Defining an Evaluation Model for Container Orchestration Operator Frameworks

André Arnesson
an8336ar-s@student.lu.se
Samuel Alberius
sa1068al-s@student.lu.se

Sinch AB

Supervisors: William Tärneberg & Josef Holmberg

Examiner: Maria Kihl

June 22, 2023

© 2023
Printed in Sweden
Tryckeriet i E-huset, Lund

Abstract

The growing complexity of cloud native applications has necessitated the introduction of operators to the container orchestration tools' suite of components. Operators affords developers the ability to encode domain knowledge and make fine-grained controllers for their Kubernetes clusters, radically extending the range of feasible applications to host. Operators are however vast software entities and places large requirements on its developing party, almost forcing the use of a software framework. This master's thesis explores the evaluation and comparison of Kubernetes operator frameworks. To address the challenge, a novel solution design is proposed, presenting an evaluation model that categorizes and assesses the frameworks using predefined attributes and accompanying metrics. The attributes were related to one of two overarching categories deemed appropriate for by the performed framework study, *open source health status* or *operator capability*. Profile assignment is done through the concordance non-discordance principle, inspired by multi-criteria decision theory. This research contributes to the understanding of evaluating Kubernetes operator frameworks and offers valuable insights for developers and decision-makers in selecting appropriate tools for managing cloud-based applications.

Key words: container orchestration, cloud native application, operator, software framework, evaluation model

Acknowledgements

We would like to thank and acknowledge our supervisor and examiner from LTH, William Tärneberg and Maria Kihl. We would also like to thank our company partner Sinch AB, especially Josef Holmberg.

Popular Science Summary

Applying evaluation models in all the right places

In today's interconnected world, online services have become an integral part of our lives. This has seen the demand for scalable, reliable, and efficient infrastructure skyrocket. In the wake of this demand cloud services and Kubernetes have taken great strides to become industry standards. However, large parts of Kubernetes's facilitating technologies are still unexplored.

Cloud services have enjoyed great success on the premise of cloud native applications. Such applications are what facilitates sought-after qualities such as scalability and reliability. However, cloud services and their applications are prone to difficulties in hosting due to their micro-service-based architecture approach. This has spurred the uprising of orchestration tools, such as Kubernetes, which serve as a central hub for management of micro-services.

Kubernetes was introduced by Google in 2014 and offers a set of building blocks that collectively provides mechanisms to deploy and maintain cloud native applications. These building blocks are heavily standardized, and for good reason. Kubernetes delves into very complex areas and developers would rather focus on their application's logic than the intricacies of microservices and orchestration. Such standardization may however prove problematic, something that is made obvious in the cases of more elaborate and multi-faceted applications. Cloud native applications are often divided on the condition of either being stateful or stateless, meaning it either keeps persistent data or it doesn't. Applications of the stateful kind requires knowledge and techniques that the native Kubernetes building blocks have difficulties providing. To rectify this gap of capability Kubernetes recently introduced the operator pattern into its suits of components.

Once implemented, operators can extend Kubernetes with encoded domain knowledge, enabling developers to provide Kubernetes with whatever capability they find missing. As with most things orchestration related, operators are complex pieces of software, and in practice developers are forced to build them from existing frameworks. This however proves a difficult task, frameworks are plentiful and provide vastly different developing conditions for its users. As Kubernetes itself, frameworks are developed through open source de-

velopment adding a new dimension of considerations, atop of the more obvious of capability and policy compliance.

To address the challenges associated with comparing and selecting operator frameworks, we present a model for their evaluation. Through literature review and a study of the Kubernetes environment, several attributes representing the qualities of an operator framework are in this study defined. When applied, this model should allow developers to draw conclusions based on familiar and relevant concepts instead of figuratively fumbling in the darkness of online articles and user-provided reviews and sales pitches. Because after all, who doesn't want a smooth operator-framework-choosing-experience?

Contributions

Author1 was the main contributor to **Chapter 2** and **Chapter 6**. *Author2* was the main contributor to **Chapter 1**, **Chapter 3**, **Chapter 4**, **Chapter 5**, and **Chapter 8**. **Chapter 7** was deemed equally split amongst the two.

Table of Contents

Chapter 1

Introduction

This chapter aims to introduce to the reader the field of cloud services and its facilitating technologies, mainly container orchestration. It will motivate cloud computing's existence and prominence in the current technology landscape as well as present existing difficulties in specific cloud deployments, those being the reason for this thesis. It will also present this thesis' aims for scientific contribution.

1.1 Background to Thesis

Cloud computing and cloud services are evergrowing in popularity due to their ability to provide organizations with flexible and scalable access to computing resources, without the need for complex on-premises infrastructure. The technology behind such services is in turn largely facilitated by the creation and deployment of isolated environments achieved by containerization and virtualization [37]. Several demands are being placed on such environments, among which scalability, availability, portability, and resource management are most prominent. Such environments or architectures are generally complex to maintain and in light of this, container orchestration is often employed [29].

Orchestration relies on pre-built platforms in order to manage creation, deployment, and supervision over containerized systems. Containers themselves are lightweight and short-lived by nature meaning a well-defined infrastructure such as an orchestration tool is necessary for any non-arbitrary production environment [29]. Container orchestration is a now well-established practice performed by a multitude of tools such as Amazon Elastic, Google Cloud Run, Centurion, and Kubernetes [47], this thesis will focus on Kubernetes. Container orchestration tools such as Kubernetes handle stateless containerized applications with built-in functionality. The problem arises when applications and systems become more complex. An example of such complexity is applications becoming stateful, meaning it keeps an internal, persisting state. Such applications are often subject to a more complex life-cycle management scheme and can therefore not be maintained by default Kubernetes functionality [18].

To rectify this operators were built atop Kubernetes, effectively extending its API to incorporate functionality based on ad hoc domain knowledge. Operators allow for Kubernetes-native, self-contained application management systems capable of handling complex logic for stateful applications. The operator continuously runs a reconciliation loop to compare the observed state of the resource against a predefined desired state, thus ensuring a healthy application. Operators may be built in different ways using various programming languages, software development kits (SDKs) and frameworks. This versatility of development presents a challenging barrier to enter for any stakeholder intending to implement an operator. The choice of framework is a largely unexplored science, leaving developers with the intimidating task of sifting through a vast array of options with varying capabilities, ease of implementation, and community support.

1.2 Motivation and Goal Formulation

Together with Sinch, we have identified a challenge presented to developers and those in technology leadership positions when deploying a Kubernetes Operator, specifically for complex, stateful applications. Effective decision-making requires extensive knowledge of both containerized applications in the Kubernetes environment and various Kubernetes operator frameworks. Our thesis aims to clarify the options available for operator frameworks, and later proposing a method for their evaluation and subsequent decision-making.

1.2.1 Research Questions

This thesis aims to answer the following questions

RQ 1 *What Kubernetes operator frameworks are most widely used today for managing the lifecycle of containerized applications?*

RQ 2 *What attributes and metrics should be used to aid in operator framework-related decision making?*

RQ 3 *How can the operator frameworks (see Research Question 1) be classified and categorized?*

1.3 Report Structure

This thesis spans eight chapters. **Chapter 1** introduces the subject at hand and provides motivation for and the scope of this study. It also brings up previous works related to this thesis's process. **Chapter 2** introduces all necessary theory. **Chapter 3** presents our framework study. **Chapter 4** and **Chapter 5** provides an overlook of the proposed solution design as well as our implementation of it. **Chapter 6** displays all found results. **Chapter 7** discusses the gathered results. **Chapter 8** concludes this master thesis and discusses potential future work.

1.4 Related Work

Related work will introduce previous research papers that lay as foundations for conclusions drawn in this thesis.

1.4.1 Evaluating the Quality of Open Source Software

A paper published by Sinellis et al. [41] discusses and attempts to remedy the current difficulties of assessing the quality of open-source software projects. Traditionally, software product quality attributes, which describe the effectiveness of software projects, have been kept proprietary and so too was the code base it assessed. However, due to the emergence and growing adoption of open-source software, there is now an opportunity for transparent evaluations of both the software products and the underlying processes that produce them. The authors present metrics relating to both product and process and share how they may provide insights into software quality. The study evaluates and finds appropriate attributes based on found metric values and the correlation of those to a set of predefined project quality profiles. A hierarchical quality model is presented in which categories of evaluation are defined, as well as attributes and their underlying metrics. The study finds two overarching attributes that comprehensively describe the quality of open source software being *Product (Code) Quality* and *Community Quality.* Under the former lies the attributes *maintainability*, *reliability*, and *security.* Under the latter *documentation quality* and *developer base quality.*

1.4.2 Software Product and Process Assessment

Morisio et al. [31] define a methodology, applied in this thesis, in which software entities, products, or processes, may be assessed and evaluated for the purpose of decision-making. The authors explain how a software entity is characterized by its defining attributes, each attribute measured through one or several metrics. Evaluation and future decision-making however greatly benefit from a compiled view of the entity, requiring attribute and metric aggregation. A commonly employed approach for aggregation is the weighted average sum (WAS) approach. This however lacks any relevance when its intended subjects are unable to be placed on interval scales, and instead only ordinal scales (such as bad, acceptable, and good). Ordinal scales imply a ranking relation among its attributes however make no statement of the magnitude of difference. WAS assumes equal intervals between scale values and when applied to ordinal scales introduces arbitrary information, invalidating future analysis. Due to the frequency with which ordinal scales occur in real-world situations, Morisio et al. introduce a methodology for a better-suited aggregation affording a holistic and complete evaluation of a software entity.

The method described in the paper is divided into two phases, with the first phase being the definition of an appropriate evaluation model and its attributes and profiles, the second phase is then metric collection and aggregation. The model definition is highly case-specific and carries no universally applicable method. It is instead a process of defining the purpose of the evaluation, what resources for evaluation are available, and what uncertainties currently trouble decision-making. This is largely based on domain knowledge and previous experiences. With due research attributes for evaluation may be defined and introduced to the model. All chosen attributes must be paired with a scale expressing preference. As attributes themselves are neutral a preference relation is placed atop them stating an intentional ranking order. Such could be the relation r set on the attribute length of code l:
More lines of code are desirable (identity scale)

$$l(x) > l(y) \leftrightarrow r(x, y) \text{ (x is better than y if l(x) is greater than l(y))}$$

or *Less lines of code are desirable* (inverse scale)

$$l(x) < l(y) \leftrightarrow r(x, y) \text{ (x is better than y if l(x) is lesser than l(y))}$$

Both are applicable relations but with obvious semantic differences. The correct or intended relation is expressed by the model attribute preference relation and should be motivated by domain knowledge. To then make possible potential categorization and feasible comparison of the software entities, profiles are defined for the model. Profiles allow for a justifiable aggregation of different metrics, importantly both placed on interval and ordinal scales. Metrics are to be assigned profiles using measurements when placed on interval scales, otherwise pending predefined preferences.

The second phase constitutes data collection and subsequent evaluation in accordance with the created model. All relevant software entities are assessed by gathering metrics that represent the various attributes, and a profile is assigned. This stage presents a first possibility of comparison as all attributes now have ordinal profiles attributed to them. Further aggregation may however be employed to achieve a single grade for comparison. Morisio et al. introduce the outranking relation, a commonly practiced tool when evaluating based on multiple criteria.

Chapter 2

Theory

The following chapter will provide the information necessary to comprehend the rationale of this thesis. It will begin by exploring the fundamental principles of cloud services and container orchestration tools, with a specific emphasis on Kubernetes. The focus will later shift to investigating the problems with deploying stateful applications in a Kubernetes environment. Finally, the section will present relevant qualities of open source development.

2.1 Cloud services

The technology architectures of modern enterprises are largely characterized by the need for on-demand scalability and flexibility in its employed IT resources. As the adoption of technology solutions is ever-increasing, so is the need for a greater financial and personnel investment to ensure IT resources are consistently available. Offered as solutions to complex on-premises infrastructure, cloud computing, and cloud services represent a paradigm shift of such architecture design, instead delivering IT resources via internet [43].

