Bachelor of Science (Hons) Applied Computing

Heimdall - A Kubernetes Extension

Author:
Bryan Keane

Supervisor:
Lucy White

Contents

List of Figures

List of Tables

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 5.

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].

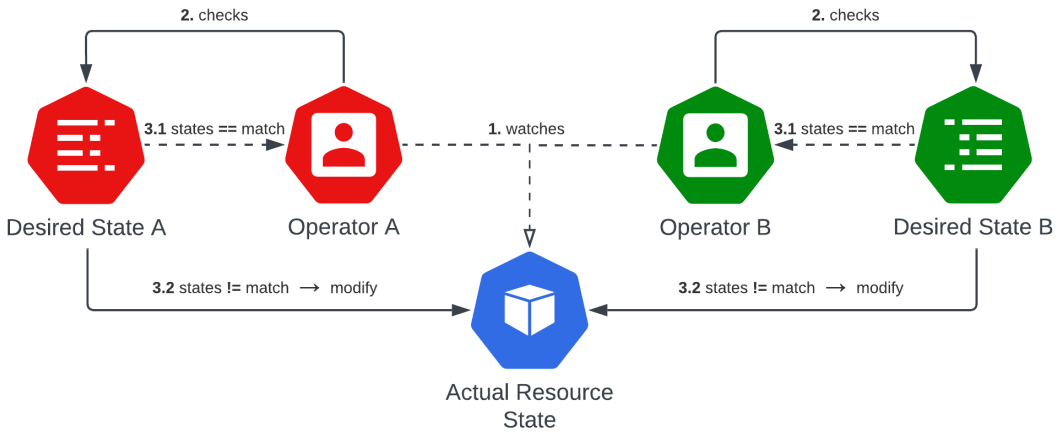

Figure 1: *Model of The Problem - Two Operators Fighting Over a Resource's State*

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.

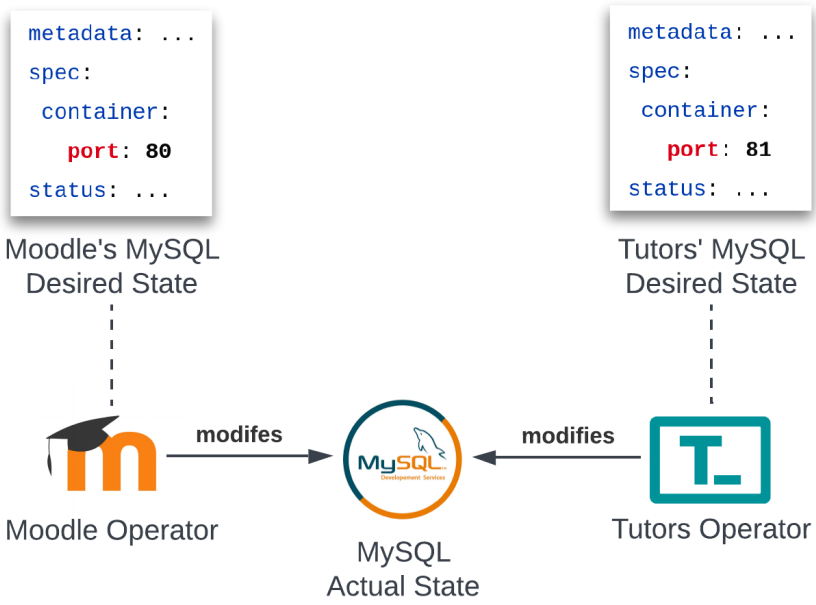

Figure 2: *Simplified Model of Moodle, Tutors, and MySQL Industry Example*

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.

