Bachelor of Science (Hons) Applied Computing

Heimdall - A Kubernetes Extension

Author:
Bryan Keane

Supervisor:
Lucy White

Contents

1 Plagiarism Declaration **4**

2 Acknowledgements **5**

3 Introduction **6**
- 3.1 Background . . . 6
- 3.2 Motivation . . . 6
- 3.3 Problem Statement . . . 7
  - 3.3.1 Industry Example . . . 7
- 3.4 Aims and Objectives . . . 8

4 Design **10**
- 4.1 Requirements . . . 10
  - 4.1.1 Functional Requirements . . . 10
  - 4.1.2 Non-Functional Requirements . . . 10
- 4.2 Architecture Model . . . 11
- 4.3 System Design . . . 12
  - 4.3.1 Controller Overview . . . 12
  - 4.3.2 Unstructured Package Implementation . . . 13
  - 4.3.3 Slack Integration . . . 13
  - 4.3.4 Role Based Access Control . . . 13
  - 4.3.5 Installation . . . 13
- 4.4 User Stories . . . 14
- 4.5 Risk Analysis . . . 14

5 Methodology **14**
- 5.1 Agile and Scrum . . . 14
- 5.2 Scrum Artifacts . . . 15
- 5.3 Version Control . . . 15
- 5.4 Continuous Integration . . . 15
- 5.5 Continuous Delivery . . . 15
- 5.6 Testing Approach . . . 15
- 5.7 Open Source . . . 15

6 Technologies **15**
- 6.1 Kubernetes . . . 15
  - 6.1.1 Resources . . . 16
  - 6.1.2 Controllers . . . 17
  - 6.1.3 Operators . . . 17
  - 6.1.4 Role-Based Access Control . . . 18
- 6.2 Languages . . . 18
  - 6.2.1 Go . . . 18
  - 6.2.2 YAML . . . 19
- 6.3 Libraries . . . 19
  - 6.3.1 Slack Go Client . . . 19

6.3.2 Unstructured Package 19

7 Tools **19**
7.1 Overleaf 20
7.2 Operator SDK 20
7.3 Minikube 20
7.4 Docker 20
7.5 Kubectl 21
7.6 Git and GitHub 21
7.7 GitHub Actions 21
7.8 Jira 21

8 Proof of Concept **22**
8.1 Slack Prototype 22
8.2 Watcher Prototype 22
8.3 Undecided Prototype 22

9 Summary **22**
9.1 Review 22
9.2 Semester Two Outline 22

List of Figures

1 *Model of The Problem - Two Operators Fighting Over a Resource's State* 7
2 *Simplified Model of Moodle, Tutors, and MySQL Industry Example* 8
3 *Model of the solution: Heimdall* 9
4 *Heimdall system architecture overview model* 12
5 *Kubernetes application Resources: Deployments, Replica Sets, and Pods* 16
6 *Example Core Resource (Deployment) comparison with Example Custom Resource* . . . 17
7 *Operator Pattern model of Custom Controller and Resources interaction* 18
8 *Operator SDK Controller scaffolding via Command Line* 20

1 Plagiarism Declaration

I declare that this material, which I now submit for assessment, is entirely my own work and has not been taken from the work of others, save and to the extent that such work has been cited and acknowledged within the text of my work. I understand that plagiarism, collusion, and copying are grave and severe offences in the university and accept the penalties that would be imposed should I engage in plagiarism, collusion or copying. I have read and understood the Assignment Regulations. I have identified and included the source of all other facts, ideas, opinions, and viewpoints in the assignment references. Direct quotations from books, journal articles, internet sources, module text, or any other source are acknowledged and the source cited are identified in the bibliography. This assignment, or any part of it, has not been previously submitted by me or any other person for assessment on this or any other course of study.

2 Acknowledgements

I would like to express my deepest appreciation to my Mentors Ciaran Roche and Laura Fitzgerald who provided invaluable experience and contributions to this project. I would also like to extend my deepest gratitude to my Project Supervisor Lucy white for guiding and encouraging me throughout the process.

3 Introduction

This project, Heimdall, is an open-source Kubernetes Extension built for multi-operator Kubernetes environments. Implementing Heimdall will allow developers to configure it to watch resources of their choosing in a Kubernetes cluster. Heimdall will prevent any unwanted Operators from changing that resource, while simultaneously sending an interactive notification to the developer via a pre-configured Slack Channel regarding the issue.

3.1 Background

In 2022 I completed an 8-month internship at Red Hat for my 3rd-year work placement. I worked on the Red Hat OpenShift API Management (RHOAM) team. RHOAM utilises multiple OpenShift Operators, which are automatically managed and configured applications, to provide a comprehensive API solution to its customers. The API management features provided by RHOAM include: [Red Hat Developer, 2020]

- Limiting the number of API requests based on the customer's quota
- Creating security policies to manage API access
- Monitoring API health
- An API portal for sign-up and documentation

While working on this team, I had the opportunity to contribute to applications which utilize various technologies like Kubernetes, OpenShift, and Go Operators.

3.2 Motivation

Throughout my internship, I ran into a number of technical issues. One such issue centred around RHOAM's rate-limiting service which has become the motivation for my final year project, Heimdall. The issue in question was with the port through which API requests were being sent in order to be rate limited. That port was being overwritten by a conflicting Operator who had a different desired state for that resource. Rate limiting in this context refers to the maximum number of API calls allowed during a specified time period [IBM API-Connect, 2022]

RHOAM uses the Marin3r Operator to provide rate-limiting for RHOAM customers. It works by injecting a rate-limiting container into a Pod. The container then acts as a middleman for API requests by only redirecting requests to the destination container if the API request limit has not been reached. A more detailed explanation of this process will be discussed in Section 6.

It took several days of debugging to figure out what was going wrong and another week passed before my fix was merged. This meant that this problem had been occurring on customer clusters before my fix was rolled out to production. After speaking to developers on various Red Hat teams, it became apparent that this problem was not unique to our team and in fact, was an issue that all Kubernetes and OpenShift product teams face. Fixing this will not only benefit Red Hat teams but any team working on a Kubernetes-based application that utilizes Operators.