Cloud computing services encompass all such services delivered from remote instances, with notable categories being Software as a Service (SaaS), Platform as a Service (PaaS), and Infrastructure as a Service (IaaS). The topic of cloud services is further sub-categorized based on the infrastructure location and utilization. A private instance may be operated for a single organization's benefit only, while a public instance instead serves over the public internet being employed by several organizations through subscriptions. Additionally, hybrid solutions exist as a cross between the two previous deployment models [44].

There are several documented benefits to utilizing cloud services, such as ease of deployment, cost efficiency, rapid flexibility, and elastic scalability. The argument for ease of deployment stems from cloud services often being an opt-in subscription model in which utilities are dispensed as needed by the provider. Since the service is provided via external sources, there are only minimal requirements for on-premises hardware and implementation expertise. Similarly, it applies to cost efficiency as the cost-model of cloud services is preferable to organizations wanting to avoid the upfront cost of deploying hardware for hosting services. Rapid flexibility and elastic scalability share the domain of adjusting the current workforce, one proactively by configuration and one retroactively through load-balancing algorithmic decision-making. Both of which are possible and encouraged through cloud services due to its nature of dynamically allocating resources based on demand [21].

2.1.1 Cloud Native Applications

Applications offered via the cloud may in theory be identical to its on-premises counterparts. However, they often differs structurally in order to better utilize the qualities of cloud services. This approach to software development, in which focus lies on efficient cloud deployment, spawned the idea of the term cloud-native development and is largely facilitated by service-distribution-focused implementation [21]. This may take many forms but is often realized through micro-service-oriented application architecture. This style of architecture structures an application as a collection of independently deployable and manageable services that, when working together, create the impression of a unified workflow. Independently existing micro-services reinforce and highlight the benefits associated with cloud services with great synergy [36].

2.2 Container Technology

In recent years, there has been a shift in application deployment from being hosted on physical machines to virtual machines, and more recently to containers. While all the different approaches are still in use, containerization has become the standard in the industry [30]. The underlying reason for the development has been driven by the need for scaling and greater flexibility deploying and managing applications.

2.2.1 Isolated Environments

All technologies relying on individually deployed micro-services require a separate and isolated environment for each involved service, thus ensuring configurations and specific dependencies are satisfied. Establishing separation of deployed micro-services is in theory rudimentary, however when applying principles of scaling, cost-efficiency, and centralized management the issue quickly grows more complex. This motivated the development of micro-services being separated virtually, a technique in which a single machine may act host for multiple separated services. Over the course of numerous years, there have been several generations of preferred technologies for virtualized separation when deploying micro-services [39].

2.2.2 From Traditional to Containerized Deployments

Going from physical deployment to hosting multiple virtual services on the same machine allowed greater utilization of resources. Every virtual machine gets allocated some hardware resources, thus making it behave like a separate machine. The next step in the evolution of application deployment was containerization, which is a form of virtualization where multiple applications are isolated from each other on a single machine, similar to virtual machines [30]. The main difference being that containers enable multiple applications to be run on a single operating system.

Apart from the hosting medium, the approach of deploying applications has evolved from a monolithic architecture to microservice-based architecture [4]. Having a traditional monolithic architecture means that the entire application is built as one big unit, whereas a microservice-based approach is composed of independent services that communicate with each other using APIs. Having a monolithic application naturally comes with a tightly coupled structure, meaning that the whole application has to be managed as one big entity. In contrast, with a microservice-based approach, you can choose what services to scale within the application based on their resource requirement. However, deploying microservice-based applications can become more complex since the individual services need to be able to communicate with each other. This is one of the reasons why container orchestration tools, such as Kubernetes, have increasingly gained popularity in recent years [42].

2.2.3 Containers

Containers are a lightweight virtualized technology that packages the application with its code and dependencies. Unlike traditional virtual machines, they do not require an operating system per application, instead, they share the OS kernel, which increases portability. These features of containerized applications simplify testing and deployment by providing a consistent environment across platforms [10].

2.2.4 Container Orchestration Tools

Managing the entire life cycle of hundreds or even thousands of containerized applications is a difficult task if done manually, especially if the application is more complex. This is why container orchestration tools have become increasingly popular in recent years [26]. The orchestration tools assist with automation in the cluster, automatically scaling, load balancing, resource allocation, and monitoring. One of the main benefits of using container orchestration, and the one that will be of focus of this thesis, is automation. By automating the process of application management the organization can save time and reduce the risk of human error [30].

There are numerous different orchestration tools available, each with its own unique strengths and weaknesses. Malviya and Dwivedi [26] present a study where they compare 4 popular orchestrators that are used in the industry: Kubernetes, Docker Swarm, Mesos, and Redhat OpenShift. The different factors that were compared amongst the orchestrators were deployment, security, stability, scalability etc. The conclusion of the study was that some orchestrators offered higher security than Kubernetes and were easier to use while Kubernetes was the most optimal one regarding scheduling features.

2.3 Kubernetes

Kubernetes is an open-source platform for automating deployment scaling and management of containerized applications. The platform provides features such as service discovery and load balancing, self-healing, automated roll-outs and roll-backs, secret and configuration management, storage orchestration, and automatic bin packing [45]. Due to it being widely used and open source, Kubernetes has a very active community advancing the technology and features [22]. To define how the Kubernetes cluster should look and operate, a state configuration is created. The state configuration will include information such as the number of replicas, resource requirements, environment variables, and other relevant parameters. Kubernetes continuously compares the desired state defined in the configuration with the current actual state [6]. This self-healing mechanism ensures high reliability and automation, eliminating the need for developers to manually manage the cluster's state.

2.3.1 Kubernetes Architecture and Components

The architecture of Kubernetes is ideally suited deploying and managing microservice-based applications, where the individual services are decoupled. The advantages compared to a monolithic approach are it becomes more resilient, flexible, and scalable and can utilize resources more efficiently [6]. A Kubernetes cluster is made up of nodes that are machines, either physical or virtual, that host the applications. It is divided into two different planes, the control plane and the application plane [18] which can be viewed in figure 2.1.

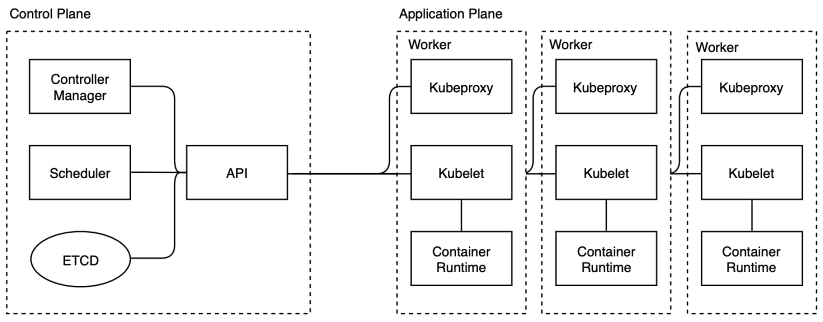

Figure 2.1: Architectural overview of a Kubernetes cluster

Control Plane

The control plane, also called master, handles the orchestration logic of the cluster and ensures that the desired state is equal to the actual state. There can exist more than one control plane in a Kubernetes cluster to achieve a more reliable cluster [25]. The control plane consists of the API server, etcd, scheduler, and controller manager [45].

- **API Server**: The API server is the front-end to all components of a Kubernetes cluster, serving as an entry point for all communications between different planes. Requests are authenticated and later handled to update corresponding objects in the etcd.
- **Etcd**: The etcd is an open source key-value store used to maintain configuration data and cluster state. It ensures consistency and availability to support correct scheduling and overall operating service [45].
- **Scheduler**: Initially when a pod is created it is unassigned, it is the role of the scheduler to assign these to nodes in the cluster. The assignment is based on the available resource metrics and pre-determined prioritization algorithms.
- **Controller Manager**: The Controller Manager is ultimately responsible for ensuring a correct and desirable state for its assigned cluster. It continuously checks the shared cluster state via the API server and attempts to rectify any discrepancies. Such rectifications may include redeploying dead pods, thus maintaining a desired state.

Application plane

The application plane consists of the worker nodes which maintain the pods. It is divided into Kubelet, Kube-proxy, and container runtime [45].

- **Kubelet**: The Kubelet is an agent that operates on every node in the cluster. It is responsible for managing containers deployed on the node and reports their status to the API Server. The Kubelet maintains communication with the API server to receive instructions on how pods should be running. It starts the containers within each pod by directing the container runtime to launch the corresponding container image.
- **Kube-proxy**: The Kube-proxy is responsible for managing network connections on deployed worker nodes. It ensures communication both internal and external to the cluster is correct and follows preset protocols.
- **Container Runtime**: Each node in the cluster has its container runtime engine which provides an environment to run the containers.

Pods

A pod in Kubernetes is the smallest unit of deployment. A pod is a collection of containers running in the same environment, where containers can share resources and communicate. Although applications in the same pod share attributes, separate pods are completely isolated from each other [6]. The selection of containers to be grouped together should be based on the specific applications and their respective usage requirements. Generally, it is not a good idea to group a simple stateless application in the same pod with a database container. The reason is that the stateful application will likely require more resources and being in the same pod will mean that they would be scaled equally.

Namespaces

Namespaces in Kubernetes are a way to isolate resources within a cluster and provide a way to organize and group objects into scopes, which makes managing and monitoring large and complex applications easier. The name of a resource must be unique within the same namespace but not across namespaces [45].

Containers in Kubernetes

Kubernetes serves as a platform for managing an application's life cycle, from deployment to maintenance. In order to achieve this, a container image needs to be built including the application and its required dependencies. The most widely used platform used to build container images is Docker [42]. The container image packages the application together with the dependencies necessary into one single artifact [6], which can then be used to run the application inside an OS container.

Kubernetes Controller

The controller is a core concept in Kubernetes which manages at least one resource. It utilizes the reconciliation loop that compares the current state with the desired state that has been defined in the controller's resource specification and makes adjustments accordingly [45]. The controller is closely related to the operator concept, which is the main subject of this thesis. An operator is used to manage a specific application or service, it builds on top of the controller concepts [45]. All operators are essentially a controller but not all controllers are operators.

2.3.2 External tools in Kubernetes

Kubernetes comes with a set of built-in tools to help manage a cluster, but an important part of the Kubernetes ecosystem is the tools that extend the functionality [8]. These tools can for example help with monitoring, management, and logging. Due to Kubernetes being such a widely used open source platform it has gained a large active community that contributes to its ongoing development and evolution [8]. Cloud Native Computing Foundation (CNCF) is a foundation subsidiary of the Linux Foundation that was created to help advance the technology surrounding cloud-native computing technologies [9]. One goal of CNCF is to foster cloud-native projects and create a collaborative platform for developers. Projects can become part of the CNCF ecosystem and gain one of three classifications: Sandbox, Incubating or Graduate. The level indicates the level of maturity and overall engagement of the project.

2.3.3 Helm Charts

Helm is a popular package manager for Kubernetes that simplifies the deployment of complex applications consisting of multiple components through the use of Helm charts. These charts function as templates that describe Kubernetes resources, streamlining their packaging and deployment [18]. Helm can also be used as an operator in the Kubernetes cluster to handle the lifecycle of an application or service. This is done by defining a chart for the application and then the Helm operator will watch for new Custom Resources (CRs) that correspond to the desired application state. The Helm operator will create and manage Kubernetes resources according to the defined YAML within the chart. The current operator that is currently in use at Sinch AB is a Helm-based operator. While Helm provides a convenient way to package and deploy Kubernetes resources, it may not be suitable for managing more complex stateful applications. Since it is limited by the underlying Helm technology.

2.3.4 Stateful Applications in Kubernetes

In a containerized environment, applications are commonly categorized as either stateful or stateless, based on whether they maintain and rely on persistent state information [1]. When a pod hosting a stateless application, such as a website, becomes unavailable, another instance can easily replace it without any lasting consequences for the application. Managing stateless applications is therefore fully possible and rudimentary using native Kubernetes capabilities [45]. However, dealing with stateful applications, such as a database implemented in a microservice architecture, presents a greater challenge, as each service has its own state that must be synchronized in the cluster and maintained in the event of a failure.

