Tampere University

Klaus Telenius

SIMPLIFIED PRODUCT CONFIGURATION THROUGH EXTENDED KUBERNETES

Master's Thesis
Faculty of Engineering and Natural Sciences
Examiners: University Teacher Mikko Salmenperä, Assistant Professor David Hastbacka
February 2023

ABSTRACT

Klaus Telenius: Simplified product configuration through extended Kubernetes
Master's Thesis
Tampere University
Automaation Tietotekniikka
February 2023

This paper describes an architecture for decoupling the development process from product implementation for a service in cloud domain. By using automation, a high enough level of abstraction can be achieved in order to effectively and easily create and deploy new instances of an application for customers without involving the developers of the underlying software. The goal of this is to free resources for enhancing the underlying code base, and to enable horizontal development for bringing in new customers. The architecture is implemented on the Kubernetes container management platform and leverages its design concepts.

Keywords: Cloud, Kubernetes, Abstraction, Software Development, Automation

TIIVISTELMÄ

Klaus Telenius: Tuotekonfiguraatioiden hallinta mikropalveluarkkitehtuurissa
Diplomityö
Tampereen yliopisto
Information Systems in Automation
Helmikuu 2023

Tämä diplomityö esittelee pilvipalveluarkkitehtuurin, jonka tarkoituksena on erottaa palvelun ohjelmistokehitysprosessi siitä tuotettujen tuotekonfiguraatioiden kehittämisprosessista. Automaation avulla voidaan saavuttaa abstraktiotaso, joka mahdollistaa ohjelmiston instanssien luomisen ja jalkauttamisen ilman ohjelmiston alkuperäisten kehittäjien panosta. Arkkitehtuurin tavoite on vapauttaa resursseja alla olevan koodikirjaston kehittämiseen ja mahdollistaa horisontaalinen skaalautuminen uusien asiakasinstanssien luomiseen. Arkkitehtuuri on implementoitu pilvessä toimivien applikaatioiden hallintaan kehitetyn pilvipalvelualusta Kuberneteksen päälle ja on kehitetty käyttäen alustalla kehitettyjä suunnittelumalleja.

Avainsanat: Pilvipalvelu, Kubernetes, Abstrahointi, Ohjelmistokehitys, Automaatio

PREFACE

Thanks to Ville Haukkala for the initial problem, his vision and his everlasting positive mindset. Thanks to David Hästbacka and Mikko Salmenperä for their work on guiding me to make this an actual thesis instead of a project description. Thanks to Matti Taivaljärvi and Juha Katajamäki for ideas and critique. Thanks to Jyrki Berg and Nina Sainio for pushing me when I lacked motivation. And most importantly thanks to my wife, family and friends for supporting me.

Tampere, 10th February 2023

Klaus Telenius

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF SYMBOLS AND ABBREVIATIONS

API	Application Programming Interface, An interface for controlling an application via requests
CI	Continuous Integration, a software development paradigm where committed code is instantly and automatically tested before integrating to the product
CRD	Custom Resource Definition, a way to extend controllable resources on the Kubernetes platform
etcd	An open-source distributed key-value storage
GUI	Graphical User Interface, a tool for allowing the user to communicate with a program by graphical elements
k8s	short for Kubernetes, a container orchestration platform for cloud domain
MB	Megabyte, a unit of measure for space on a computer
SOA	Service-Oriented Architecture, a software architecture where components are used as services by each other
XaaS	Everything-as-a-Service, all layers of the cloud stack provided as services
XML	Extensible Markup Language, a document encoding for human- and machine readability

1 INTRODUCTION

Imagine a world class violin player. Any notes you put in front of her she can play flawlessly and beautifully, but she cannot compose. The record company is known for its contemporary reputation and Bach just won't cut it anymore, so they hire an army of composers to help her provide new experiences to the audience. She gets to play, composers get to compose, and the record company makes money.

In the world of data, the violin player would be a data streaming framework and its developers. This paper is about enabling horizontal scaling in developing of data streaming applications for a large international telecom corporation. In other words to facilitate the role of a composer: a domain expert.

With an increasing customer base, developing and administering new customer use cases becomes a time consuming task. This promotes new business requirements for development efficiency. The workforce should concentrate on their specific field of work in order to achieve maximum productivity. In an evolving data ecosystem, the tools used must provide a way to distinguish responsibilities between software developers and domain experts.