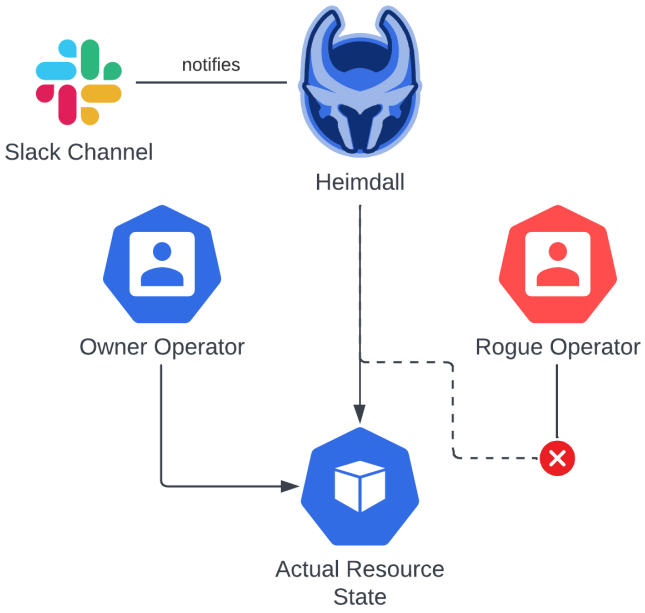

Figure 3: *Model of the solution: Heimdall*

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 involve allowing for an atomic owner of a resource to be set and blocking changes 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 7.3.

3.5 Risk Analysis

Various risks are associated with the implementation of Heimdall, 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 7.2 and 7.3. 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 4, the use of Sprints and following a Continuous Delivery approach to development should mitigate any concerns about time.

4 Methodology

The chosen methodology for software development teams is paramount for the successful planning and development of a product. Historically, the Waterfall Methodology was commonplace - but now Agile is the gold standard. Waterfall involved a distinct sequence of actions for engineers to follow. In short, it involved extensive design and planning before ever writing a line of code. Long documents detailing product design and strategy were written to fit the stakeholder's requirements. This proved to be ineffective as in most cases, holes in the design are discovered after beginning the implementation. In recent years, Agile has begun replacing this framework as it has proven much more efficient for software development teams [M. McCormick, 2012].

4.1 Agile

"Agile is an iterative approach to project management and software development that helps teams deliver value to their customers faster and with fewer headaches" [What is Agile?, 2022].

4.2 Scrum Roles

Scrum has three main roles: Scrum Master, the Product Owner, and the Development Team. These are used to help describe the responsibilities of each stakeholder for a product.

4.3 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.

4.4 Version Control

What is VCS, Git, GitHub

4.5 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.

4.6 Continuous Delivery

Continuous Delivery is an approach

4.7 Testing Approach

4.8 Open Source

5 Technologies

5.1 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.

5.1.1 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.

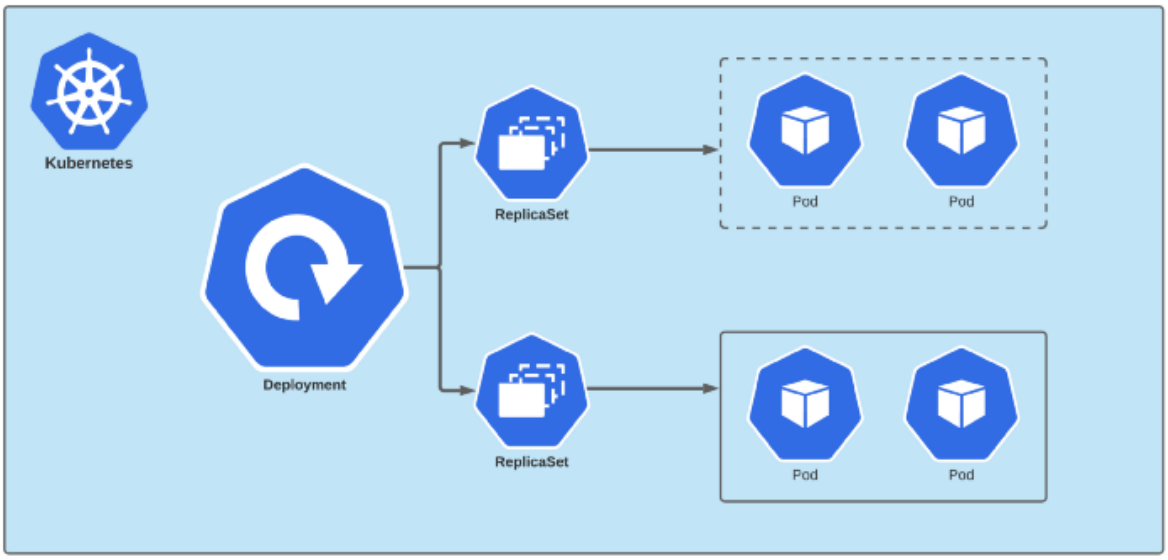

Figure 4: *Kubernetes application Resources: Deployments, Replica Sets, and Pods*