3.3 Problem Statement

In Kubernetes (K8s), resources (or objects) have a desired state which describes what the actual state of that resource should look like. The desired state can either lie in a YAML definition or in the code of a Controller. Operators and Controllers can watch that resource's actual state and continuously compare it with the desired state. If they do not match then it is the job of that Operator or Controller to make the necessary changes in order to synchronise them [Operator Pattern, 2022].

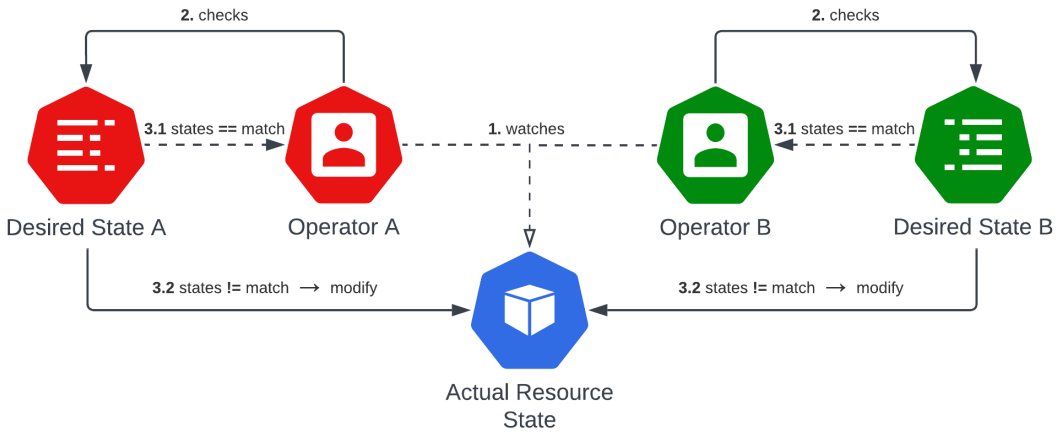


Figure 1: *Model of The Problem - Two Operators Fighting Over a Resource's State*

This pattern works well until two Operators with conflicting desired states are both set to reconcile a resource. This will cause the resource's state to continuously change as both Operators attempt to synchronise its actual state with their unique desired state. The model shown in Figure 1 shows this problem in action. Each Operator will do the following:

1. Watch a resource's actual state.
2. Check the desired state for that resource.
3. Compare the actual state with the desired state and
    3.1. *If they match* then return to Step 1. and repeat.
    3.2. *If they don't match* then modify the resource's actual state.
4. Repeat this process continuously.

3.3.1 Industry Example

As an example, the open-source e-learning platform Moodle can be used. In an educational environment, one might have a Moodle Operator and a MySQL Operator to serve as Moodle's database. The MySQL Operator manages the storing, of course, student, and module information. This application may then install a Tutors Operator, which is another e-learning platform which houses course notes and lab work. Since there is already a MySQL database via the MySQL Operator, Tutors can use the same database to store course notes and lab work.

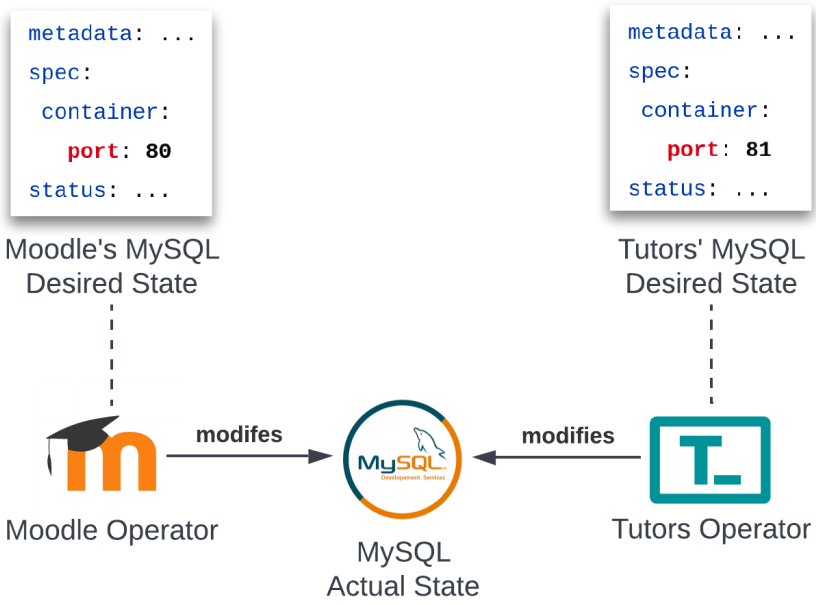


Figure 2: *Simplified Model of Moodle, Tutors, and MySQL Industry Example*

In this example, it would be very easy to misconfigure one of the Operators to have a dissimilar desired state for the MySQL resource. This would cause both Operators to constantly change that resource. Imagine Moodle wanted the port in which database access occurred to be port 80 and Tutors wanted the database access to occur through port 81. This would cause the MySQL resource's access port to be changed back and forth. If a lecturer or student attempts to access information through Moodle, but at that point in time the MySQL resource's port was configured by the Tutors Operator, the user would not be able to retrieve the data.

3.4 Aims and Objectives

The solution is to create a custom Kubernetes controller which will monitor resource states. A Kubernetes Operator can create a resource, become its owner, and will set the controller to watch that resource for changes. If another operator changes the resource the controller will trigger an alert, notify the developer via a slack integration and allow the developer to fix the problem without the need for time-consuming debugging.

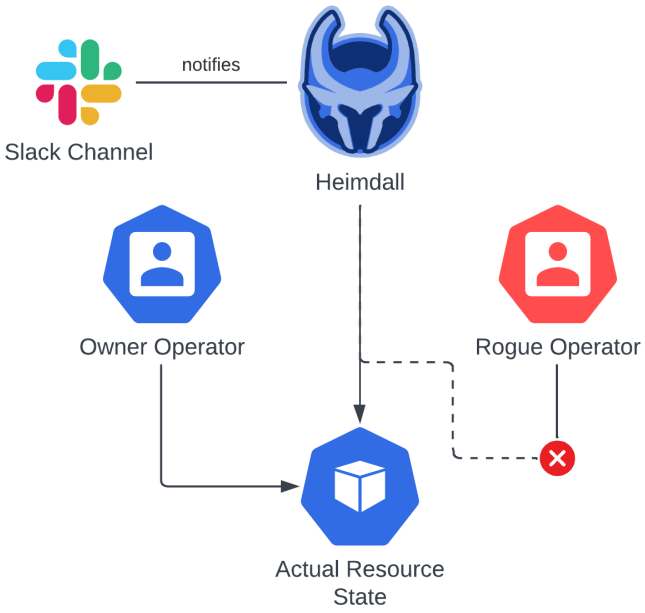


Figure 3: *Model of the solution: Heimdall*

Figure 3 models the proposed solution where the Owner Operator and Rogue Operator are attempting to change a resource's actual state. Once the Owner Operator is installed, it creates the resource with the addition of a label that Heimdall is looking for. Heimdall sees a new resource created with this label and begins monitoring its state. The Rogue Operator is installed and begins changing the resource.