The organization that ordered this thesis has domain experts and software developers. Domain experts are *not* expected to possess deep knowledge of software, nor are software developers expected to know everything about the field of telecommunications. However, to construct a sound data pipeline for the domain, knowledge of both is needed. These stakeholders must work together in order to produce data-pipelines that bring value to customers. Currently domain experts would produce a specification of the data flow in the pipeline. No requirements exist for the formatting of the specification and the software developers are left with interpreting the specification in order to build the pipeline. The specification evolves as the projects evolve and introduce change that is not managed.

The software ecosystem at hand consists of multiple different projects. A generic base project exists. It contains sub-projects for different kinds of data pipelines, their application code and deployment. Also the codes for a different service utilizing some parts of the same framework also reside in this project, making it an enormous code base with multiple developers from multiple different countries in multiple timelines. Most of the code is made by developers who have since been re-assigned to different duties. The metrics and analysis are reporting increasing technical debt. While the developing team is adept to fix all this, pressuring timelines force the team to develop new pipelines to

customers and thus, the debt keeps increasing. A solution was proposed to decouple the development processes for the code base and the actual products *by software*.

The decoupling of the development processes requires a proper interface. A domain expert should be able to create and modify data processing workflows without low level programming skills. To tackle this problem the following concepts are introduced as prerequisites. The business domain of this thesis is the Cloud concept defined in chapter 2. The implementation domain is Kubernetes, which is an application management system designed to fit the needs of the cloud domain, and on which the applications are run. To consider a data processing workflow created or modified, it is expected to be able to be delivered to a production system without a software developer. Thesis will explore architectural patterns and paradigms of software development in the cloud. Thesis investigates existing resources and implementations that Kubernetes and its design choices provide and enable.

The goal for this thesis is divided into three research questions. It starts out by resolving the question: *What requirements are there for the domain experts to configure the data processing workflows*? Answering this question will require defining the stakeholders of the development process. The business requirements will ultimately drive the functional and non-functional requirements of the system.

After these requirements have been clarified the thesis aspires to further clarify the details of the resulting construct. The goal of the abstraction is to make the creation, configuration and productification of a sound data pipeline easier for a domain expert. With the requirements defined the next question is brought forward: *In Kubernetes domain, what design choices do we have to implement such an abstraction*? The thesis proceeds by examining the tools in our disposal and what prerequisites such tools impose on the architecture.

After the type and characteristics of interface are clear the final implementation details of the constructs are considered. This prompts the third research question: *In Kubernetes domain, what tools do we have to implement an interface that fulfills the requirements*? This includes building the implementing constructs based on previous design constructs and evaluating them against the requirements.

The research method for this thesis is the design-oriented constructive research method by Järvinen P. and Järvinen A. [1] pictured in figure 1.1. It's goal is to solve real world problems by developing a set of constructs from which a prototype or an artefact is developed. Constructs can be handled as abstract concepts that have a limitless amount of possible implementations. Constructs can be defined to mean any and all human-made artefacts such as diagrams, designs, models, organizational constructs, commercial products, or data structures. In this thesis the constructs revolve around a specific domain which is the Kubernetes platform and thus have a well defined set of implementations available.

First the current state of the project is identified. From the project a requirement is defined

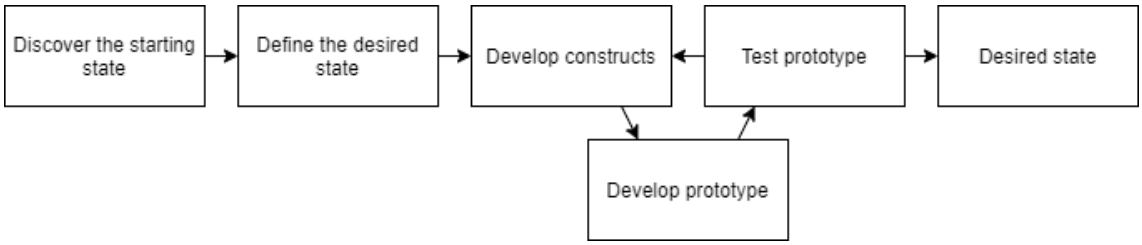

Figure 1.1. *Design-oriented constructive research [1]*

for a decoupled development process with a proper abstraction. This is the desired state. From this demand a set of requirements are derived which take the state of the project towards the desired one. These requirements prompt constructs that will be used to implement a prototype. The prototype is then tested and by iterating, new requirements are identified and implemented until the desired state of the project and the desired quality of the prototype is reached. [1]