Kubernetes Native Stateful Handling

There are built-in resources in Kubernetes made for handling basic stateful applications, the prominent one being a StatefulSet. Unlike native Kubernetes types made for stateless applications, such as Deployment, a StatefulSet assigns labels to each pod that remain consistent meaning pods may maintain a consistent association with other Kubernetes resources [45]. Another notable feature of StatefulSet is the ordered creation and deletion of pods, which can be important for stateful applications requiring predictable behavior. In StatefulSet, storage is composed of Persistent Volumes (PVs) defined by Persistent Volume Claims (PVCs). PVCs contain information about the PV's characteristics and act as a request for storage while PVs represent the actual storage units assigned to a pod based on those requests. Each pod has its own dedicated PV, accessible only by that specific pod. This differs from the Deployment controller, where all pods share the same PV [1].

Unlike stateless applications that benefit from pod replication for improved availability, stateful applications mainly rely on repairing failed pods [1]. This is because each pod is isolated with its own storage and lacks awareness of the state of other pods within the application. StatefulSets typically perform well when applied to simpler use cases and applications, but suffer limitations in contexts of automation and high availability [34].

Kubernetes Add-ons for Stateful Handling

As mentioned previously there are many requirements put on applications offered through cloud services. Among which are requirements poorly coinciding with the capabilities of StatefulSets. This has motivated the development of alternative tools, focusing on the implementation of both operational and domain-specific knowledge into custom Kubernetes controllers. Such controllers allow for fine-grained control over even complex and stateful applications, offering capabilities reaching over those of StatefulSets [17]. The utilization of a custom controller within a Kubernetes cluster is referred to as a Kubernetes operator.

By leveraging a custom controller, an operator can customize the behavior and automation of clusters to meet the specific requirements of their stateful applications. Operators encapsulate operational best practices, application-specific logic, and automation, providing a higher level of control and customization, especially when compared to StatefulSets. Custom controllers can handle various tasks related to stateful application management, including advanced deployment strategies, automatic scaling based on application-specific metrics, intelligent failover, and recovery mechanisms, and seamless integration with storage systems and databases.

2.4 Open Source Software

The concept open source software (OSS) refers to software or code that is publicly accessible for anyone to modify and distribute [46]. The advantages of open source include increased transparency, flexibility, and collaboration. By allowing anyone to access and modify the source code, open source enables a wider community of contributors to innovate, improve, and customize the software to fit their specific requirements [32]. In the Kubernetes ecosystem, external tools that improve functionality and capability is an important role in the growth and success [6]. In their report, Nasserifar [32] emphasizes the importance of multiple actors in maintaining a healthy open source ecosystem. While end-users and bug reporters make valuable contributions, developers are considered niche players who has a central role within the ecosystem. Therefore, it is essential to create an environment that encourages developer involvement and recognizes their contributions.

2.4.1 Evaluating Open Source Software

In order to evaluate Open Source Software (OSS) in a structured way, different approaches can be used. Bolling and Gustafson [3] conducted interviews and found that users typically want to compare all available solutions before making a selection of what open source software package to use. Users normally evaluate the package in terms of security, future compatibility, and longevity. As well as more specific metrics such as downloads or how fast issues got resolved. The interviews also revealed that the evaluation process varied greatly depending on the size of the software. For bigger architectural decisions, such as selecting a framework, the process was more collaborative and longer. Bolling and Gustafson categorize the metrics into: popularity, activity, license, quality, and others.

Petrinja et al. [35] introduced the Open Maturity Model, which is an assessment model to evaluate OSS consisting of three layers: Basic, intermediate, and advanced. Through interviews, they identified the twelve most important metrics to consider to assess the quality of an open source product. Documentation, popularity, availability, maintainability, licenses, and technical environment were some of the metrics identified. Ardagna et al. [2] proposed a quantitative framework to evaluate and analyze security-related OSS systems, combining metrics into a single value for ease of use. They divide their metrics into six broad categories: Generic Aspects, developers community, users community, software quality, documentation, Interaction support, and integration, and adaptability with new and existing technologies.

In a literature review conducted by Lenarduzz et al. [24], the authors investigated 262 papers to analyze OSS models and identify commonly referenced metrics. The findings revealed that cost, support and service, license, code quality, and reliability were some of the recurrent cited metrics. In a similar empirical study, Zhao et al. [48], classified metrics into five types: code, license, popularity, developer, and sponsorship. They identified similar metrics as previously mentioned papers but also pointed out vulnerability and project activity as useful metrics. Additionally, Zhao et al. analyzed the correlation between different metrics and found that some metrics, such as project age, activity, status, and license, influenced the popularity of OSS projects.

In general, commonly identified metrics that were considered important by the mentioned studies were: Documentation, popularity, user community, maintainability, license, security, and compatibility.

Chapter 3

Operator Framework Study

The aim of this framework study is to provide comprehensive information for analyzing and comparing software frameworks, with a specific focus on Kubernetes operator frameworks. To achieve this, we will present an example Kubernetes environment as well as an in-depth view of the operator pattern.

3.1 WhatsApp Business API Client

This thesis is motivated by the existence of complex, stateful applications and the subsequent need for operators. To aid in understanding the decisions taken during the solution design chapter we present an overview of the WhatsApp client cluster and its components, a fitting example-application which Sinch AB has experience provisioning.

3.1.1 Architecture and Configuration

Unlike a majority of REST APIs, the WhatsApp Business API requires all customers to provision and manage the WhatsApp Business API Client as an on-premises service in order for it to access the WhatsApp internal systems. This structure is motivated by Meta's desire for messages on the WhatsApp service to be encrypted end-to-end, a major selling point for its end users. A new client cluster is provisioned for every customer wanting to communicate over the WhatsApp messaging channel, meaning maintenance of multiple customers quickly presents a repetitive and time-consuming administrative task. Each API client runs as a cluster of individual components, all run on separate machines or virtual machines. The final configuration setup of the API client may vary depending on each individual business constraint for service availability and wanted maximum throughput, but ultimately share major traits with each other. An architectural overview of the WhatsApp Business client may be seen in figure 3.1, where it also integrates into Sinch's as well as WhatsApp's internal systems [28].

3.1.2 Components

The client is built using three major node types. The first node type is the WebApp node which communicates directly with an external business service and dis-

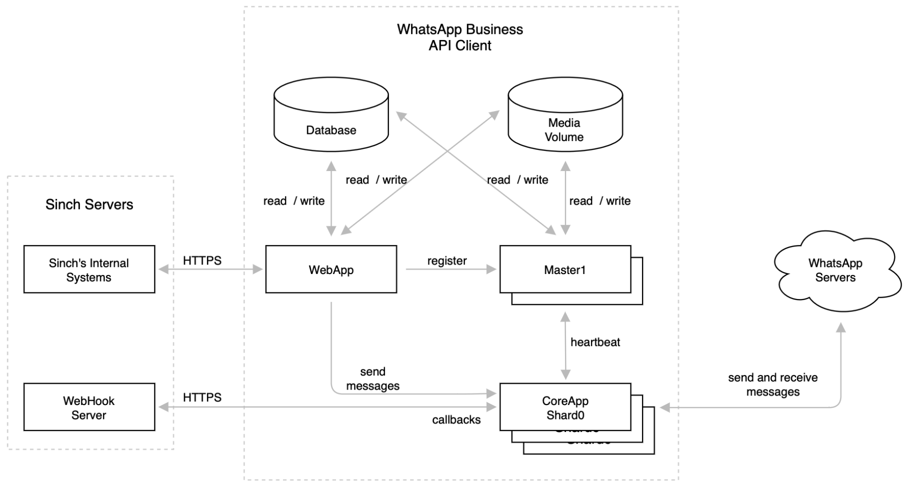

Figure 3.1: An architectural overview of the WhatsApp Business API Client

tributes messages. It responds to REST API calls with message statuses, storing relevant messaging information such as message content and contact information in storage nodes. The storage nodes are subcategorized into a database and a media volume, with the media volume handling media files transmitted to or from the client. The database instead stores data such as message content, relational contact information, and configuration details. The CoreApp node is deployed to facilitate communication between the WebApp node and WhatsApp's internal services, accepting REST API calls and forwarding resulting messages and returning messages to the business service. The configuration of CoreApps in the client is a key determinant of its conditions for high availability and load sharing to increase message throughput. CoreApps support sharding to effectively increase message bandwidth and avoid message backlogs in the system. Multiples of CoreApp nodes require master nodes for coordination and status checks. Seeing that the client maintains stateful data critical to its operation it may be considered a stateful application [28].

3.2 The Operator Pattern

A Kubernetes operator is an extension of the existing Kubernetes API, augmenting ad hoc domain knowledge and life-cycle management capabilities. Similar to Kubernetes controllers, they reside in the control plane of the Kubernetes architecture but differ in that they serve a function tailored to a specific application, instead of a general purpose [23]. The operator pattern was introduced as an abstraction to the task of implementing logic that was previously performed by a human operator managing running services. Maintaining the state of implemented infrastructure to ensure a productive and healthy system requires many repetitive activities and tasks, usually without lasting value to the system. This motivated the introduction of computerized intervention, enabling implementation of automated supervision and appropriate action-taking, completely devoid of human interaction. Kubernetes Operators offer intelligent, specialized, and dynamic application management capabilities aimed at maintaining a desirable state for even complex and intricate deployments. Figure 3.2 shows the general structure according to the Operator design pattern [18].

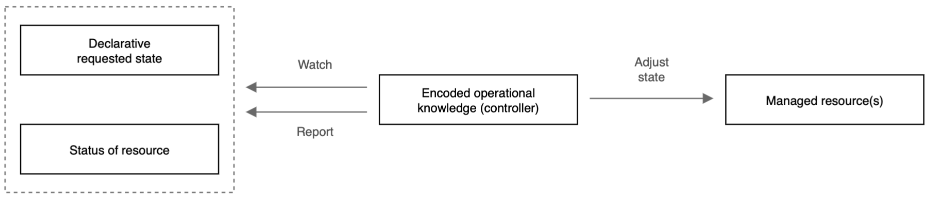
Declarative requested state
Status of resource
Watch
Report
Encoded operational knowledge (controller)
Adjust state
Managed resource(s)

Figure 3.2: The Kubernetes Operator design pattern

3.2.1 Operator Components

The increasing complexity of cloud-hosted applications and the growing responsibility for managing them have led to higher demands for availability, fault tolerance, and scalability. To meet these demands, the operator pattern relies on declarative configuration, allowing a desired state of a resource to be defined and maintained [23]. Figure 3.3 illustrates the main components of the operator pattern at a high level: the managed application or infrastructure, a mechanism for defining declarative, domain-specific requirements for a desired state, and a custom controller that reads the current state and initiates actions based on a reconciliation loop. There are three major components involved in building a Kubernetes operator.

- **Custom Resource Definition**: A Custom Resource Definition (CRD) is a user-specified object schema that is not provided by native Kubernetes. It defines the characteristics of a custom resource type, including its name, field structure, and behavior. The definition is supplied in YAML or JSON format and can be dynamically inserted into a running Kubernetes environment.
- **Custom Resource**: Custom Resources (CRs) enable the extension of the Kubernetes API CRDs. Within a running cluster, CRs can be dynamically installed to expand the existing suite of types, including pods, deployments, or services. Once installed, a CR can be queried and manipulated through the Kubernetes API, similar to Kubernetes native types.

- **Controller**: The controller is responsible for implementing all decision-making and automated orchestration within a Kubernetes Operator. It executes this logic using a reconciliation loop, which can be triggered by specific events or run at predefined intervals. During each iteration of the loop, the controller accesses the state of the objects it monitors and ensures they align with their declared and desired state. Controllers are a native feature of Kubernetes, but when deployed to handle custom resources, they become custom controllers. The ability to manage user-defined resources empowers controllers to administer clusters using domain-specific knowledge [27].