There are two other Resources which make up the structure of a running application in Kubernetes. Figure 4 [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 5: *Example Core Resource (Deployment) comparison with Example Custom Resource*

Aside from core resources, there are also 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 5 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.

5.1.2 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.

5.1.3 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.

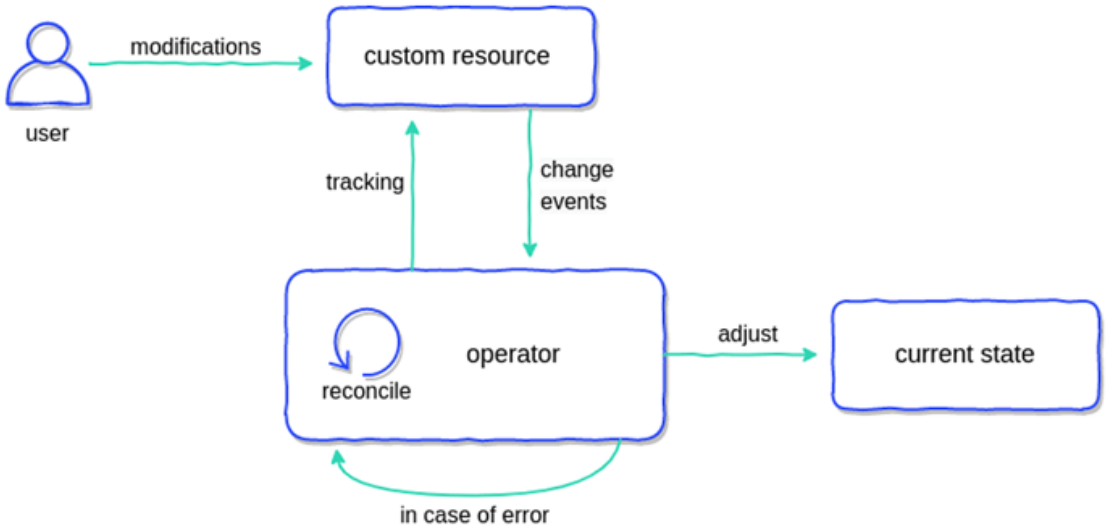

Figure 6: *Operator Pattern model of Custom Controller and Resources interaction*

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 6 [P. Perzyna, 2020]. In order for Operators to carry out the tasks they need to perform, they need to be granted permission to complete them.

5.1.4 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 7.3.

5.1.5 Unstructured Package

Section 5.1.2 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 8.3.