The minimum viable product for Heimdall involves the controller watching for non-owners changing the state of a resource. It will then generate an interactive notification for slack with details on the issue to allow the developers to find and fix the problem with relative ease. The stretch goals for Heimdall will achieve the following:

1. Allow for an atomic owner of a resource to be set.
2. Block changes to resources from non-owners.

This will not only allow the developer to fix the problem with ease but also stop the issue from occurring and prevent any downtime for the end user. These will be discussed further in Section **??**.

4 Design

This section will begin by outlining the functional and non-functional requirements that the controller must fulfil in order to be effective, including the ability to install on any Kubernetes environment, connect to Slack for notifications, block changes to resources from unwanted operators, and provide supporting documentation. Next, an overview of the architecture of Heimdall will be presented to give a deeper understanding of how the technical components of the Controller will operate. Lastly, each aspect of the project will be discussed in depth in order to fully understand the project and its components.

4.1 Requirements

There are various requirements for the successful implementation of Heimdall. These requirements can be divided into two categories: functional requirements and non-functional requirements. Functional requirements describe the specific capabilities or features that the system must have, while non-functional requirements describe constraints that the system must adhere to [P. Gorbachenko, 2021]. It is important to carefully define both types in order to ensure that Heimdall meets the needs of its users and operates in a reliable manner. These will serve as a guide for the design and development of the Controller and will help to ensure that it is able to effectively detect and resolve conflicting Resource changes between Operators in a Kubernetes cluster.

4.1.1 Functional Requirements

Functional requirements are features and capabilities that the system must possess in order to meet the needs and expectations of the user. In the case of Heimdall, these requirements are primarily focused on the functionality that will be most useful to developers implementing the controller into their environments, as well as site reliability engineers installing it onto customer Kubernetes clusters. Heimdall's Functional Requirements are as follows:

1. Developers must be able to install the controller on any Kubernetes cluster.
2. Developers must be able to configure the controller to connect to Slack for notifications.
3. Developers must be able to define an Operator that is unable to make changes to the specified Resource.
4. Developers must have access to appropriate documentation which outlines the proper configuration and use of Heimdall.

With the functional requirements outlined, the following section will delve into the technical details necessary for the successful implementation of these end-user features.

4.1.2 Non-Functional Requirements

Non-Functional Requirements are technical specifications that help ensure that the system meets the desired functional requirements. They define the performance, reliability, security, and other characteristics that the system must possess in order to function effectively. Heimdall's Non-Functional Requirements are as follows:

1. Heimdall must be able to use the Unstructured Package to watch Resources of any type, including Core and Custom Resources.

2. When a Resource is watched, events on it should trigger a Reconcile for that Resource.
3. During a Reconcile, Heimdall must verify if events are coming from the owner Operator or not.
4. If the event is coming from a non-owner, Heimdall must configure a new Role and Role Binding to grant the correct permissions to that Operator's Service Account.
5. The Controller must also handle the reconciliation of the Slack integration, including:
    a. A Config Map storing the Slack channel information that Heimdall sends notifications to.
    b. A Secret storing the Slack API Key that Heimdall uses to send Slack messages.
6. Heimdall must use the Slack Go Client to send messages to a Slack Channel with information from the Config Map and Secret.
7. These messages must be interactive and provide a link to the affected Resource.
8. Finally, Heimdall must have a working and publicly available Docker image built for installation.

These requirements provide the foundation for the design of the system, as they outline the necessary capabilities and technical considerations that must be taken into account. In the following section, we will delve deeper into the architecture of Heimdall and discuss how these requirements will be implemented and integrated into the overall design of the system.

4.2 Architecture Model

The design of Heimdall is illustrated in Figure 4, which shows how the controller will function when the minimum viable product and stretch goals are implemented. This figure includes the responsibilities of the Controller in the context of an example scenario and details the various moving parts involved in the implementation. In the following section, we will delve deeper into the specific components and processes of Heimdall, providing a more detailed understanding of how the controller functions and achieves its functional and non-functional requirements.