This thesis focuses on the first three stages of the design-oriented constructive research. The research questions aspire to reach the state of a developed construct, from which building and testing a prototype can be started in order to reach the desired state. The document is structured as follows. Chapter 2 discusses the general domain in which our application operates. Chapter 3 takes a look at the tools utilized in a defined cloud domain, Kubernetes. Chapter 4 defines the requirements for our configuration model. Chapter 5 presents the proposed construct. Chapter 6 evaluates the results and how well the research questions were answered. Thesis closes by conclusions in chapter 7.

2 CLOUD COMPUTING AND SERVICE ORIENTATION

Cloud in its modern context was coined around the year 2006 [2]. David C. Wyld describes the metaphor of cloud in the world of computing to have been of almost mystical in nature, describing the internet in diagrams and mapping operations of the networked environments. Essentially the emerging of cloud computing represented a rudimentary evolution in the location of computing resources and the economics of computing. Services are delivered over the Internet from a remote location when the actor needs it as opposed to the resources residing on the devices in the actor's physical environment.

More profoundly anything from email communication to batch processing large data sets can be moved away from individual PC's or corporate data centers. It also abstracts away all the underlying layers of the technology stack leaving the user with only the computing power at their use without the need to manage any of the complexity or technology that is included in providing the power. [3]

Since then the field has evolved to a collection of principles and well-established models for developing, providing and using cloud based applications. One of the dominant taxonomies is the everything as a service (XaaS) service provisioning model. It describes an outline for Service-Oriented architecture (SOA) and design to support the development and deployment of software applications as individual services. [4]

A service is an isolated, compact and logical representation of a composable business or technical task with a certain outcome. Services as modular units of functionality can be arranged into multiple different systems. An example of such composing is a mechanism called layering [5] Figure 2.2 describes the different 'aaS' concepts in a software as a service business model and how they map into different layers of the cloud stack. In XaaS, the resources, including infrastructure, storage, networking and computing systems, can be abstracted as SOA-based services through layering and other common mechanisms such as modularity and loose coupling. The result is that, network and cloud services may be orchestrated to create compounded services that are delivered to end users as entire systems. [4]

Software as a service is a business model for delivering a software service to customers. The goal of software as a service is to minimize the cost of managing software owned by different customers. Software as a service introduces a single code base. The