5.2 Go

5.2.1 Slack Client

5.3 YAML

6 Tools

6.1 Operator SDK

6.2 Minikube

6.3 Docker

6.4 GitHub Actions

6.5 Version Control

6.6 Jira

6.7 Overleaf

7 Design

7.1 System Architecture Overview

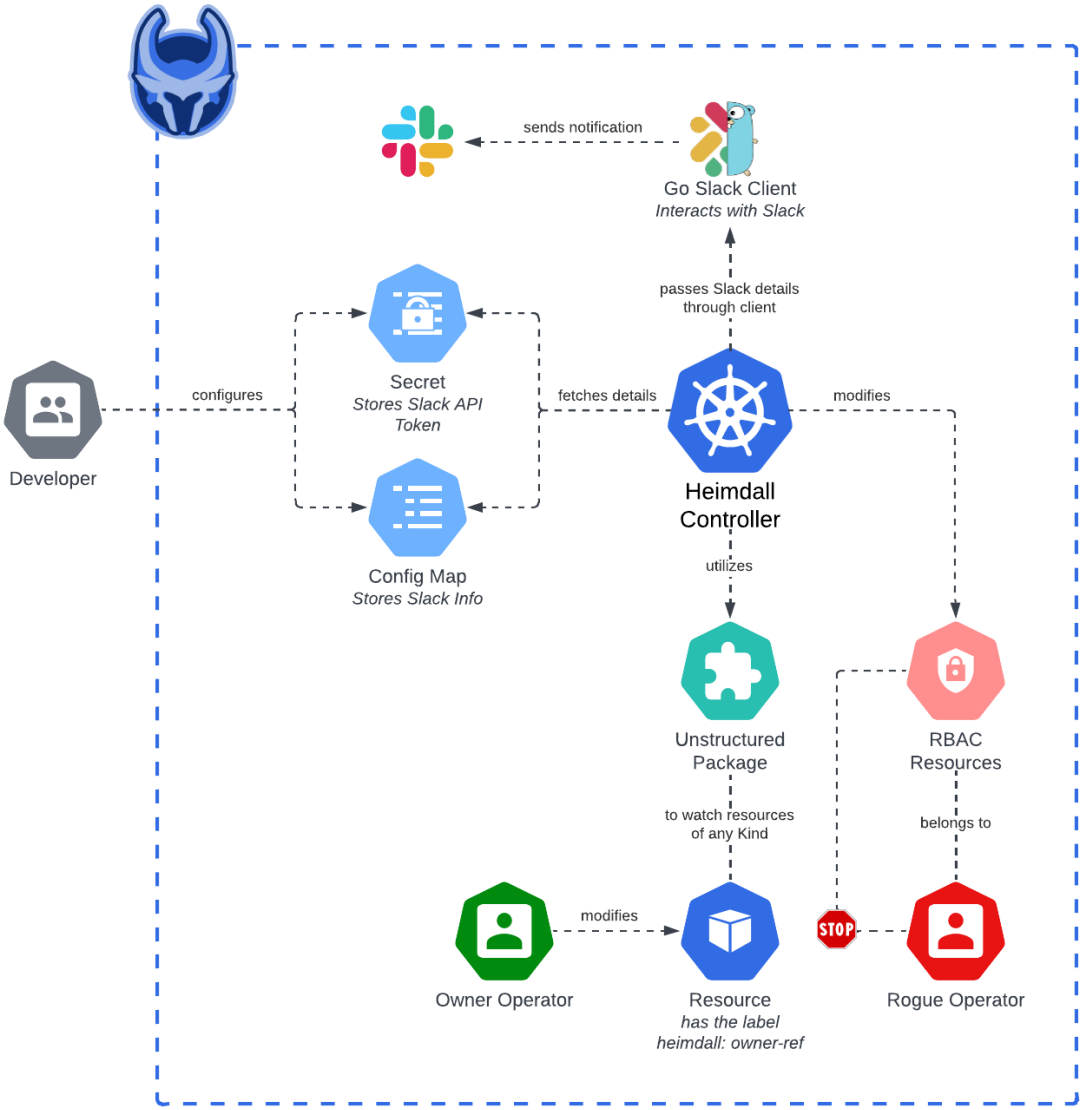

Figure 7: *Heimdall system architecture overview model*

Create a diagram

7.2 Minimum Viable Product

7.3 Stretch Goals

7.4 Requirements

7.4.1 Functional Requirements

What the project actually does From a devs POV

7.4.2 Non Functional Requirements

Under the hood what I need to get working. Security

7.5 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.

7.6 Personal Stories

As a student, I want to have achieved First Class Honors, so that I can reach my academic goals.

As an aspiring Software Engineer, I want to contribute a solution to a common problem faced by Software Engineers working with Kubernetes, so that I can make a valuable contribution to the open-source community.

7.7 User Definitions

7.8 Models

8 Proof of Concept

There are three prototypes necessary for the planning of Heimdall.

8.1 Slack Prototype

8.2 Watcher Prototype

8.3 Unstructured Prototype

9 Reflection

10 Summary

10.1 Review

10.2 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

[M. McCormick, 2012] *Waterfall vs Agile Methodology*, M. McCormick,
2nd ed. MPCS, Inc., 2012.

[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

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