same structure, reducing errors.
- It encourages a user-friendly interface where essential configurations are exposed while hiding complex details. More advanced or critical values (like CPU, RAM limitations) could be exposed just to the Administrator role within Infrahub.
- We develop a tool with benefits for the User and not a duplicate of the Kubernetes Dashboard.

Cons:

- The User cannot customize the solution as much as they could with the Administrator solution. The User can only fill in the values which are exposed in the GUI.
- The User cannot deploy their own solutions, only predefined solutions which are provided by the Administrator.

- The Administrator needs to create a template for each solution which the User can deploy like a web app, VM, or database.
- The User may face limitations if their application requires non-standard Kubernetes configurations not covered by existing templates.
- Troubleshooting issues needs to be done on Kubernetes and could be harder since deeper Kubernetes knowledge is abstracted away.
- Adding new application types requires Administrator intervention to create or maintain templates.

Decision We decided that Infrahub should be a place for a Developer or other User with little knowledge about Kubernetes to deploy their own applications. The Administrator should use the Kubernetes Dashboard or the Kubernetes CLI to manage the Cluster.

Justification We do not want to build a duplicate of the Kubernetes Dashboard and want to keep Infrahub small and organized. We want to provide a tool that solves a problem that is not solved yet.

Future Enhancements An additional approach could involve blending administrator and developer needs by offering both granular control for advanced users and simplified presets for typical use cases. This might include:

- A parameter switch for toggling between a guided template mode and a comprehensive configuration mode.
- Role-based access that displays or hides complex settings based on user permissions.
- Reusable components stored in a dedicated library to speed up new template creation.

Implementation Outline In order to realize the chosen option, a few steps are required:

1. Identify reusable application templates (web app, VM, database).
2. Define key configuration parameters for each template.
3. Streamline the user interface to expose only these parameters.
4. Verify compatibility with the existing Kubernetes environment.
5. Provide documentation for further customization and troubleshooting.

Next Steps Future updates should focus on:

- Adding more application templates based on user feedback.
- Improving validation for critical parameters.
- Enhancing observability and logs for production deployments.

2.2. Do we need Profiles or Templates in Infrahub?

We had to decide if we should use Profiles or Templates in Infrahub.

Option 1 - Template Profiles on the other hand store the value of certain attributes, and objects that are using this profile inherit its value.

Pros:

- Profiles allow shared attributes, making it easy for the Admin to change a value for multiple objects.
- Profiles are allowed to be used by multiple objects defined individually.
- Profiles can be added after the object is already created.

Cons:

- Profiles can't define attributes that are required.
- Profiles can't be set on attributes that have to be unique.

Option 2 - Object Template feature allows creating a reusable blueprint for any object. We could use templates with default values for the User to deploy their application.

Pros:

- Easy to use to create one object.
- Easy to create in Infrahub with a GraphQL query.
- Can define every attribute.

Cons:

- Can't use Resource Manager in the template.
- Doesn't support shared attributes. On a change, the Admin has to change the value on each object.

Decision We decided to use Templates instead of Profiles, because our schema defines most attributes to be required and as well the combination of the name and namespace needs to be unique. Therefore, Profiles can't be used for our use case.

Future Enhancements An additional approach could use both Templates and Profiles, so for the object creation we use a Template with links to some attributes in the Profile.

- A Template to use for object creation.
- Profiles which define default values, easy for the admin to change.

Implementation Outline In order to realize the chosen option, a few steps are required:

1. Identify reusable application templates (web app, VM, database).
2. Define key configuration parameters for each template.
3. Streamline the user interface to expose only these parameters.
4. Verify compatibility with the existing Kubernetes environment.
5. Provide documentation for further customization and troubleshooting.

Next Steps Future updates should focus on:

- Adding more flexibility in the creation of the objects.
- Improving validation for critical parameters.