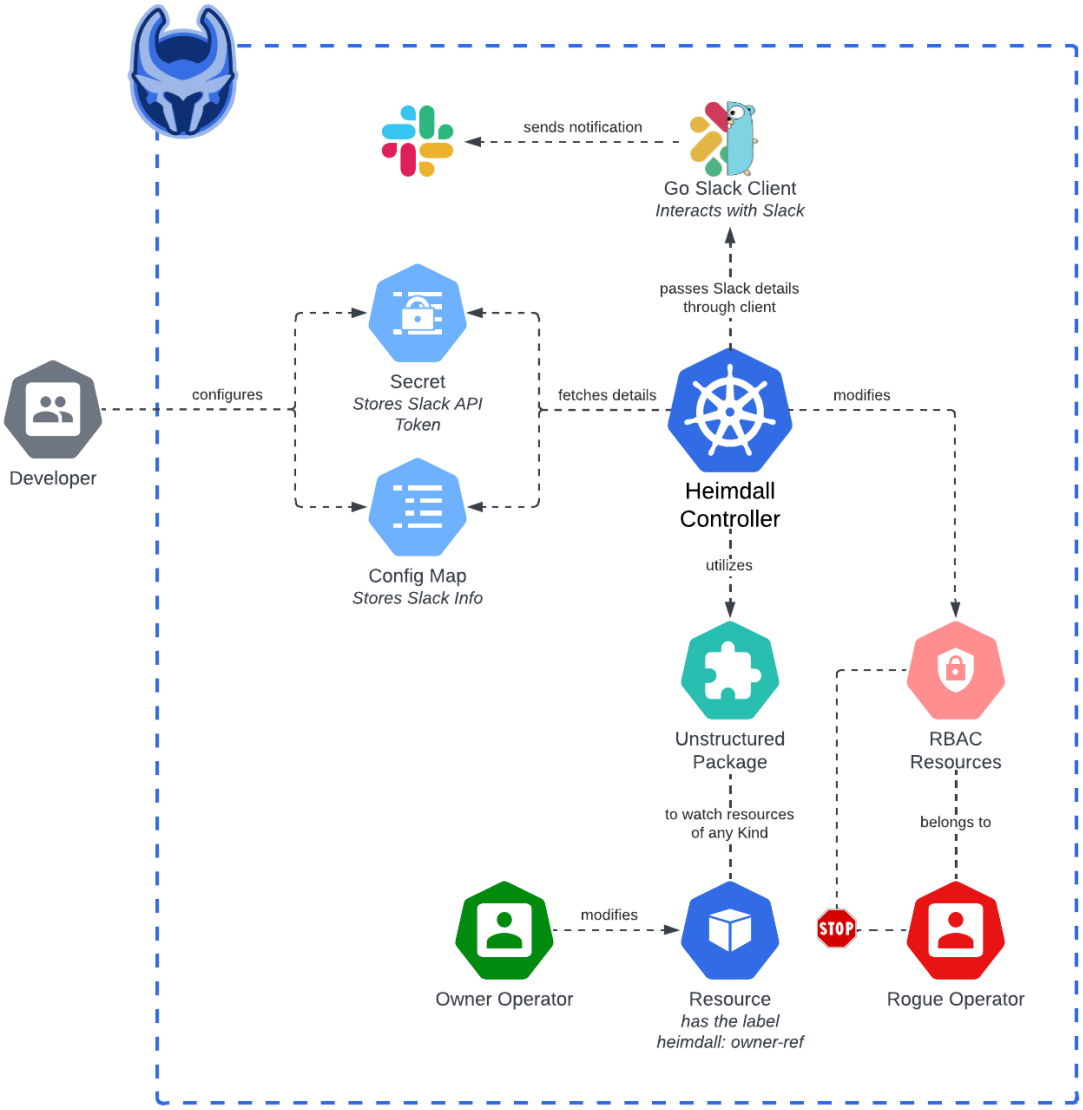


Figure 4: *Heimdall system architecture overview model*

4.3 System Design

The system design aims to provide a detailed description of the various components involved in the implementation of Heimdall. This section will cover the Controller Overview, the use of the Unstructured Package, Slack Integration, Role Based Access Control, and Installation.

4.3.1 Controller Overview

The Controller is the centre-piece of Heimdall. It is what makes this entire project work and will house the majority of the code. As most Kubernetes Controllers do, Heimdall's Controller will have two main functions at the very least. The first of which is the *Add* function. This will initialize the Controller itself using the *controller-runtime* package, which is discussed in Section 7.2, and will utilize the Unstructured Package to set a Watch on specific Resources. Once a Watch is set on a Resource,

this triggers the second main function *Reconcile*. This function will house a lot of the logic carried out by Heimdall. When an event occurs on a Watched Resources, the Reconcile function will carry out the logic to identify whether the change came from the owner or a rogue Operator. If the event came from a rogue Operator, it will then trigger two main pieces of code: the Slack Notification and the reconfiguration of Role Based Access Control Resources to block any future changes to that Resource from that Operator.

4.3.2 Unstructured Package Implementation

As part of the Heimdall Controller, the Unstructured package must be used in order to Watch Resources of any type in a Kubernetes cluster. This package provides a means by which to convert any type of Object in a Kubernetes cluster and turn it into a generic type. This will allow Heimdall to not have to declare the resource type when setting a Watch on Resources, so it can essentially Watch any resource in a cluster. This is especially helpful for Custom Resources as Heimdall needs to be installed and configured to work with any Kubernetes environment, so it needs to be dynamic in the way that it deals with Resources of different types.

4.3.3 Slack Integration

The integration with Slack involves a few different steps. The main components are the Config Map, the Secret, and the Slack Go client. Heimdall will first need to Reconcile the Config Map and the Secret. If either of them does not already exist in the cluster, the Controller will create them. This is so both Resources are guaranteed to be present on the cluster and ready for the Developer to configure them. Once they are configured correctly with the Slack API Token and the Slack Channel name, Heimdall can use that information to send Slack messages. The messages sent will need to be somewhat interactive. Ideally, this would include a link to the affected Resource, but at the very least it should contain the Name and Namespace of the Resource, the event type, and the source of the unwanted changes.

4.3.4 Role Based Access Control

Upon a watched Resource being changed by a non-owner, Heimdall will detect that change and trigger the Reconcile function for that Resource. To stop any new changes from being made by that Operator, the Controller must create new RBAC Resources like Roles and Role Bindings. The Roles will be bound to the Operator's Service Account with new Role Bindings so that when that Operator attempts to make another change to the watched Resource, it won't have the correct permissions. This functionality is subject to change as it has not yet been attempted and is not confirmed to be possible in a production environment. Upon actually implementing this functionality, it may differ slightly from this design.

4.3.5 Installation

Lastly, Heimdall needs to be installable via a public Docker image. An image can be built and pushed to a public container image repository like Docker Hub or Quay. This image can then be used by any developer that needs to implement Heimdall into their environment and run on the cluster. Appropriate documentation will need to exist so that the installation and configuration are easy to complete. This documentation should also include steps to create a Slack API for the developer's Slack Channel, and how to configure the Config Map and Secret correctly.

4.4 User Stories

User stories are a way of expressing requirements for a product in a way that is easily understood by a product's stakeholders. They are used in Agile development, as discussed in Section 5.1, and are an important tool for creating a shared understanding of the desired functionality of a product [Agile Alliance, 2015]. User stories help to focus on the value that a feature will provide to the end user and provide a framework for refining its requirements. The following are Heimdall's user stories:

- As a Developer, I want to be aware of changes to resources that I own so that I can take action if an unwanted change is made.
- As a Developer, I want to control the cadence of alerts so that I can control the noise created by those alerts.
- As a Developer, I want to claim ownership of the resources that I control.
- As a Developer, I want to control the changes to resources that I own.