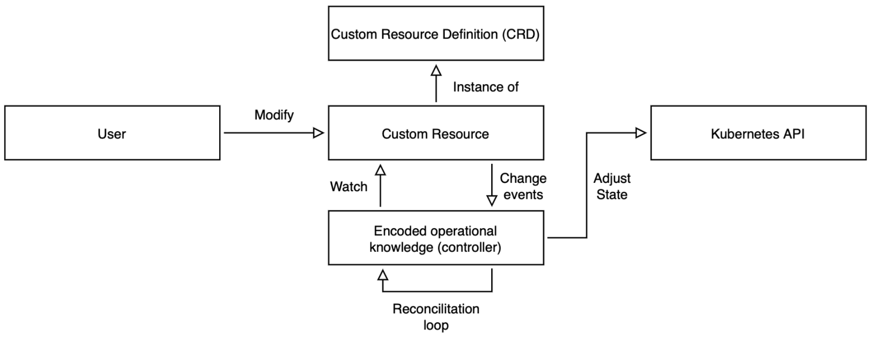

Figure 3.3: Typical workflow for Kubernetes Operator

3.2.2 Benefits of using the Operator Pattern

As previously stated, an operator's underlying purpose is to extend the capabilities of the Kubernetes API, making it specifically tailored to an existing infrastructure and ensuring a functional and desirable state. This addition to native Kubernetes is largely necessary due to its inability to otherwise proficiently manage the life-cycle of complex and especially stateful applications [25]. The supplemented features may further be sub-categorized into dynamic configuration, operational automation, and extended domain knowledge. The first of which refers to cluster configuration through Kubernetes operators' interaction with custom resources.

Native Kubernetes provides built-in tools to interact with and query configuration files, the most commonly used are ConfigMaps and Secrets. However, such solutions are often inadequate as they are designed for generic application purposes and lack the ability to provide detailed specifications [18]. An operator may declare and subsequently query a custom resource to express a particular application configuration in a Kubernetes context. This allows operators to dynamically configure the underlying infrastructure based on the specific requirements of the application being deployed. This dynamic configuration capability is one of the key benefits of using Kubernetes operators, as it allows for more fine-grained control over the application's environment, resulting in better performance, reliability, and scalability.

Operational automation is another key benefit of Kubernetes operators. By automating the deployment, scaling, and management of complex applications, operators may reduce the burden on human operators, freeing them up to focus on more strategic tasks [18]. This is particularly important in large-scale environments, where managing the life-cycle of multiple applications may quickly become overwhelming. Finally, Kubernetes operators may provide extended domain knowledge by leveraging the expertise of developers and operators who are intimately familiar with a particular application or technology stack. By encapsulating this knowledge in the form of an operator, organizations may ensure that their applications are deployed and managed in a way that is optimized for their unique requirements [18].

3.2.3 Operator Maturity

The quality of an operator has a significant impact on metrics regarding performance for its intended application, including availability and overall quality of service. In an effort to aid in quantifying the proficiency and capability of an operator a maturity scale was devised. Based on the used framework and implementation of an operator it is said to achieve varying maturity levels. The maturity scale can be seen in figure 3.4 and comprises five set levels covering basic install, seamless upgrade, full lifecycle, deep insights, and autopilot [19].

- **Basic Install**: Basic install allows for simple install and workload configuration via pre-defined Custom Resources, the very basics for any operator.

- **Seamless Upgrade**: Seamless upgrade allows for version upgrading without loss of service or data loss.

- **Full Lifecycle**: Reaching the level of full lifecycle indicates the operator manages backup and fail-over duties. It then automates processes such as data backup and restorations of failed pods

- **Deep Insights**: Deep insights refer to the operator's ability to monitor and alert the health statuses of its application components. This may be aided by external software such as Prometheus, a prominent open-source monitoring solution.

- **Auto Pilot**: Auto pilot is the highest level of the maturity scale. This level requires the operator to maintain a stable and efficient application lacking human intervention. Relevant to cloud services is the operator's capabilities of application scaling, done either using horizontal or vertical scaling.

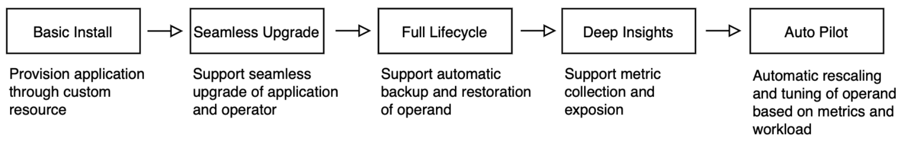
Basic Install
Provision application through custom resource
Seamless Upgrade
Support seamless upgrade of application and operator
Full Lifecycle
Support automatic backup and restoration of operand
Deep Insights
Support metric collection and exposion
Auto Pilot
Automatic rescaling and tuning of operand based on metrics and workload

Figure 3.4: Five maturity levels for operators

3.3 Operator Frameworks

Frameworks are frequently employed in all areas of software development. Such frameworks are characterized by a set of established practices, guidelines, and standards that offer a structured and swift approach to development. By leveraging pre-built structures and components designed to handle general case problems, frameworks simplify and streamline the development process. While Kubernetes operators were introduced fairly recently to the Kubernetes ecosystem, they still show widespread adoption of frameworks being used to aid their development. Operators make use of large parts of the Kubernetes capabilities exposed by its API, a luxury largely facilitated by frameworks providing a layer of abstraction to the API's otherwise complicated nature. This abstraction releases operator implementation from intricate and complex solutions for integrating to stock Kubernetes, allowing them instead to focus on the more relevant domain-specific logic. As demonstrated in the above sections both Kubernetes and its extending operator pattern are complex, multifaceted entities requiring deep and specific technical knowledge. Different frameworks pertaining to operators were developed in an effort to counter the high barrier to entry.

3.3.1 Qualities of Operator Frameworks

Operator frameworks have several central qualities defining both their potential use cases and capabilities. There are numerous existing operator frameworks available, which can make the selection process challenging for specific scenarios. A major point of contention for operator frameworks is their allowed operator complexity. There exists a trade-off between allowing operators developed through a framework to exhibit the desired capability and maturity and the difficulty of actual implementation. This often takes shape as the development method and the support language offered by a framework.

The majority of operator frameworks share the trait of being open source, meaning the existence of a willing and able developing group is needed. Open source development is a common and strategic development tactic for framework development seeing as frameworks greatly benefit from qualities common in open source projects such as high transparency, flexibility, and a lack of vendor lock-in. The development tactic does however expose the project to pitfalls of community engagement and a lack of structured leadership and contribution. They are additionally, by standard practice released under licenses that define the permitted usage, modification, and redistribution of the framework. If unsuitable, the license may affect or even strictly prohibit the use of a framework, especially applied to business settings.

3.3.2 Operator Framework Availability

As open source projects the operator frameworks are mostly available to find and explore online, either through the version-control platform GitHub or through dedicated distribution channels such as CNCF themselves.

Chapter 4

Solution Design

This chapter will present the proposed solution design, define what attributes were considered appropriate for evaluation as well as present the evaluation profiles. It will also introduce a selection of currently available operator frameworks.

4.1 Proposed Operator Framework Evaluation Model

This thesis aims to perform an evaluation and subsequent comparison of operator frameworks, all with varying origins and stated goals. To accomplish a yielding experiment, we defined an evaluation process based on findings in the literature study. The evaluation process is based on the paper written by Morisio et al. [31], introduced in section 1.4.2. The evaluation was to encapsulate all relevant observations in order to produce a model able to be used for comparison. The process comprised two phases, the definition of the evaluation model and the measuring and data collection and aggregation.

Phase 1: Definition of evaluation model

1. Definition of model attributes
2. Definition of attribute metrics

Phase 2: Definition of Metrics and Data collection

1. Definition of profiles
2. Measurement collection
3. Aggregation of measures

From investigation of operator frameworks and the performed literature study we concluded two main aspects should be considered for the evaluation model, the first being *operator capability* and the second being *open source project health.* Operator capability would include all notions of the potential quality of the implemented operator. This would encapsulate its capabilities, implementation process, use cases, and any restrictions placed on the developing party. The open source project health would instead focus on the health status of the project, with respect to operator frameworks being sustained through open source development. The model aimed for a standardized list of attributes and while there are numerous available, only a few were deemed attainable and relevant enough. Also due to the diversity of the framework application, notice was taken to select attributes that were applicable to all. Further motivation to the attribute selections is provided in section 4.1.1.

4.1.1 Proposed Attributes

Attributes suitable for the evaluation model were informed by the previous literature study. Some attributes were regarded as the industry standard for evaluation of software and therefore included, others were by us deemed necessary based on previous learning. By considering a wider range of attributes, we aim to provide a holistic assessment framework that facilitates effective comparison and selection of operator frameworks for various use cases. Each attribute was to be accompanied by a single or several relevant metrics, making them quantitatively measurable and comparable. The final evaluation model may be seen in figure 4.1. Each attribute is later justified, highlighting its relevance and motivating its inclusion in the model.

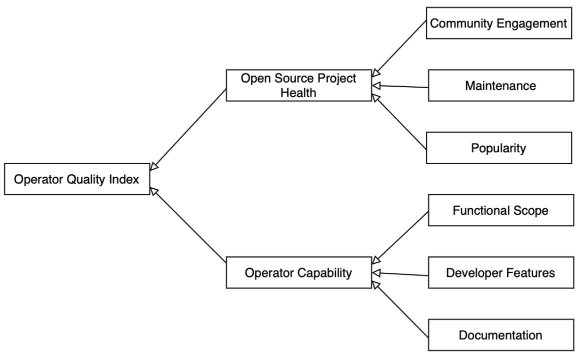

Figure 4.1: Proposed Evaluation Model with Accompanying Attributes

Community Engagement

Ardagna et al. [2] lists user community as one of the six core macro areas in their framework model. The success of open source projects depends heavily on the support and engagement offered by its community. While the projects often are launched and maintained by groups or organizations community involvement is crucial for further advancement at every stage, including design, development, and testing.

Maintenance

Maintenance is an essential aspect of software development that ensures that the framework continues to function as intended and remains secure. Bolling and

Gustafson [3] study showed that active development is an important aspect according to several developers. Maintenance involves fixing bugs, implementing new features, and updating dependencies. The level of maintenance required depends on the complexity and scope of the framework, as well as the rate of changes in the environment in which it operates. The quality of maintenance may have a significant impact on the usefulness and longevity of a framework and should be a consideration when evaluating software options.

Popularity

Popularity is a metric used for easy understanding of a framework's market dominance, often an indication of its perceived value and usefulness within the development community. According to Bolling and Gustafson's thesis [3], interviews revealed that project popularity was often the primary metric considered when selecting an open source package. A more popular framework typically indicates that it has been more thoroughly tested, making it a more secure choice.

Documentation

Effective software documentation plays a vital role in ensuring that public, open source software is accessible and comprehensible to its intended audience. In a literature review done by Lenarduzzi et al. [24], documentation was one factor that was taken into consideration when opting to use an open source project. Documentation should include any and all relevant information for understanding a software entity's qualities such as intended use, technical requirements, or guides for deployment. Documentation may therefore be a crucial point of consideration when selecting what software to use.

Functional Scope

Implemented operators are burdened with requirements in both functionality and policy compliance. They are often employed in heavily controlled environments where rules restricting the use of licenses, programming practices and languages and package dependencies exist. The functional scope of an operator framework should therefore convey its feasibility of use pending the requirements set on its environment.

Developer Features

What features a framework provides can potentially be an important aspect to consider when selecting a framework. In accordance with the Open Maturity Model proposed by Petrinja et al. [35], it is recommended to consider the framework's technical environment, which encompasses various aspects such as tools and development environment in their model. By examining the available features and tools, developers can not only save time but also ensure that they adhere to best practices throughout the development process.