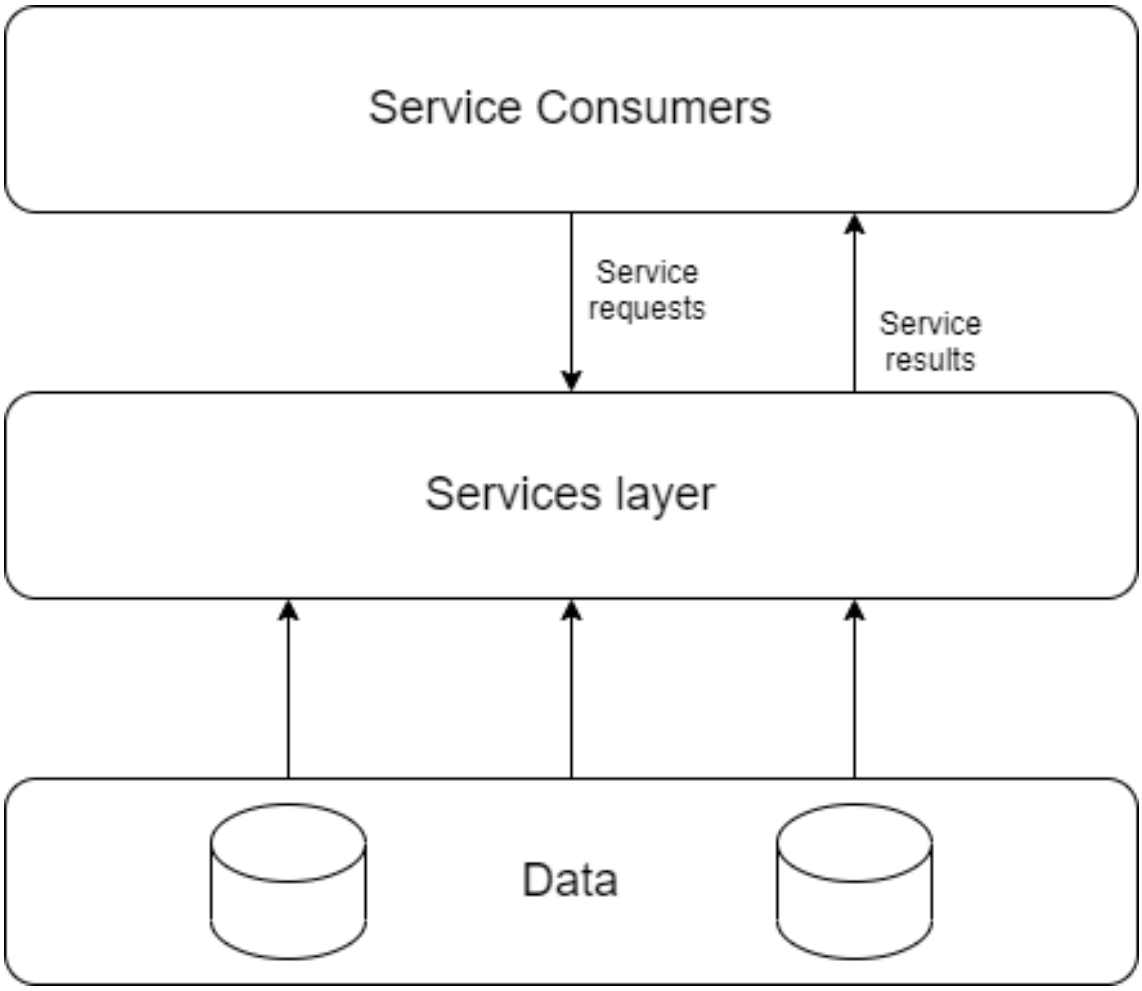

Figure 2.1. *Service oriented architecture [5]*

built software is run on cloud infrastructure and accessed by clients through various client devices. The customers only own their configurations, or, the interpretation of the application and it's data. This eliminates the task of handling different software versions on different clients. Upgrades done to the code base are provided to each customer simultaneously. [7]

The constructs in this thesis adapt the ideology behind the SaaS (software as a service) model in figure 2.3. In this thesis the principles are applied to in-house development process, where 'customers' are experts within the organisation producing customers different solutions. While derived from the same code-base, each customer gets the software delivered to their on-premise cloud, only with different configurations.

2.1 Designing Software for the Cloud

When software harnesses the advantages of the cloud computing model, it is designed in a "cloud-native" way. In their definition The Cloud Native Computing Foundation restricts cloud-native development to only use open source software stack that is microservices-oriented, dynamically orchestrated and containerized. Cloud-native implies that the