These provide clear and concise descriptions of the desired functionality of Heimdall and will serve as a guide for the development team next semester.

4.5 Risk Analysis

There are a number of risks associated with this project's implementation, one of which is far more significant than the others. During the initial research into the viability of this project, there were no examples or previous attempts found that try to implement such a solution. This poses a significant risk as it means that it is not guaranteed that the implementation will be technically possible. An appropriate risk mitigation strategy is needed in order to ensure that the creation of Heimdall can still be completed to some degree. To do this, the project has been split up into a Minimum Viable Product (MVP) and stretch goals which are discussed further in Sections **??** and **??**. If after exhausting all attempts to implement the stretch goals, it is found that it is not possible, the MVP still provides a valid solution to the problem.

If the implementation is confirmed to be technically possible, the next risk will be that the project cannot be completed within the given time constraint. Again, this can be partially mitigated by the use of the MVP and stretch goals. Although, with the use of the chosen methodologies discussed in Section 5, the use of Sprints and following a Continuous Delivery approach to development should mitigate any concerns about time.

5 Methodology

This section will outline the Agile methodology that has been adopted for the development of Heimdall, as well as the version control, continuous integration, and continuous delivery practices that will be implemented to ensure the smooth development of the project. Additionally, we will delve into the testing approach taken and the decision to open-source the project.

5.1 Agile and Scrum

Agile is a project management and development method that helps teams deliver value to customers more efficiently. It is based on the idea of continuously iterating on and improving a product through

the collaboration of various teams [What is Agile?, 2022]. Agile has been chosen because it allows for a more iterative and flexible approach to development. By focusing on delivering the most important features first, there is a higher probability of being able to successfully implement the minimum viable product as well as the stretch goals.

Scrum on the other hand is a framework for Agile development that emphasizes collaboration, flexibility, and the ability to respond to change. One way that Agile teams track progress is through the use of Scrum artifacts such as burndown charts. While the Scrum roles of Scrum Master, Product Owner, and Development Team may not be applicable to a one-developer project like Heimdall, the use of these artifacts will still be useful in monitoring progress and ensuring that the project stays on track.

5.2 Scrum Artifacts

Teams who practice Agile and Scrum methodologies often collect Scrum Artifacts. These are pieces of information that a product's stakeholders and the team developing it use to describe its development. The main Scrum Artifacts used for this project include Product Backlog Refinement, Sprint Planning, and Sprint Reviews. There are also various Extended Artifacts that are not included in the official Scrum Artifacts definition. These include reporting mechanisms like Burn down Charts.

5.3 Version Control

What is VCS, Git, GitHub

5.4 Continuous Integration

Version control lies at the heart of Continuous Integration. CI is an Agile practice of integrating code changes to a product automatically from various contributors (product teams and open-source community contributions). It is a method used to consistently merge code changes into one central repository which runs automated tests and builds to ensure code functionality and integrity.

5.5 Continuous Delivery

Continuous Delivery is an approach

5.6 Testing Approach

5.7 Open Source

6 Technologies

6.1 Kubernetes

Kubernetes is an open-source tool used to implement modern micro-service-based architectures. It can be used to create, manage, and deploy containerized applications and is commonly known as a container orchestrator [Kubernetes Overview, 2022]. Containers are small processing units which house applications and their dependencies. They are bundled up into a singular image that is able to run on any hardware. This removes the need for installing required packages when attempting to run

an application. Kubernetes orchestrates the deployment of many containers to form larger applications that are highly available, fault-tolerant, and have high degrees of redundancy.

6.1.1 Resources

Resources, also known as objects, are how the state of a Kubernetes cluster is represented. These resources are detailed in *.yaml* format [K8s Objects, 2022]. They usually describe the following key pieces of information:

- Which container image is being used
- The compute resource available to the application, namely CPU and Memory limits
- Resource policies on how things like fault tolerance, upgrade behaviour, and restart behaviour should operate
- General resource information like the Namespace, Labels, and Annotations

The lowest-level Kubernetes resource is a Pod. Pods house the containers that run applications. They are expendable and can never be edited. If the desired state of a Pod is changed, it is destroyed and recreated to match the Pod's actual state with the new desired state. This also increases the cluster's degree of fault tolerance as if one Pod fails, instead of trying to recover it the Pod can be deleted and redeployed.

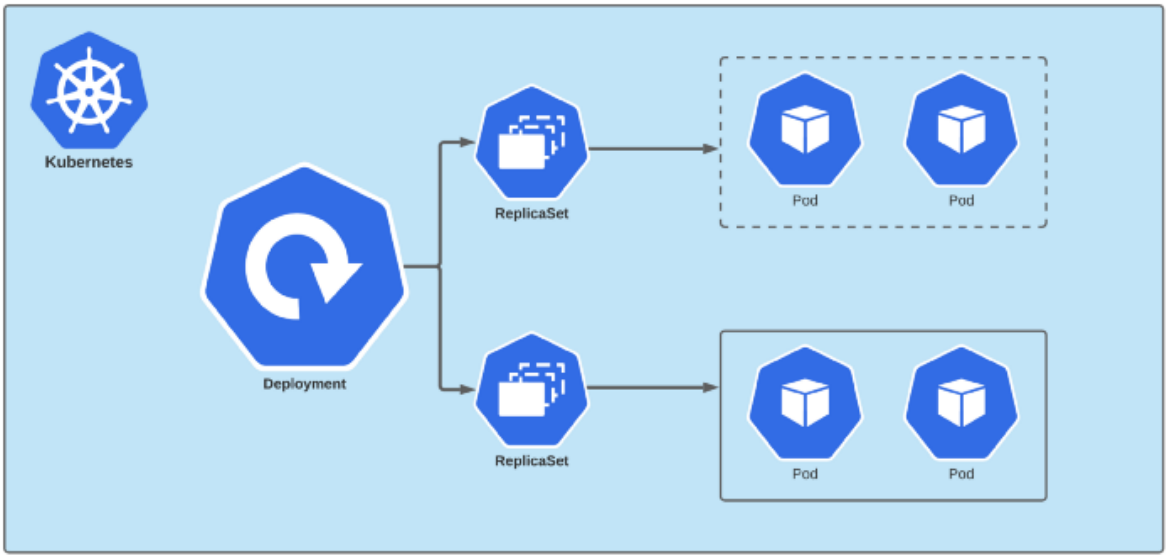
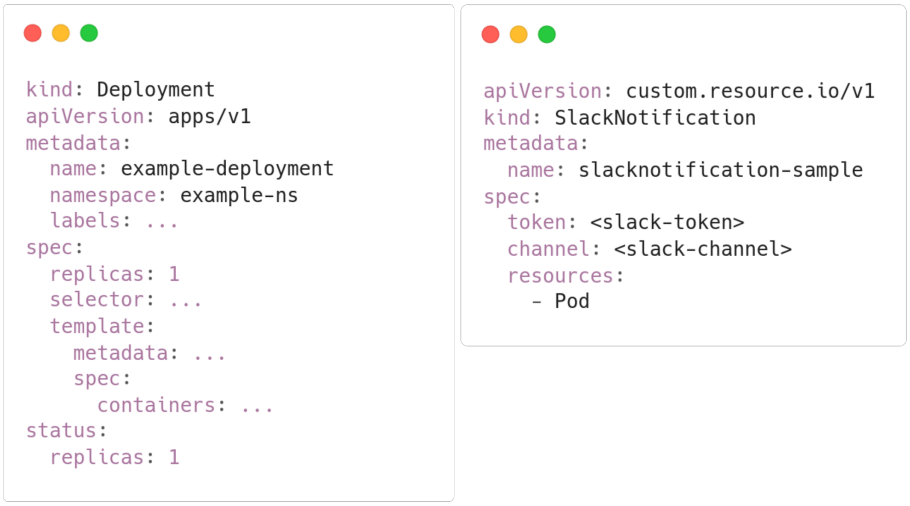