4.2 Attribute Profiles

For evaluation and comparison of different frameworks a systematic metric aggregation method was required, as argued by Morisio et al. [31]. To accomplish this, a scaling method was adopted in which a set number of levels ranking each metric was introduced. The levels were to each other rankable in order. Morisio et al. introduce the notion of having three to five attribute categories that are suitable for the supplied methodology. This thesis opted for four levels being *Acceptable* (A), *Good* (G), *Very Good* (VG), and *Excellent* (E) in which A < G < VG < E. The ordinal levels allow for a standardized and consistent assessment of the frameworks. Each level represents a minimum measurement required to reach it, if not the value falls into its lower neighbour. The levels and scales are also defined separately for each metric of the model. This model will encompass scales of different types, both nominal and absolute and while some metrics are placed pending a numerical measurement, some are instead placed by qualitative evaluation or subjective judgment. This will be made clear when metrics are decided and presented.

4.3 Considered Operator Frameworks

This study will implement its evaluation model on a select number of operator frameworks. Following is a compiled list of operator frameworks currently available accompanied by their respective motivation for use supplied by their chief maintainer. This will lay grounds for future operator framework filtering. The listed frameworks are not the only ones available but the ones that we found mentioned in sources about Kubernetes operators. The list is meant to provide an overview of the breadth of the current scene of operator frameworks. Not all will be included in the performed experiment.

Operator Framework

The Operator Framework was first introduced in 2016 by CoreOS but is now currently managed by Red Hat. The framework provides libraries, tools, and best practices to simplify the process of implementing operators for more complex applications. It consists of three main components [17]:

- The Operator SDK: The main part of the Operator framework which is used to build the operator, providing developers with scaffolding tools, high-level APIs, and abstractions to build operators efficiently. It is built on top of the Kubernetes controller-runtime, which is a set of libraries for building controllers.
- Operator Lifecycle Manager (OLM): A declarative way to install, build and manage the operator on the cluster.
- OperatorHub: Provides a platform to distribute finalized operators with the open source community.

The Operator SDK provides three different ways to build an operator [33]:

- Helm: This option allows you to use Helm charts to define your Kubernetes resources and manage their lifecycle.
- Ansible: This option allows you to use Ansible playbooks to manage the lifecycle of your Kubernetes resources. Ansible is an automation tool that is widely used in IT operations and allows for the management of complex deployments and configurations.
- Go: This option allows you to write operators in the Go programming language. This provides a flexible option for building operators, as you have direct access to the Kubernetes API and can write custom controllers to manage your resources.

Kubebuilder

Kubebuilder is an open source framework used to build Kubernetes APIs using Go. By providing various tools and libraries, Kubebuilder streamlines the process of building controllers and customer resources [38]. Additionally, Kubebuilder offers a local development environment for testing purposes. Kubebuilder is extensible and it is embedded within Operator SDK in the Operator Framework. This means that a Kubebuilder project is also compatible with the Operator SDK.

Dotnet Operator SDK

Dotnet Operator SDK was inspired by Kubebuilder but is based on dotnet, making it a suitable option for developers who prefer working in C#. It provides a set of tools and libraries which simplifies the implementation of an operator. The motivation behind the project according to their own documentation was to provide an alternative to the Go-based frameworks like Kubebuilder and Operator SDK [5].

Shell Operator

The Shell Operator provides tools to run event-driven scripts within a Kubernetes cluster. The framework is tailored to system administrators by providing a way to implement operators in a way that they are familiar with [20]. It leverages the flexibility of shell scripts and treats them as hooks that are triggered by specific events in the cluster [11]. The Shell operator listens to certain Kubernetes events and executes the hooks when these events are triggered. A hook is an executable file written either as a script or compiled program in any programming language.

KUDO

Kubernetes Universal Declarative Operator (KUDO) is a framework that streamlines the development of a Kubernetes operator by utilizing mostly declarative YAML-based configuration while most other frameworks require written code [15]. KUDO offers various abstractions and APIs and also features both a command line interface and a web-based user interface that enable the management and monitoring of operator deployments. The operator is made up of several *plans* that act as a runbook. Each plan is made out of different phases which can contain multiple steps. A phase can for example include installation, upgrade and uninstall.

Kopf

Kopf (Kubernetes Operator Pythonic Framework) is a Python framework for building Kubernetes operators. It was developed by Zalando in 2019 to address the high entry barrier that existing operator frameworks had, which required significant amounts of boilerplate code to be written [40]. Zalando wanted to create a framework that would allow developers to create operators more easily by only displaying high-level concepts. Kopf has a set of Python decorators that defines actions the operator should take when an event occurs, such as a pod getting created, deleted, or updated.

Java Operator SDK

Java Operator SDK is a higher-level framework for building Kubernetes operators using Java. Allowing developers to focus on the operator's business logic instead of focusing on low-level interactions with the Kubernetes API. The goal of the framework is to simplify operator implementation by providing an API that is familiar to Java developers to define custom resources and controllers [12].

Quarkus Operator SDK

Quarkus Operator SDK is a Java-based operator framework. It is an extension to Java Operator SDK, therefore, providing additional features simplifying the implementation of operators [16].

kube-rs

Kube-rs is a CNCF-hosted project that enables the implementation of Kubernetes operators using the Rust programming language. By abstracting away the complexities of interacting with the Kubernetes API, Kube-rs simplifies the process of building controllers and custom resource definitions (CRDs) [14].

Charmed Operator Framework

The Charmed Operator Framework provides a Python library to streamline the operator implementation. The motivation behind the project was to simplify the process for Python experienced developers to create an operator [13]. The Charmed Framework also contains features for testing operators as well as a hub called CharmedHub.io, where developers can share their finished operators.

Chapter 5

Solution Implementation

This chapter aims to explain how the metrics for comparing Kubernetes operator frameworks were defined and later collected. It will provide an in-depth view of our methodology for metric data collection, including the selection of appropriate data sources, tools, and techniques for gathering such information. The primary goal of this section is to motivate how the metrics selected for evaluating the frameworks were reliable, accurate, and representative of the frameworks' effectiveness and their practical difficulties.

5.1 Initial Filtering of Operator Frameworks

Despite being a relatively new concept in Kubernetes, there is a wide range of frameworks available to use when designing and creating operators. To conduct a comprehensive and yielding experiment, we first wanted to filter these tools. The remaining list of frameworks should provide a good representation of the current market for operator frameworks, but also be concise enough to perform a thorough investigation. Section 4.3 presented the basis for the selection process and was considered a starting point for determining which frameworks to examine. Furthermore, frameworks should have been ordered by relevance to the current development scene of Kubernetes operators, with framework popularity in terms of GitHub statistics being a main point of consideration. Additionally, the study prioritized frameworks with varying origins, utilizing different technologies and having diverse stated intentions. This was done to provide as broad a coverage of the current market as possible, making the experiment a utilizable and topical evaluation.

5.2 Employing Goal-Question-Metric

With a proposed model outlining all attributes to be considered when evaluating operator frameworks the next step was finding metrics. This process involved defining metrics that were both available and measurable, accurately representing the attributes of the model. To aid in methodologically choosing metrics suitable for attributes in the model, the Goal-Question-Metric (GQM) practice was implemented. This approach involves iteratively formulating questions related to predetermined goals in order to generate metrics [7]. Following is a demonstration of how metrics were chosen using the GQM method.

The goal was defined.

G. *"Assess the health status of an open source project"*

From the stated goal a set of investigatory questions were formulated.

Q. *"Is the open source project maintained sustainably?"*

The questions were perceived as second iteration goals, following a similar pattern to their predecessor, the initial goal. As so, they were repetitively reformulated into more specific and segmented goals and following questions.

G. *"Evaluate the maintenance of an open source project"*

Q. *"Are known code issues being actively addressed?"*

Finally leading to clearly and easily interpretable metrics, derived from the objectives.

M. *"The quotient of number of resolved issues and the number of posted issues"*

The above example details a single branch of objectives stemming from the original question. The process was repeated exhaustively until all attributes had clear and descriptive metrics.

5.3 Defined Metrics and Metric Collection

As described previously, the evaluation model is defined with attributes measured by appropriate and available metrics. This section outlines the metrics established through GQM as well as the methods used to collect and measure them.

5.3.1 Community Engagement

Community Engagement was deduced to be measured best through statistics of stakeholder and patron interaction and involvement. The metrics were specifically designed to measure the level of interest in learning and or teaching about the framework and the level of interest in actively contributing to its code base. Metrics for community engagement were sourced from the framework's hosting service, therein the number of project contributors and the total number of posted issues. Additionally, data was retrieved from GitHub discussions and the question-and-answer platform, Stack Overflow. Specifically, the total number of questions related to the framework and the corresponding number of questions that received an accepted answer was collected. The primary focus was to retrieve the number of posts on the framework's GitHub discussion. If that was unavailable, Stack Overflow was used as an alternative source.

Another approach to measure community engagement is to consider the range of community platforms and communication channels provided by the framework. These platforms can take various forms and provide diverse opportunities for interaction. This comparison will focus on the presence of weekly meetings and mailing lists, as well as whether the project is part of the CNCF (Cloud Native Computing Foundation) ecosystem. Additionally, the existence of official channels on the communication platforms Slack and Discord will be assessed. The metrics are presented in table 5.1.

Attribute	**Metric**
Community Engagement	# of issues posted on GitHub repository
	# of contributors on GitHub repository
	# of public forum posts on topic
	# of public forum posts on topic w/ accepted answer
	Community channels

Table 5.1: Found Metrics for Community Engagement

5.3.2 Project Maintenance

The maintenance score of a framework was made to capture the framework's current health, as well as provide an indication of its potential future prospects. This attribute was measured through statistics pertaining to the framework's repository status and visible trends. For this, we measured the number of active contributors and lines of codded added, modified, or removed in the past 6 months. In addition, the relation between resolved and total issues, mean issue resolution time, and the mean contributor lifetime was deemed relevant and valuable.

Since all frameworks are hosted on GitHub, the metrics could be fetched with the help of GitHub's API. The relation between resolved issues and total issues was simply calculated as the current balance between the number of posted issues and the number of resolved issues. The resolution time was captured as the mean time for a posted repository issue to be marked as resolved, calculated as the time difference between its post time and its resolved time. The contributor life team was calculated as the mean time a contributor was active in the project. Both mean contributor lifetime and mean issue resolution time was measured in number of days. A table showing resulting metrics may be seen in table 5.2.

Attribute	**Metric**
Project Maintenance	# of active contributors (6 months)
	# of LOC added, modified, or removed (6 months)
	Quotient of resolved issues and total issues
	Mean issue resolution time
	Mean contributor lifetime

Table 5.2: Found Metrics for Project Maintenance

5.3.3 Project Popularity

Popularity was considered to be the culmination of publicly available statistics regarding the extent of a framework's usage and digital footprint. This included a selection of relevant data available on their hosting service, being Github. The data was gathered by employing a Python script fetching repository data from GitHub's public API. A table for all collected metrics may be seen in table 5.3.

Attribute	**Metric**
Project Popularity	# of stargazers on GitHub repository
	# of forks on GitHub repository
	# of used by on GitHub repository
	# of watchers on GitHub repository

Table 5.3: Found metrics for Project Popularity

5.3.4 Documentation

Documentation was made to take into account the extent of the provided documentation, the percentage of commenting in supplied code-scaffolding, and what criteria were achieved by the supplied documentation. The amount of documentation was measured by compiling all relevant code documentation provided by the framework maintainer and counting the total amount of words using a Python script. Comments on code were also retrieved using a Python script. It was calculated as the quotient of lines of code and lines of code containing comments using the provided source code for the investigated frameworks. To exclude non-relevant code, only the files with languages relevant to the framework were included. The documentation criteria used were formed based on the perceived needs of a developing party wanting to undertake the creation of an operator, as well as asking experienced operator developers on forums such as Slack and Discord. Some criteria were industry standards, generally accepted to be necessary in software documentation and others were instead found needed for the specific use case.