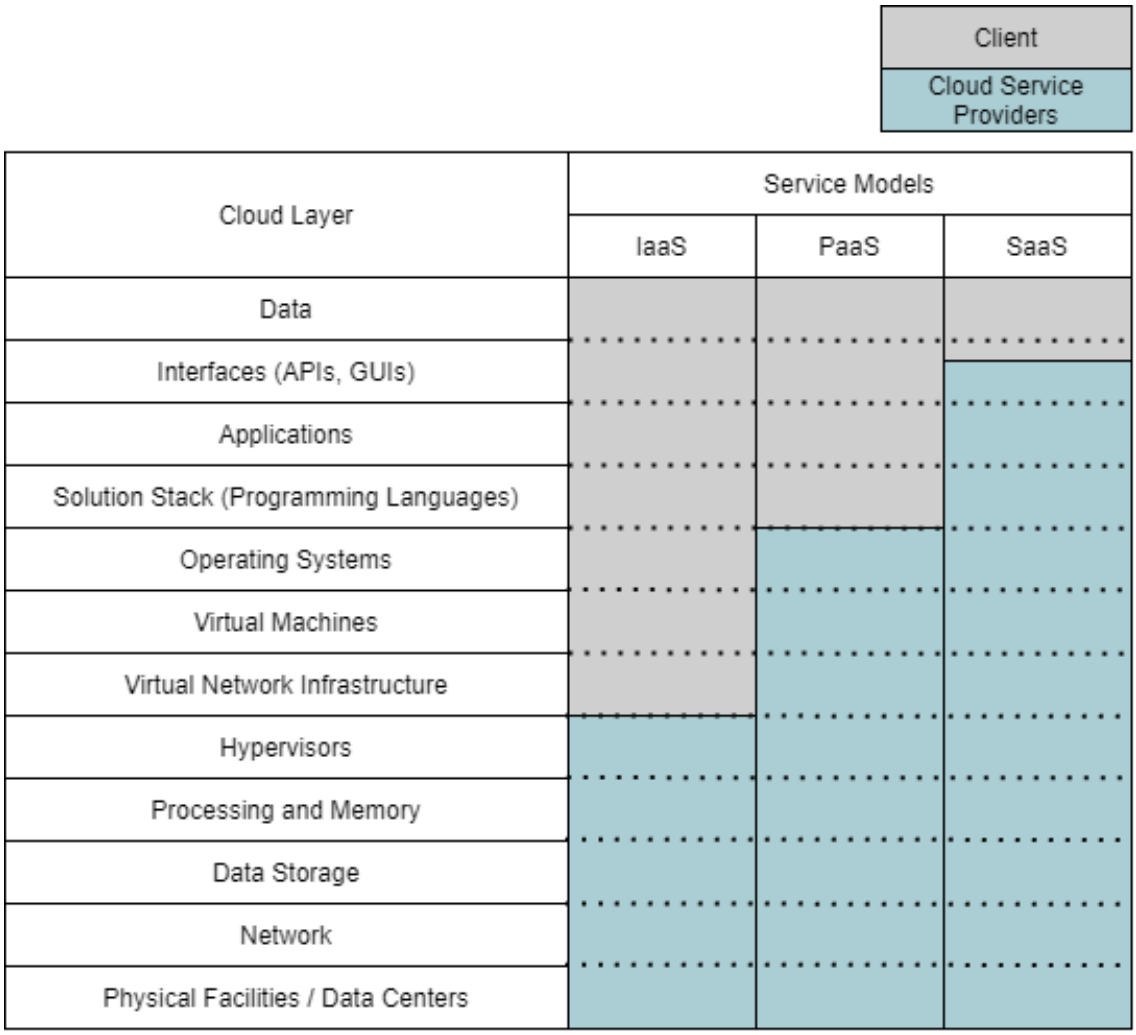

Figure 2.2. *Infrastructure, platform and software as a service responsibilities [6]*

applications reside in a public cloud, in contrast to them living in an on-premises data center. [8]

Pahl et. al [5] demonstrate a distinct software architectural style regarding continuous service systems operation and development. They introduce a group of principles and patterns for designing model-based, control-theoretic architectures for the cloud. This is to address the challenges of a multi-tiered, highly fragmented and distributed environment architecture. The style takes into account the full provisioning stack all the way from application to platform and resources, how they are operated, and how they are managed dynamically. To support this thesis, a few key characteristics for the cloud are defined from this architectural style.

Service orientation follows principles of modularity, layering, and loose coupling. *Virtualization* requires providing services for portable application containers and shared, virtualized resources. *Adaptivity and Elasticity* demand support for dynamic adaptation and variability management. *Uncertainty* is a fundamental factor in cloud environments caused by heterogenity, multi-user involvement, changing contexts and distribution. [5]