Figure 5: *Kubernetes application Resources: Deployments, Replica Sets, and Pods*

There are two other Resources which make up the structure of a running application in Kubernetes. Figure 5 [Y. Maharjan, 2020] details how Deployments and Replica Sets work with Pods in order to complete an application's workflow. Deployments are the top-level Resources in this structure and typically act as the source of truth (desired state) for Pods. If a Deployment's *.yaml* specification is changed by another Controller, then its Pod's desired state will change and be redeployed. The Deployment also acts as a desired state for Replica Sets, which are responsible mainly for managing the number of Pod replicas currently running.



```yaml
kind: Deployment
apiVersion: apps/v1
metadata:
  name: example-deployment
  namespace: example-ns
  labels: ...
spec:
  replicas: 1
  selector: ...
  template:
    metadata: ...
    spec:
      containers: ...
status:
  replicas: 1
```

```yaml
apiVersion: custom.resource.io/v1
kind: SlackNotification
metadata:
  name: slacknotification-sample
spec:
  token: <slack-token>
  channel: <slack-channel>
  resources:
    - Pod
```

Figure 6: *Example Core Resource (Deployment) comparison with Example Custom Resource*

Additionally, there are Custom Resources. They extend the Kubernetes API outside of the typically available resource types. They allow for customisation of the typical resource fields like *.spec* and can be reconciled (watched and maintained) by custom controllers [Custom Resources, 2022]. An example *yaml* definition of a Deployment (Core Resource) and a Custom Resource (Slack Custom Resource Example) can be seen in Figure 6. As seen, the Custom Resource (on the right) can have any custom fields in the Resource's *.spec* and the Kind and API Version values are different to that of Core Resources.

6.1.2 Controllers

As already explored in Section 3.3, Controllers are predominately responsible for watching the actual state of resources and matching it with their desired state. An example would be the Deployment Controller which ships natively on Kubernetes clusters. It ensures that the actual state of all Pods matches their desired state described in its Deployment. Custom Controllers can also perform various different operations including interacting with services outside of a cluster, like sending notifications to a Slack Channel [Controllers, 2022]. Custom Controllers and Custom Resources packaged up together to create one application are what Kubernetes refer to as the Operator pattern.

6.1.3 Operators

The core purpose of Operators is to attempt to perform the actions a human operator might. This includes monitoring, managing, and maintaining an application. This is done instead through code where an Operator is installed onto a cluster and will automatically handle any failures, react to changes in the cluster, and carry out actions upon specific events occurring [Operator Pattern, 2022]. This could be as simple as managing a web server and scaling up the number of web server pods as it gets more traffic to prevent Pod failure.

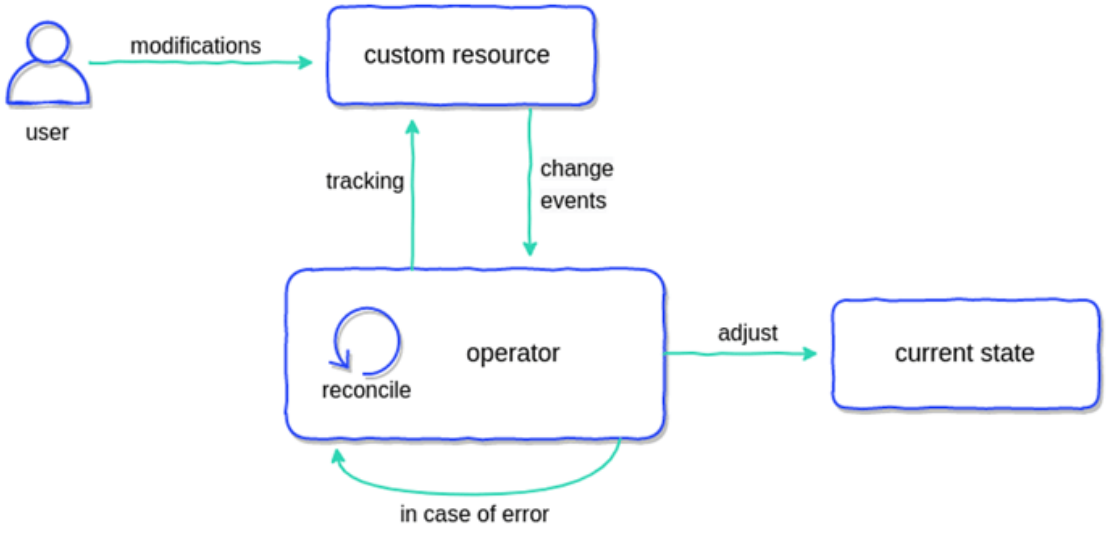


Figure 7: *Operator Pattern model of Custom Controller and Resources interaction*

Whatever the use case, they serve the purpose of automating tasks beyond what Kubernetes provides out of the box. A model of the Operator pattern taken from the Container Solutions blog can be seen in Figure 7 [P. Perzyna, 2020]. In order for Operators to carry out their intended functions, they need to be granted the necessary permissions.