- Section: Quick start/Getting started
- Section: Tutorials
- Section: System architecture
- Dedicated web page
- Example operator implementation(s)
- Documentation has been updated within a year

A table containing the collected metrics may be seen in Table 5.4.

Attribute	**Metric**
Documentation	Quotient of LOC containing comments and total LOC
	# of words in compiled documentation
	# of documentation criteria fulfilled

Table 5.4: Found metrics for Documentation

5.3.5 Functional Scope

The functional scope became largely based on known limitations and requirements set on operator frameworks. We deduced it encompassed what level of operator maturity, introduced in section 3.2.3, an implemented operator was theoretically able to reach using the framework, the permissiveness of the license shipped with the framework, and the number of dependencies. The maturity level was evaluated by analyzing the documentation and finished operators publicly available for each framework, as well as talking to the developers of the framework via Slack or email. The number of dependencies, along with licenses, were sourced from the GitHub repository of the project. The resulting metrics can be seen in figure 5.5

Attribute	Metric
Functional Scope	Reachable level of operator maturity model
	Level of License permissiveness
	# of subsidiary dependencies

Table 5.5: Found metrics for Functional Scope

5.3.6 Developer Features

Developer features can aid the development of an operator and lower the entry barrier to using a framework. To assess the features provided by the framework, we evaluated the presence of a testing environment, if a custom-made Command Line Interface (CLI) existed, and the availability of code-generation tools. The code generation tools will often provide boilerplate code for new resources. For this comparison, we limited the code generation tools to the following functionalities:

- Project initialization
- CRD manifest generation

To evaluate the presence of a testing suite, CLI, and available code generation tools, we examined the framework's documentation. The table for showing the resulting metrics can be seen in figure 5.6.

Attribute	Metric
Developer features	Extensiveness of code generation tools
	Provided command line interface
	Provided testing suite

Table 5.6: Found metrics for Developer Features

5.4 Profile Definition

Since we used threshold-based techniques to rate and profile the selected metrics, we needed to determine what thresholds to use. There are generally two accepted techniques used to derive thresholds (a) expertise-driven and (b) data-driven. The former relies on the domain knowledge and previous experiences of, to the subject, well-informed and knowledgeable parties. Such thresholds may be found in literature alternatively produced via active consulting of experts. For thresholds where this was applicable thresholds were collected as off-the-shelf, established values. However many metrics used in this thesis lacked previously defined ranges and were therefore subject to the data-driven technique, a technique backed by previous papers [49]. For this the profiles were defined by dividing the collected ranges of data into pre-defined percentiles being [P0, P20], [P20, P50], [P50, P80], and [P80, P100], correlating to the profiles A, G, VG, and E.

For metrics that were not explicitly defined by specific retrieved values, such as developer features, license, and community platforms, customized thresholds were established to determine profile ratings. The same was done for maturity level and documentation criteria. In the case of licenses, frameworks with more permissive licenses were assigned higher profiles, as they tend to attract greater attention. Similarly, for community platforms, frameworks offering a wider range of platforms to engage with the community received higher profiles. For developer features, if a custom CLI and a testing environment existed it was given the highest profile, otherwise placed at the lowest. Code generation tools were given a profile based on the extensiveness of the tools provided. If both initialization tools along with CRD generation tools were provided it would be given the highest profile.

5.5 Aggregation

The proposed evaluation model composes a hierarchical tree structure in which attributes form branches, while their various metrics make up corresponding sub-branches. Measurable evaluation is only performed for the metrics offering little in the way of high-level concept evaluation and comparison. To rectify this it was necessary to establish an aggregation method, decomposing evaluation of sub-branches into a unified measure for their respective main branch. This aggregation would transform the easily measurable values of the metrics into higher-level concepts that are more readily understandable and usable for the purposes of this thesis.

5.5.1 The Outranking Relation

In order to achieve a single resulting profile for attributes comprised of multiple metrics an aggregation procedure was implemented. The method was inspired by the ELECTRE-TRI methodology and has its basis in establishing an outranking relation S, read as *is at least as good as* for each attribute and profile. The outranking relation S is said to hold should both a concordance C and a non-discordance $\neg D$ test be satisfied, such that

$$S(a,b) \leftrightarrow C(a,b) \wedge \neg D(a,b)$$

- The *concordance test* simply uses the weighted majority rule to decide between the preference of an element and a profile. More precisely, it determines whether the combined weights for all metrics *at least as good as the* *profile* exceed the concordance threshold. All weights were for our experiment considered to be equal, with a value of $1/n$, where n represents the total number of metrics composing the attribute. The threshold was set to 0.6, a commonly used value in literature [31].

- The *non-discordance* introduces a complementary notion that stating no metric minorities strongly conflicts the imposed relation. Formally this is known as the metrics right to veto the relation S by exhibiting strong enough disagreement with it. A veto is present should the difference between a metric m_i from the attribute and the profile threshold p_i be larger than an accepted limit v. Formally $m_i - p_i > v$. Since our scales are not exclusively numerical, the threshold was instead interpreted as exceeding the limit of two profiles below the currently evaluated. In other words, if an attribute was evaluated against profile P_i a metric assigned P_{i-2} would veto the evaluation.

Should an attribute a achieve the outranking relation for a profile p, such that it may be said that *a is at least as good as p*, a is assigned that profile. Should it hold true for multiple profiles, the best profile is always chosen.

5.5.2 Aggregation through Profile Average

To later aggregate profiled attributes in order to produce a profile for the overarching categories *operator capability* and *open source project health* a simple averaging method was employed. Each profile was assigned a numerical value later summarized for all attributes in a category and divided by the number of categories. This achieved a final number corresponding best to one of the profiles, being the final aggregated grade.

Chapter 6
Results

This chapter will present all findings produced by the previously described solution implementation. Firstly we will present the finalized model profiles as well as the frameworks resulting from filtration. Later the results from the implemented model for each framework will be presented, finally achieving grounds for evaluation and comparison.

6.1 Filtering

In Section 4.3, several operator frameworks currently available for use are introduced. However, not all of them were examined in this experiment, leading to the need for a filtering process. Following the filtration, six operator frameworks were identified and selected for further analysis: *The Operator Framework*, *Shell Operator*, *KUDO*, *Kopf*, *Java Operator SDK*, and *Kube-rs*.

6.2 Attribute Profiles

Chapter 4 outlines the evaluation model, emphasizing the need for attribute profiles that are detailed, clearly separable, and capable of being ordered on a semantically appropriate preference scale. Table 6.1 and 6.2 display the resulting profiles for each attribute's underlying metrics as well as its accompanying preference scale. Each profile threshold will carry with it notation telling whether it was found through data-driven (*) or expertise-driven techniques (†).

6.3 Framework Evaluation

The selected frameworks were all evaluated, in accordance to the previously described methodology. For each metric a value was appointed, either stemming from empirical measurements alternatively by expertice based judgment. Following are the results for operator capability as well as open source health. The tables presents the attributes' multiple metrics as well as their individual profile thresholds and appointed scale. The identity scale indicates that greater numerical values are preferred, while its inverse represents the opposite. A profile scale simply follows the ordinal preferences layed out in section 4.2.

Attribute	Metric	[A, G, VG, E]	Scale
Documentation	Comments of code	[0.049, 0.086, 0.140, 0.195]*	Identity
	# of words in documentation	[6965, 20713, 41334, 61955]*	identity
	# of doc. criteria	[2, 3, 5, 6]†	identity
Functional Scope	Maturity level reachable	[1, 3, 4, 5]†	identity
	Licence permissiveness	[A, G, VG, E]†	profile
	# of subsidiary dependencies	[436, 357, 238, 119]*	inverse
Developer Features	Code generation tools	[A, G, VG, E]†	profile
	CLI	[no, yes, yes, yes]†	no<yes
	Testing suite	[no, yes, yes, yes]†	no<yes

Table 6.1: Resulting profiles for attributes: Documentation, Functional Scope and Developer Features

Attribute	Metric	[A, G, VG, E]	Scale
Community Engagement	# of issues	[130, 605, 1317, 2029]*	Identity
	# of contributors	[33, 91, 179, 267]*	Identity
	# of public forum posts	[12, 32, 62, 92]*	Identity
	# of public forum posts (waa)	[2, 7, 14, 21]*	Identity
	Community channels	[A, G, VG, E]†	profile
Project Maintenance	# of active contributors	[0, 5, 12, 19]*	Identity
	# of LOC changed	[0, 4376, 10939, 17502]*	Identity
	Resolved issues quotient	[0.72, 0.77, 0.84, 0.91]*	Identity
	Mean issue resolution time	[94, 86, 74, 61]*	Inverse
	Mean contributor lifetime	[44, 81, 136, 190]*	Identity
Popularity	# of stargazers	[585, 1768, 3543, 5317]*	Identity
	# of forks	[101, 421, 901, 1380]*	Identity
	# of used	[7, 874, 2174, 3473]*	Identity
	# of watchers	[15, 37, 70, 103]*	Identity

Table 6.2: Resulting profiles for attributes: Community Engagement, Maintenance, and Popularity

6.3.1 Evaluation of Operator Capability

Tables 6.3 and 6.4 display the collected metrics for operator capability along with the assigned profiles, where the resulting value for each metric is presented in a performance array. The evaluation stems from the multi-criteria aggregation method explained in section 5.5. In the interest of table-space the presentation of the group of frameworks were divided into two, otherwise identical tables. The tables identifies *The Operator Framework*, *Kopf* and *Kudo* as the most favourable frameworks with respect to operator capability. The evaluation suggests no strong opposing metric to their assignment of *Very Good*, and we may therefore assume them to be well rounded and capable frameworks. The results of the evaluation provide a compelling justification for considering The Operator Framework, Kopf, and Kudo as top choices when evaluating frameworks for potential use.

Attribute	**TOF**	**Shell Operator**	**KUDO**
Documentation	[0.158, 75703, 5]	[0.064, 6965, 4]	[0.087, 19593, 5]
Eval.	*Very Good*	*Acceptable*	*Good*
Functional Scope	[5, E, 436]	[5, E, 117]	[3, E, 40]
Eval.	*Good*	*Excellent*	*Very good*
Developer Features	[E, Yes, Yes]	[A, No, No]	[G, Yes, Yes]
Eval.	*Excellent*	*Acceptable*	*Very good*
Operator Capability	**Very good**	**Good**	**Very good**

Table 6.3: Evaluation of operator capability for *The Operator Framework (TOF)*, *Shell Operator* and *KUDO*

Attribute	**Kopf**	**JOSDK**	**Kube-rs**
Documentation	[0.205, 33070, 6]	[0.0497, 14426, 5]	[0.232, 23114, 4]
Eval.	*Very good*	*Acceptable*	*Good*
Functional Scope	[5, E, 52]	[5, E, 100]	[5, E, 161]
Eval.	*Excellent*	*Excellent*	*Excellent*
Developer Features	[A, Yes, Yes]	[G, No, No]	[G, No, No]
Eval.	*Good*	*Acceptable*	*Acceptable*
Operator Capability	**Very Good**	**Good**	**Good**

Table 6.4: Evaluation of operator capability for *KOPF*, *Java Operator SDK (JOSDK)* and *Kube-rs*

6.3.2 Evaluation of Open Source Health

Tables 6.5 and 6.6 present the results for open source health using the same evaluation method employed for operator capability. The results reveal that *The Operator Framework* is the standout framework achieving the profile *Very Good* regarding the overall open source health based on the attributes. *Java Operator SDK* and *Kube-rs* trails behind *The Operator Framework*, achieving the profile *Good*. When considering the open source health status of the project alone, The Operator Framework emerges as the top choice for implementation of an operator.