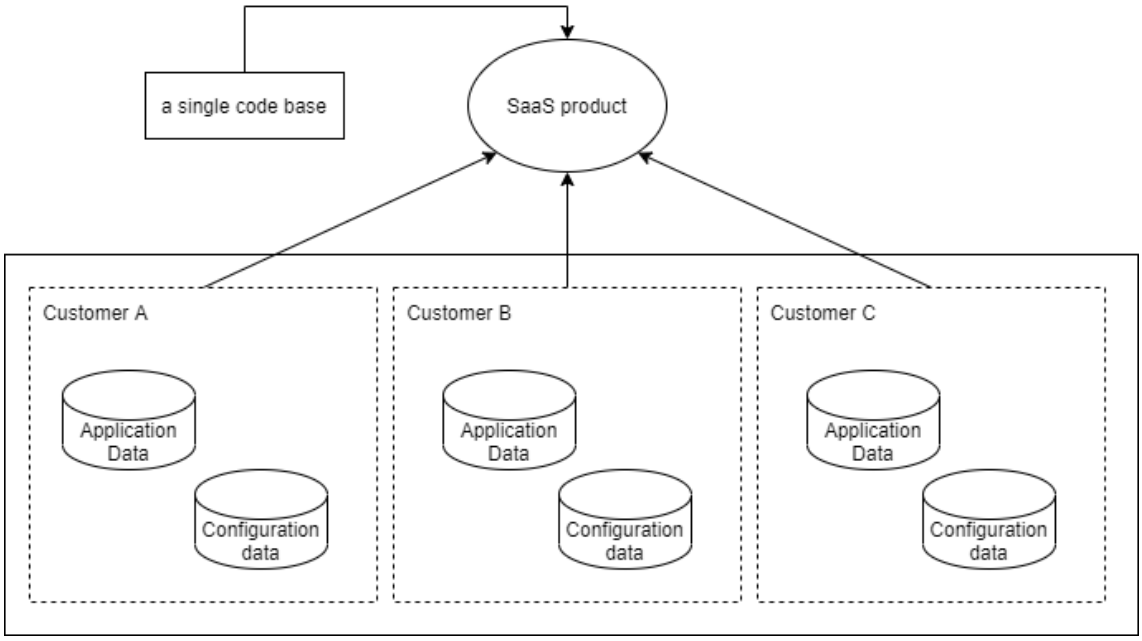

Figure 2.3. *SaaS model [7]*

These characteristics are met with certain patterns such as microservices, dynamic models and control at runtime with controller-based feedback loops, DevOps and continuous development with the goal to manage quality. [5]

2.1.1 Virtualization

Virtualization makes it possible to elastically manage and provide resources as services to the applications. Pahl et. al [5] distinguish three forms of virtualization, depicted in 2.4.

Infrastructure virtualization divides physical infrastructures, creating committed resources of virtual nature. This enables multiple operating systems and applications to be run concurrently on the same server.

Platform virtualization abstracts the technology stack (platform) so the virtualized applications can be managed flexibly. This can be achieved through containerization, which means using portable light weight application containers, who build on infrastructure virtualization techniques for process isolation, assembled on top of platform components.

Application virtualization targets to separate the underlying platform and infrastructure from the applications. This way they may run concurrently while shifting through platforms.

2.1.2 Adaptivity and Elasticity

Pahl et. al [5] used a set of model dimensions aligned with the cloud to contextualize *adaptation*. In the cloud system objectives evolve and can be expressed more flexibly,