6.1.4 Role-Based Access Control

Kubernetes environments use a Role-Based Access Control (RBAC) method to allow applications to perform certain tasks. Since the API Server acts as a mediator for all Kubernetes actions, the rules which describe a user's permissions use the verbs common to APIs like *GET*, *POST*, and *DELETE* [Using RBAC Authorization, 2022]. First, applications (Controllers, Pods, Operators, etc) are assigned a Service Account. Roles can then be created which describes what actions any application assigned with that Role can take. Finally, Bindings can be created to link these Roles to specific Service Accounts. This method of permission granting will likely prove useful for the implementation of the Stretch Goal discussed in Section **??**. These resources are discussed in context in Section 4.3.4.

6.2 Languages

There will be two main languages used in the creation of Heimdall, Go and YAML. They are the most commonly used languages when developing Kubernetes applications. This means that supporting documentation is plentiful, and experience with these languages is invaluable.

6.2.1 Go

The Go Programming Language is an open-source language created by Google. It is predominately used to develop cloud and network services, command-line interfaces, web applications, and Developer Operations style projects [Go Docs, 2022]. Kubernetes Controllers fall under the latter category and since I already have an elementary proficiency with Go, this is the chosen language for Heimdall. The

completion of this project will aid in increasing my knowledge and experience with Go, which will be beneficial for my future career.

6.2.2 YAML

YAML is a language highly similar to JSON. It is used frequently for defining configurations and is syntactically easy to use [Understanding Automation, 2021]. Similarly to Go, I have encountered and written YAML at Red Hat, and will be looking to further develop my skills with it during the creation of Heimdall. YAML is the language which defines the desired and actual state of Resources in Kubernetes as discussed in Section 6.1.1.

6.3 Libraries

Various libraries will be used for the development of Heimdall. Many of these are packaged up in the Software Development Kit discussed in Section 7.2.

6.3.1 Slack Go Client

The Slack Client for Go (slack-go) allows interaction with Slack channels via code. This client will allow Heimdall to make a connection to a Slack Channel, and send custom notifications containing key information about the problem occurring. This client will be used in the Slack Prototype discussed in Section **??**.

6.3.2 Unstructured Package

Section 6.1.2 detailed how Controllers can watch specific resources for events. Controllers can be configured to watch Resources which contain a specific label like *heimdall: watching*. This is useful as it means a Controller can watch and perform actions on any Resource which can be dynamically configured. An issue arises when we consider the Resource kind the Controller is being told to watch. If a Controller is set to watch a specific Resource, it requires a Kind to be specified, which could be *Pod*, *Deployment*, etc. What if the Controller is configured to only watch Pods with a specific label but then that label is added to a custom resource which does not contain the kind *Pod*. This is where the Unstructured Package comes into play. This package allows for a Controller to watch and interact with Resources of any Kind. This will be integral to the implementation of Heimdall as the Controller will need to complete its task on any Resource in a cluster regardless of its Kind. This is discussed further in the final proof of concept in Section **??**.

7 Tools

The following section will provide an overview of the various tools that will be used throughout the development of Heimdall. These tools are essential for the development of the Controller, the report, and the prototypes. From online LaTeX editors to command line tools for interacting with Kubernetes clusters, these tools will be integral to the success of the project. The section will also provide a brief overview of the purpose and use of each tool, as well as any relevant background information.

7.1 Overleaf

Overleaf is an online LaTeX editor and is the tool used to create this report. It will also be the tool used to create the Semester Two report. LaTeX allows the creation of professionally formatted documents using plain text instead of formatted text as typically done with other document writing tools like Google Docs and Microsoft Word [Overleaf About Us, 2022].

7.2 Operator SDK

Operator SDK is a Software Development Kit created for building Kubernetes Operators. It provides the necessary tools for the creation of custom Controllers and Resources (custom APIs) and generates a plethora of scaffolding code to get started on an Operator project [Operator SDK Overview, 2022]. During this scaffolding process, as seen in Figure 8, it can be specified whether or not to generate a Controller and Custom API. In the case of Heimdall, a Custom Resource is not needed so the flag to generate one is left to false.

```
/code/github.com/heimdall-controller/heimdall git:(master) ○
...
❯ operator-sdk init --domain=heimdall.io --repo=github.com/heimdall-controller/heimdall
❯ go get sigs.k8s.io/controller-runtime@v0.13.0
❯ go mod tidy

~/code/github.com/heimdall-controller/heimdall git:(master) ○
❯ operator-sdk create api --group=meta --version=v1 --kind=Pod --controller=true --resource=false
...
Writing scaffold for you to edit...
controllers/pod_controller.go
❯ go mod tidy
```

Figure 8: *Operator SDK Controller scaffolding via Command Line*

Aside from code generation, Operator SDK also initialises the use of essential libraries like *controller-runtime* and *controller-tools*. These are sets of libraries that aid in the creation of custom Controllers and will be used extensively during the development of Heimdall.

7.3 Minikube

Minikube is a small open-source Kubernetes environment that uses one node (a virtual machine that Kubernetes runs on). It can be run locally and is typically used for the development of Kubernetes applications [Minikube Docs, 2022]. Minikube provides a dashboard which allows the use of a graphical user interface to configure the Kubernetes cluster and interact with resources. This will be the environment in which Heimdall will be tested.

7.4 Docker

Docker is an open-source tool for developing, deploying and shipping containerized applications. Kubernetes uses Docker as its container runtime tool (to run containers) so it plays a large part in a Kubernetes environment [Docker Overview, 2022]. For Heimdall, it will be used to build and push

custom images of Heimdall so that they can be deployed to any Kubernetes environment. This process of pushing images will also be integral to the release process of Heimdall as discussed in Section 5.4. An alternative to Docker for this purpose is Podman. Podman is another open-source tool created by Red Hat for the same purpose. I have had previous experience with both, but I chose Docker as it is more commonly used so I would like to improve my skills with that during Heimdall's development. Since the two tools are very similar with the commands mostly being identical, any development of skill with Docker will transfer to Podman as well.