Attribute	**TOF**	**Shell Operator**	**KUDO**
Com. eng.	[2504, 325, 112, 12, VG]	[140, 33, 15, 6, A]	[698, 62, 12, 2, G]
Eval.	*Very good*	*Acceptable*	*Acceptable*
Maintenance	[24, 21878, 0.959, 66, 106]	[9, 10572, 0.723, 94 , 227]	[0, 0, 0.756, 75, 137]
Eval.	*Very good*	*Good*	*Acceptable*
Popularity	[6500, 1700, 4340, 125]	[2000, 185, 7, 32]	[1100, 101, 18, 24]
Eval.	*Excellent*	*Acceptable*	*Acceptable*
OSS Health	**Very Good**	**Acceptable**	**Acceptable**

Table 6.5: Evaluation of open source health for *The Operator Framework (TOF)*, *Shell Operator* and *KUDO*

Attribute	**Kopf**	**JOSDK**	**Kube-rs**
Com. eng.	[717, 50, 29, 9, A]	[627, 65, 28, 11, VG]	[461, 101, 60, 26, G]
Eval.	*Acceptable*	*Good*	*Good*
Maintenance	[2, 1108, 0.759, 54, 44]	[8, 12036, 0.918, 80, 149]	[19, 12276, 0.835, 94, 98]
Eval.	*Acceptable*	*Very good*	*Good*
Popularity	[1600, 124, 531, 24]	[585, 154, 57, 15]	[2100, 243, 3989, 32]
Eval.	*Acceptable*	*Acceptable*	*Good*
OSS Health	**Acceptable**	**Good**	**Good**

Table 6.6: Evaluation of open source health for *KOPF*, *Java Operator SDK (JOSDK)* and *Kube-rs*

Chapter 7

Discussion

The upcoming chapter will present a discussion of the result gathered in this study. Specifically, the metrics that were selected and the reasoning behind their choice and measurement will be examined. This will culminate in answering the research questions of the thesis. Finally, the limitations of the study will be discussed.

7.1 Abilities and Limitations of the Evaluation Model

This thesis aimed to investigate the existing landscape of Kubernetes operator frameworks and further propose a method for their evaluation. Evaluating software entities is an old practice, however highly topical as views differ on what methodology is best. Relevant for this evaluation is also its chosen application, namely operator frameworks. The subject itself was only recently introduced and is therefore largely unexplored by previous papers. In this thesis, the opted-for evaluation method was chosen on the basis of (a) evaluation, comparison, and decision-making problems, even though metrics and measurements often lay as base, questions of preference and judgment. And (b) evaluation of software entities often uses ordinal scales and measurements, meaning a robust and suited ordinal aggregation method is required. As discovered through the performed literature review a significant amount of metrics relevant for comparison of operator frameworks carry no numerical representation and as so is not applicable on an interval scale, but rather an ordinal one. The profile-based evaluation model proposed in this paper fairs well with a combination of interval and ordinal metrics, indicating its suitability for the issue at hand. The proposed aggregation method, inspired by multi-criteria filtering using concordance and non-discordance was also observed as fitting given its attributed qualities. By negotiating the majority strength's vote with the right of a single metric veto we observe some auspicious behaviors of the system such as penalizing alternatives presenting good but irregular profiles, in favour of good and well-balanced alternatives.

The application of a profile-based evaluation model has shown definition of suitable attributes and metrics to be difficult tasks. Several metrics are available and deemed standard practice for evaluating generic software entities, however, prove of little relevance for this thesis, investigating operator frameworks. Such metrics are typically heavily focused on code quality, code complexity, and code efficiency all of which largely prove superfluous to developers studying frameworks as their interaction with previously written code is very limited, instead focusing on their own independent and largely contained code. There were metrics rather obvious to the model which warranted little discussion and further analysis. Such would be metrics measuring non-compensatory characteristics like *maturity level reachable* as it obviously and directly affects the feasibility of use for the framework. Others were instead more exploratory in nature, resembling that of a predictive model. For instance, metrics pertaining to the popularity of a repository project carry no intrinsic value to the feasibility of use but rather serve as node or factor for a prediction of quality. These metrics benefit from being easily interpretable and measurable, but the difficulty lies instead in validating their contribution.

Profile assignment, as presented in Chapter 5, was done using lower limits for qualification where the limits were either data-driven or expertise-driven. Both procedures carry varying degrees of accuracy, subjectivity, and potential bias. The model would prefer, as is also argued by Morisio et. al, limits be set prior to measurement and data-gathering. This would emphasize what experience the model designer possesses, clearly circumventing any bias to the gathered results. However, for selected metrics, it was deemed unfeasible to, prior to data-gathering, formulate limits. As so instead the data-driven process was employed which used pre-defined percentiles, applied to the range of values found. Both empirical and intuitive indications hint at this being an at least partly flawed approach. As observed in the results we encounter the issue of always assigning the lowest value found *Acceptable*, likewise always assigning the greatest value found *Excellent*. These are likely erroneous assignments as they rely on only the six gathered data points, lacking other expertise or motivated preference. Likewise, the percentiles are estimates based on available data and assume no statistical distribution, otherwise a common point of interest when determining percentile limits.

7.2 Initial Selection Pool of Frameworks

The goal of the first selection was to get a wide overview of the existing operator landscape, with a focus on frameworks that were most relevant and commonly used within the Kubernetes ecosystem. To narrow down the selection for a more in-depth comparison, we applied certain filters to identify the top six remaining frameworks. Despite Kubebuilder being the second most popular, trailing only slightly behind the Operator Framework in terms of GitHub statistics, we decided to not analyze it further. The reasoning was that it was deemed very similar to The Operator Framework, with The Operator Framework even utilizing Kubebuilder internally. The framework with the lowest popularity which was included was the Java Operator SDK, and all frameworks with lower popularity got filtered out. The remaining pool of frameworks was considered to cover a wide range of functionality and being representative of the current operator framework ecosystem.

7.3 Profiles and Metrics

The attributes were decided to be divided into multiple metrics in order to receive a more fair profile ranking. For instance, a high percentage of code comments does not automatically translate to having good documentation. But paired with the overall amount of documentation and certain criteria were considered to give a more accurate representation. Considering the frequent updates in Kubernetes, the documentation criteria included that it should have been updated within a year. KUDO was the only framework that failed that criterion, with its documentation remaining unchanged for two years. Additionally, the framework as a whole had not been updated in the recent six months, as indicated by the maintenance attribute. These findings lead us to conclude that KUDO is no longer actively maintained. Frameworks that it is not up to date with Kubernetes could make it potentially inaccurate and a less secure option. Which is why developers should take this into consideration before choosing KUDO as framework for their operator development.

One potential problem with the evaluation model is that all metrics that are data-driven are equally weighted using a percentage of the interval. One could argue that certain metrics are more important than others. For example, the result revealed that The Operator Framework, KUDO, and Kopf as the highest profiling for *Operator capability*, all with an overall score of *Very good.* However, the operator framework only had a *Good* profile for functional scope due to the number of dependencies, while Kopf received *Excellent.* This metric heavily favours smaller projects with a smaller code base, even though having more dependencies does not necessarily indicate a weaker project. Rather, it implies that it might be more difficult to maintain and can lead to compatibility issues and security vulnerabilities. Thus, one possible approach would be to assign a lesser weight to this metric.

In *Open source project health*, the only framework which received the profile *Very good* was The Operator Framework. This most likely being the result of it being the longest-lasting framework of the selection. As well as being featured in a multitude of books and media, such as for example [18] and [17]. The significantly higher overall popularity and engagement of The Operator Framework created high thresholds, which made it challenging for other frameworks to gain a profile better than *Acceptable.* This is due to using a data-driven approach giving us somewhat skewed results.

Besides using mostly using data-driven for *Community engagement*, the metric *Community platforms* was an expertise-driven metric that was assessed based on the offered platforms accessible for the community. The standout frameworks here were The Operator Framework but also the Java Operator SDK. Which is a bit surprising considering the size difference between the two frameworks. In contrast to many other frameworks in the comparison, Java Operator SDK has an active discord, has weekly meetings for its community as well as being part of CNCF, as of the 18th 2023.

In the comparison, GitHub discussions served as the primary source for retrieving data within the *community engagement* attribute. However, for The Operator Framework and KUDO, where GitHub discussions were disabled on their repositories, Stack Overflow was utilized instead. Initially, our intention was to only include posts from Stack Overflow for all frameworks. But ultimately GitHub was chosen as the main source since Stack Overflow yielded unfair results. For instance, a search query like *Shell Operator* would provide results completely unrelated to

Kubernetes Operators, while a *Java Operator SDK* would give very specific results. But it is important to acknowledge that comparing two separate platforms can introduce some bias to the model.

Overall, the model provides a way to determine a framework that best aligns with the developing parties' requirements, from a perspective of both open source and capability. The model relies on domain knowledge, which introduces subjectivity into the process. Ultimately it is up to the developers to interpret the result, based on preference and need. A senior developer might prioritize familiarity with programming languages and the frameworks's capability, rather than focusing on documentation or community engagement. On the other hand, a developer with less experience with operators might opt to choose a framework based on its documentation, tutorials, and community support.

7.4 Answers to Research questions

What Kubernetes operator frameworks are most widely used today for managing the lifecycle of containerized applications?

There are numerous alternatives of operator frameworks available to develop an operator for handling an application's lifecycle in a Kubernetes cluster. Through the study performed in this thesis, the most popular and commonly used were evaluated and compared. The frameworks compared were The Operator Framework, KUDO, Kopf, Shell Operator, Java Operator SDK, and Kube-rs. Worth noting that the area is relatively new and there exists alternatives as well as new frameworks being created.

What technical metrics should be used to aid in operator framework-related decision making, based on the task presented in section 1.1?

Attributes and metrics were defined through an extensive literature study and the gathered domain expertise available to us through working professionals. Based on our findings, the selection of metrics should consider a combination of metrics related to operator-specific capabilities and the overall health of the open-source project. By assessing both *Operator Capability*, and *Open Source Health*, developers can identify a framework that both meets their required capabilities, as well as a robust and healthy community. The specific attributes and accompanying metrics are presented and motivated in Chapter 5.

How can the operator frameworks (see Research Question 1) be classified and categorized based on the task presented in section 1.1?

The proposed answer to this question is two-fold and dependent on at what depth a party is interested in evaluation frameworks. The process presented by this thesis defined firstly a novel evaluation model with overarching categories and suitable attributes. Secondly, it presented ordinal profiles in which varying framework qualities were assigned using metric aggregation. Categorization for the purpose of decision-making may hence be performed either using attribute profiles or the more compiled, holistic grade. A proposed method for the former is defining strict, binary requirements on selected attributes most relevant for their situation/environment/technical specifications. Such would generate a sort of mapping procedure, marking frameworks either acceptable or not so, pending the decision maker's preference applied on top of the evaluation model. The latter simply being classifying frameworks on their final, compiled grade.

7.5 Limitations

In this report, we decided to narrow down the comparison to six different frameworks in order to get a more concise comparison. Although, it is important to acknowledge that there are other frameworks and approaches available for implementing an operator that was not covered in this study.

The selection and gathering of metrics for this study also introduce certain limitations. Our aim was to obtain a combination of open source-related metrics as well as operator-specific metrics. The open source-related metrics were relatively straightforward to choose and were supported by previous research. Conversely, selecting operator-specific metrics presented more challenges. Kubernetes operators are complex, and determining which qualities are relevant requires extensive knowledge and expertise. The attributes and metrics in this thesis were a result of a combination of what we learned was important during our research and want developers looked for when searching for an operator framework.

Several Python scripts were developed to collect certain metrics, such as word count and comment percentage. These scripts may contain bugs that can impact the accuracy of the results. Another issue with the metric collection was the ones that required interpretation to determine. One example is the maturity of the operator, which is typically determined after implementation. However, implementing an operator with the ambition to reach the highest maturity is not feasible within the timeframe of a master thesis.

As mentioned briefly in the discussion, there are certain aspects of the model that may be flawed. For instance, in the model, all metrics have equal importance. One approach would be to assign weights to metrics considered less important. Currently, if one metric within an attribute is a negative outlier it significantly impacts the overall profile of the attribute. Another flaw is the way the threshold was determined based on thresholds of the interval between the lowest and highest data points. This caused the worst framework to always get an acceptable ranking, regardless of the result. Another appropriate approach would be to determine limits beforehand which would cause certain data points to fall under *Unacceptable* profile.