3. Detailed HTML and CSS Code

3.1. HTML

Metadata

At the beginning of the document, essential metadata is defined—including character encoding, responsive design settings, and a link to the CSS file:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Resource Request</title>
    <link rel="stylesheet" href="{{ url_for('static', filename='style.css') }}">
```

JavaScript Logic

JavaScript is used to dynamically adapt the form fields depending on the selected resource type.

Dynamic Fields The `toggleFields()` function ensures that only the relevant fields for the selected deployment type are displayed. If "VM" is selected, the form shows the container image URL field; if “Webserver” is selected, it shows a Docker image selection instead.

```
<script>
    function toggleFields() {
        const deploymentType = document.getElementById("deployment_type").value;
        const urlField = document.getElementById("url_field");
        const imageField = document.getElementById("image_field");

        urlField.style.display = "none";
        imageField.style.display = "none";

        if (deploymentType === "VM") {
            urlField.style.display = "block";
        } else if (deploymentType === "Webserver") {
            imageField.style.display = "block";
        }
    }
    window.onload = toggleFields;
```

Button Deactivation To prevent multiple submissions during processing, the submit button is disabled once clicked:

```
    function disableButton() {
        const btn = document.getElementById("submit-btn");
        btn.disabled = true;
        btn.innerText = "Please wait...";
    }
</script>
```

Form Structure

The form is submitted via `POST` and contains the following fields:

- **Resource Type:** A dropdown menu to choose between `Webserver` and `VM`.
- **Name:** A text input with validation (e.g., character length).
- **Description:** An optional text field (max 50 characters).
- **Namespace:** Kubernetes namespace (required).
- **VM URL Field:** Shown only when "VM" is selected—allows choosing a container image (e.g., Ubuntu, Fedora).
- **Webserver Image Field:** Shown only when "Webserver" is selected—allows choosing between Docker images like Apache or Nginx.

Form Excerpt (HTML)

The following snippet shows a part of the request form:

Listing 3.1: Excerpt from the HTML Form

```
<form method="POST" onsubmit="disableButton()">
    <label for="deployment_type">Resource:</label>
    <select name="deployment_type" id="deployment_type" onchange="toggleFields()">
        <option value="Webserver" selected>Webserver</option>
        <option value="VM">VM</option>
    </select>
    ...
</form>
```

3.2. CSS Styling

The visual appearance of the web interface is defined in a separate CSS file. It ensures a clean, modern, and responsive design that improves the user experience and aligns with the functionality of the dynamic form.

Global Styles

The `body` tag is styled to use a sans-serif font and centers all content both vertically and horizontally using Flexbox. It also sets a soft background color to maintain a clean look.

Listing 3.2: Global Body Styling

```
body {
    font-family: 'Segoe UI', Tahoma, Geneva, Verdana, sans-serif;
    background-color: #f4f7f8;
    color: #333;
    display: flex;
    justify-content: center;
    align-items: center;
    flex-direction: column;
}
```

Form Layout

The form element is given a fixed width and height (both set to 60% of the container) to keep it centered and well-proportioned on most screens.

```
form {
    width: 60%;
    height: 60%;
}
```

Inputs and Form Controls

All input elements, buttons, dropdowns, and labels share consistent styling to ensure visual harmony. They are sized for high readability and accessibility.

```
input, button, select, label {
    width: 100%;
    font-size: 30px;
    height: auto;
}

input {
    padding: 5px;
}
```

Headings

To emphasize titles and section headers, the font size for `<h1>` and `<h2>` elements is significantly increased:

```
h1, h2 {
    font-size: 40px;
}
```

Buttons

The submit button is styled with a soft blue background and padding for better interactivity. On hover, it changes color to provide visual feedback to the user.

```
button {
    margin-top: 5%;
    width: 100%;
    padding: 5px;
    background-color: lightskyblue;
}

button:hover {
    background-color: lightblue;
}
```

Paragraphs

Paragraph text is displayed in a larger font for better readability:

```
p {
    font-size: 30px;
}
```

Image Container and Styling

The container used for displaying images (e.g., logos) uses Flexbox to align images side by side horizontally. Images are resized and spaced evenly with rounded corners.

```
container {
    display: flex;
    flex-direction: row;
    width: 100%;
    align-items: center;
}

img {
    width: 25%;
    margin-left: 15%;
    margin-right: 10%;
    border-radius: 5%;
}
```