7.5 Kubectl

Kubectl is a command line tool used to interact with Kubernetes clusters. Using Kubectl allows a developer to interact directly with the Kubernetes API Server to make changes to Resources in a cluster [Kubernetes Tools].

7.6 Git and GitHub

Git and GitHub are integral tools to implement a viable version control system as discussed in Section 5.3. Git is the command line tool used to track changes to a code base while GitHub is the cloud-based platform that hosts repositories (projects) for tracking those changes. Git and GitHub are being utilized to implement the version control system for the Heimdall Controller, the report, and the prototypes. All of these repositories are stored in a GitHub Organization as discussed in Section 5.7 in order to separate them from personal projects. Git and GitHub will be used to work on multiple different Heimdall features at once, track changes so they can be reverted if a bug is introduced, and host the Continuous Integration and Continuous Delivery systems via GitHub Actions as discussed in Sections 5.4, 5.5, and 7.7

7.7 GitHub Actions

GitHub Actions provides a Continuous Integration and Continuous Delivery system for the automation of building, testing, and deploying software products. GitHub Action Workflows can be triggered upon a particular event like a push to the *master* branch. Once triggered, it can perform actions like ensuring the project builds correctly, checking code formatting standards are met, and running tests. They can also be used to automate product releases [Learn GitHub Actions, 2022]. This tool is already being utilized to build the LaTeX document on every Pull Request event and Push event for the Heimdall Report repository as seen in Appendix E. This implementation provides a familiarity with GitHub Actions in preparation for Semester Two, while also ensuring the report is building successfully whenever changes are made.

7.8 Jira

Lastly, Jira is a tool which can be used by teams who adopt the Agile methodology discussed in Section 5.1. It belongs to the Atlassian software set and provides templates for Scrum and Kanban boards for project management. Jira is the tool of choice for tracking pieces of work for this project. Originally, Trello was being used for this purpose. Trello is lightweight and easy to use but does not provide many of the features that come with Jira. The Burndown charts discussed in Section 5.2 are not possible to generate out-of-the-box with Trello and are only available through paid extensions while they are

free with Jira. Other useful features like Sprint management, release creation, and automatic linking of GitHub Pull Requests to Jira Tasks made Jira the more favourable option.

8 Proof of Concept

8.1 Slack Prototype

8.2 Watcher Prototype

8.3 Undecided Prototype

9 Summary

9.1 Review

9.2 Semester Two Outline

References

[What is Agile?, 2022] *Atlassian*,
URL: https://www.atlassian.com/agile

[Operator Pattern, 2022] *Kubernetes*,
URL: https://kubernetes.io/docs/concepts/extend-kubernetes/operator/

[IBM API-Connect, 2022] *Understanding rate limits for APIs and Plans*,
URL: https://www.ibm.com/docs/en/api-connect/10.0.1.x?topic=connect-understanding-rate-limits-apis-plans

[I. Sommerville, 2021] *Engineering Software Products: An Introduction to Modern Software Engineering*, 2021.

[Kubernetes Overview, 2022] *Kubernetes*,
URL: https://kubernetes.io/docs/concepts/overview/

[Red Hat Developer, 2020] *Red Hat OpenShift API Management*, Red Hat, 2020.
URL: https://developers.redhat.com/products/red-hat-openshift-api-management/overview

[K8s Objects, 2022] *Understanding Kubernetes Objects*, Kubernetes.
URL: https://kubernetes.io/docs/concepts/overview/working-with-objects/kubernetes-objects/

[Y. Maharjan, 2020] *How Rolling and Rollback Deployments work in Kubernetes*, Medium, 2020.
URL: https://yankeexe.medium.com/how-rolling-and-rollback-deployments-work-in-kubernetes-8db4c4dce599

[Custom Resources, 2022] *API Extension*, Kubernetes.
URL: https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/

[Controllers, 2022] *Kubernetes Architecture*, Kubernetes.
URL: https://kubernetes.io/docs/concepts/architecture/controller/

[Using RBAC Authorization, 2022] *API Access Control*, Kubernetes.
URL: https://kubernetes.io/docs/reference/access-authn-authz/rbac/

[P. Perzyna, 2020] *Kubernetes Operators Explained*,
URL: https://blog.container-solutions.com/kubernetes-operators-explained

[Go Docs, 2022] *The Go Programming Language*,
URL: https://go.dev/

[Understanding Automation, 2021] *What is YAML?*, Red Hat.
URL: https://www.redhat.com/en/topics/automation/what-is-yaml

[Overleaf About Us, 2022] *About us*, Overleaf, Online LaTeX Editor.
URL: https://www.overleaf.com/about

[Operator SDK Overview, 2022] *Operator SDK Documentation*,
URL: https://sdk.operatorframework.io/docs/overview/

[Minikube Docs, 2022] *Minikube*,
URL: https://minikube.sigs.k8s.io/docs/

[Docker Overview, 2022] *Docker Documentation*, 2022.
URL: https://docs.docker.com/get-started/overview/

[Kubernetes Tools] *Install Tools*, Kubernetes.
URL: https://kubernetes.io/docs/tasks/tools/kubectl

[Learn GitHub Actions, 2022] *Understanding GitHub Actions*, GitHub Docs.
URL: https://docs.github.com/en/actions/learn-github-actions/understanding-github-actions

[P. Gorbachenko, 2021] *Functional vs Non-Functional Requirements*, Enkonix.
URL: https://enkonix.com/blog/functional-requirements-vs-non-functional/

[Agile Alliance, 2015] *Agile Alliance*, Dec. 17, 2015.
URL: https://www.agilealliance.org/glossary/user-stories/

Appendices

A Heimdall GitHub Organization

https://github.com/heimdall-controller

B Slack Prototype GitHub Repository

https://github.com/heimdall-controller/slack-prototype

C Watcher Prototype GitHub Repository

https://github.com/heimdall-controller/watcher-prototype

D Unstructured Prototype GitHub Repository

https://github.com/heimdall-controller/unstructured-prototype

E Heimdall Report GitHub Repository

https://github.com/heimdall-controller/heimdall-report