Chapter 8

Conclusion

The Conclusions section serves as the culmination of our study and aims to offer a summary of the key insights and outcomes derived from our analysis and evaluation. It will also feature our thoughts on potential future work to be done on the subject.

8.1 Conclusion

This thesis presented an evaluation model for operator frameworks produced through a literature review and framework study. The evaluation model is an implementation of a process proposed by Morisio et al. and stems from a technique of defining and conducting profile-based evaluations of software entities. The research objectives of this study were meant to investigate the current scene of operator frameworks and their surrounding technical environments. As well as assess what attributes and metrics may be utilized in order to effectively evaluate and compare such frameworks. Cloud services, container technology, and Kubernetes are all technologies quickly rising in popularity due to the growing demand for their services. Similarly, the market for Kubernetes operators is also growing, making their frameworks a highly engaging topic for many, especially obvious in the open source community. A vast number of frameworks are currently available, lead in popularity by frameworks such as *Operator Framework*, *Kopf*, *KUDO*, *Shell Operator*, *Java Operator SDK*, and *Kube-rs*. However, popularity only represents a single aspect of evaluation. Through this study we identified a range of attributes suitable for evaluation, all falling under two overarching categories being *operator capability* and *open source project health.* Further investigation showed the approach of defining a profile-based evaluation model to be a suitable option for framework evaluation, however requiring substantial effort and extensive domain knowledge. The process proposed by Morisio et al. was deemed utilizable and effective in the context but should be viewed and considered preference-based as strong empirical evidence is largely missing, due to the difficulties of verifying its result.

8.2 Future Work

This thesis applied established practices to a modern and emerging subject, being Kubernetes operators and their frameworks. While the practices themselves are largely explored and well studied the subject and application of it are not, meaning much is yet to discover. An obvious branch of future work is the inclusion of additional attributes and or metrics to extend the evaluation model. The attributes chosen for this experiment were found using the GQM-technique, applied with the support of a literature study. However, there are suitable alternatives for establishing a model, with the most prominent being the inclusion of discussions, questionnaires, or interviews with stakeholders. Such options would thrive especially if participants represented a wide range experiences, knowledge, and self-interests. Another approach would be implementing a basic template for all chosen frameworks in order to get a better grasp of their differences. Preferably this would involve a more complex stateful application. This would reveal multiple distinguishing attributes of interest for different frameworks. Such as the difficulty of achieving a higher level of capability, how time-consuming the different implementations are and what previous knowledge is required to efficiently utilize the framework.

The experiment carried out for this thesis was also largely exploratory in nature, and would see great benefits in being supplemented by an empirical study. Such a study could evaluate the results against more theoretically accepted and established knowledge in order to validate its results. A feature-based comparison, such as this thesis was, provides a rather structured approach to evaluating software entities based on specific capabilities and ensuring alignment with project requirements. However, to gain deeper insights into the frameworks' performance and efficiency, performance benchmarking seems a suitable option for future work. By measuring key metrics and assessing real-world performance, performance benchmarking allows for a more comprehensive evaluation of the frameworks' abilities and their suitability for potential scenarios.

References

[1] Leila Abdollahi Vayghan et al. “A Kubernetes Controller for Managing the Availability of Elastic Microservice Based Stateful Applications”. In: *arXiv e-prints* (2020), arXiv–2012.

[2] Claudio Agostino Ardagna, Ernesto Damiani, and Fulvio Frati. “Focse: an owa-based evaluation framework for os adoption in critical environments”. In: *Open Source Development, Adoption and Innovation: IFIP Working Group 2.13 on Open Source Software, June 11–14, 2007, Limerick, Ireland 3*. Springer. 2007, pp. 3–16.

[3] Filip Bolling and Jonna Gustafson. “Exploring Business Value and User Experience of Open Source Health”. In: *Faculty of Engineering, Lund University, Sweden* (2021).

[4] Antonio Bucchiarone et al. “From monolithic to microservices: An experience report from the banking domain”. In: *Ieee Software* 35.3 (2018), pp. 50–55.

[5] Steve Buehler. *dotnet-operator-sdk GitHub repository*. 2021. URL: `https://buehler.github.io/dotnet-operator-sdk/`.

[6] Brendan Burns et al. *Kubernetes: up and running*. " O’Reilly Media, Inc.", 2022.

[7] Victor R Basili1 Gianluigi Caldiera and H Dieter Rombach. “The goal question metric approach”. In: *Encyclopedia of software engineering* (1994), pp. 528–532.

[8] Carmen Carrión. “Kubernetes as a Standard Container Orchestrator-A Bibliometric Analysis”. In: *Journal of Grid Computing* 20.4 (2022), p. 42.

[9] CNCF. *Who we are*. Accessed on April 19, 2023. URL: `https://www.cncf.io/about/who-we-are/`.

[10] *Containers*. `https://www.ibm.com/topics/containers`. Accessed on April 125, 2023.

[11] Java Operator Contributors. *Java Operator Docs.* Accessed on April 19, 2023. 2021. URL: `https://flant.github.io/shell-operator/`.

[12] Java Operator SDK Contributors. *Java Operator SDK GitHub Repository.* Accessed on April 19, 2023. 2021. URL: `https://github.com/java-operator-sdk/java-operator-sdk`.

[13] Juju Contributors. *Juju Repository.* Accessed on April 19, 2023. 2021. URL: `https://github.com/juju/juju`.

[14] Kube-rs Contributors. *Kube-rs Repository.* Accessed on April 19, 2023. 2021. URL: `https://github.com/kube-rs/kube`.

[15] KUDO Contributors. *KUDO GitHub repository.* 2021. URL: `https://github.com/kudobuilder/kudo`.

[16] Quarkiverse contributors. *Quarkiverse Java Operator SDK.* Accessed on April 20, 2023. 2021. URL: `https://quarkiverse.github.io/quarkiverse-docs/quarkus-operator-sdk/dev/index.html`.

[17] Michael Dame. *The Kubernetes Operator Framework Book: Overcome complex Kubernetes cluster management challenges with automation toolkits.* Packt Publishing Ltd, 2022.

[18] Jason Dobies and Joshua Wood. *Kubernetes operators: Automating the container orchestration platform.* O'Reilly Media, 2020.

[19] Ruxiao Duan, Fan Zhang, and Samee U Khan. "A Case Study on Five Maturity Levels of A Kubernetes Operator". In: *2021 IEEE Cloud Summit (Cloud Summit).* IEEE. 2021, pp. 1–6.

[20] Flant. *Go? Bash! Meet the shell-operator.* Accessed on April 19, 2023. 2020. URL: `https://medium.com/flant-com/meet-the-shell-operator-kubecon-36c14ba2f8fe`.

[21] Dennis Gannon, Roger Barga, and Neel Sundaresan. "Cloud-native applications". In: *IEEE Cloud Computing* 4.5 (2017), pp. 16–21.

[22] Shazibul Islam Shamim et al. "Benefits, Challenges, and Research Topics: A Multi-vocal Literature Review of Kubernetes". In: *arXiv e-prints* (2022), arXiv–2211.

[23] Kubernetes. *Operator pattern.* 2023. URL: `https://kubernetes.io/docs/concepts/extend-kubernetes/operator/` (visited on 05/01/2023).

[24] Valentina Lenarduzzi et al. "Open source software evaluation, selection, and adoption: a systematic literature review". In: *2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA).* IEEE. 2020, pp. 437–444.

[25] Marko Luksa. *Kubernetes in action.* Simon and Schuster, 2017.

[26] Anshita Malviya and Rajendra Kumar Dwivedi. “A Comparative Analysis of Container Orchestration Tools in Cloud Computing”. In: *2022 9th International Conference on Computing for Sustainable Global Development (INDIACom).* IEEE. 2022, pp. 698–703.

[27] Philippe Martin et al. *CNCF Operator WhitePaper.* Tech. rep. Cloud Native Computing Foundation, 2022.

[28] Meta. *WhatsApp Onpremises Platform.* 2023. URL: `https://developers.facebook.com/docs/whatsapp/on-premises/overview` (visited on 05/01/2023).

[29] Marek Moravcik and Martin Kontsek. “Overview of Docker container orchestration tools”. In: *2020 18th International Conference on Emerging eLearning Technologies and Applications (ICETA).* IEEE. 2020, pp. 475–480.

[30] Marek Moravcik et al. “Kubernetes-evolution of virtualization”. In: *2022 20th International Conference on Emerging eLearning Technologies and Applications (ICETA).* IEEE. 2022, pp. 454–459.

[31] Maurizio Morisio, Ioannis Stamelos, and Alexis Tsoukias. “Software product and process assessment through profile-based evaluation”. In: *International Journal of Software Engineering and Knowledge Engineering* 13.05 ().

[32] Javad Nasserifar. “Open Source Software Ecosystem: A Systematic Literature Review”. In: (2016).

[33] *Operator SDK Documentation.* `https://sdk.operatorframework.io/docs/`. Accessed: 10 may 2023. 2020.

[34] Jun Xiang Tee Palek Bhatia. *Best practices for building Kubernetes Operators and stateful apps.* `https://cloud.google.com/blog/products/containers-kubernetes/best-practices-for-building-kubernetes-operators-and-stateful-apps`. 2018.

[35] Etiel Petrinja, Ranga Nambakam, and Alberto Sillitti. “Introducing the opensource maturity model”. In: *2009 ICSE Workshop on Emerging Trends in Free/Libre/Open Source Software Research and Development.* IEEE. 2009, pp. 37–41.

[36] Chris Richardson. *Enabling rapid, frequent and reliable software delivery.* 2023. URL: `https://microservices.io` (visited on 05/01/2023).

[37] Joachim Schaper. “Cloud Services”. In: (2010). DOI: `10.1109/DEST.2010.5610668`.

[38] Kubernetes SIGs. *Kubebuilder GitHub repository.* 2021. URL: `https://github.com/kubernetes-sigs/kubebuilder`.

[39] Alen Šimec, Bruno Držanić, and Davor Lozić. "Isolated Environment Tools for Software Development". In: *2018 International Conference on Applied Mathematics Computer Science (ICAMCS).* 2018. DOI: `10.1109/ICAMCS46079.2018.00016`.

[40] Nikita Sobolev. *Kopf GitHub repository.* Accessed on April 19, 2023. 2021. URL: `https://github.com/nolar/kopf`.

[41] Diomidis Spinellis et al. "Evaluating the quality of open source software". In: *Electronic Notes in Theoretical Computer Science* 233 ().

[42] Statista. *Leading containerization technologies market share worldwide in 2022.* Accessed on May 10, 2023. 2022. URL: `https://www.statista.com/statistics/1256245/containerization-technologies-software-market-share/`.

[43] Jayachander Surbiryala and Chunming Rong. "Cloud Computing: History and Overview". In: *2019 IEEE Cloud Summit.* 2019, pp. 1–7. DOI: `10.1109/CloudSummit47114.2019.00007`.

[44] Heyong Wang, Wu He, and Feng-Kwei Wang. "Enterprise cloud service architectures". In: *Information Technology and Management* 13 (2012).

[45] *What is Kubernetes?* `https://kubernetes.io/docs/concepts/overview/`. Accessed on April 19, 2023.

[46] *What is open source?* `https://www.redhat.com/en/topics/open-source/what-is-open-source`. Oct. 2019.

[47] Bibin Wilson. *Container Orchestration Tools and Services.* 2022. URL: `https://devopscube.com/docker-container-clustering-tools/` (visited on 02/01/2023).

[48] Yuhang Zhao et al. "Evaluation indicators for open-source software: a review". In: *Cybersecurity* 4.1 (2021), pp. 1–24.

[49] Chen Zhi et al. "Quality Assessment for Large-Scale Industrial Software Systems: Experience Report at Alibaba". In: *2019 26th Asia-Pacific Software Engineering Conference (APSEC).* 2019, pp. 142–149. DOI: `10.1109/APSEC48747.2019.00028`.