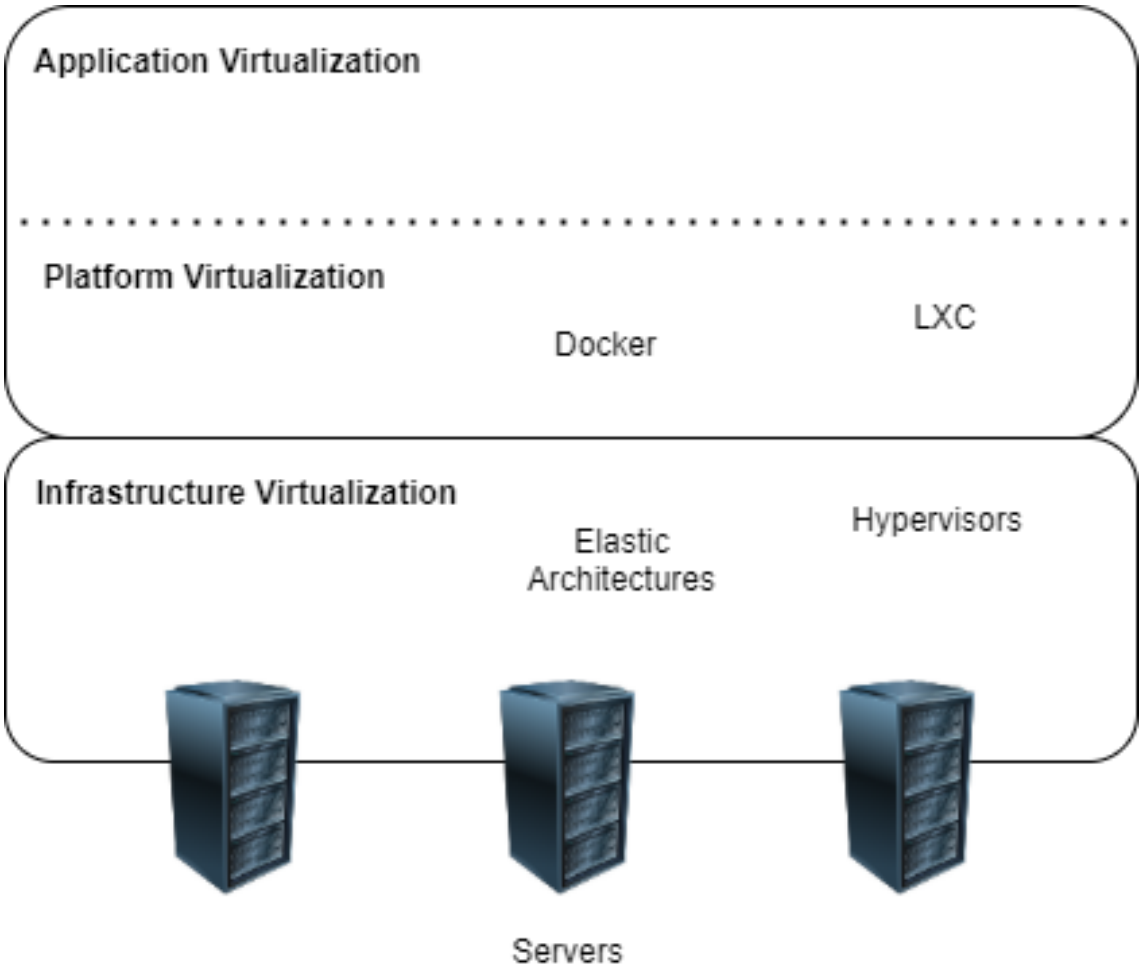

Figure 2.4. *Forms of virtualization [5]*

which result in adaptation to contain a set of goals. By monitoring the applications and resources *change* can be captured as a cause for adaptation. Cause has the parameters source, type, frequency and anticipation. Since the frequency varies it is essential to use prediction techniques to anticipate change.

Adaptation as action is reacting to change. The implementation of adaptation will have the following attributes: type, autonomy, scope, duration, timeliness, and triggering. When finished there will be an impact of the adaptation on the system. The impact will be measured by criticality, predictability, overhead and resilience.

Scaling

Scaling can happen vertically or horizontally [9]. Vertical scaling means adding resources to the system by increasing the capacity of existing servers. More of the hardware's resources are directed towards the processing related tasks. Vertical scaling is limited by the maximum capacity of the hardware.

Horizontal scaling means adding more servers to the system. In modern cloud architectures this can be done dynamically. As traffic increases the demand is met by

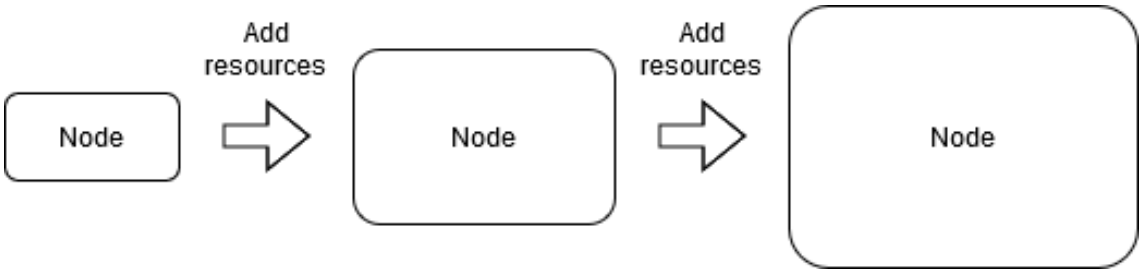

Figure 2.5. *Vertical scaling*

adding more servers to the system. With lowered traffic the system optimizes resources by decreasing the number of servers automatically. [9]

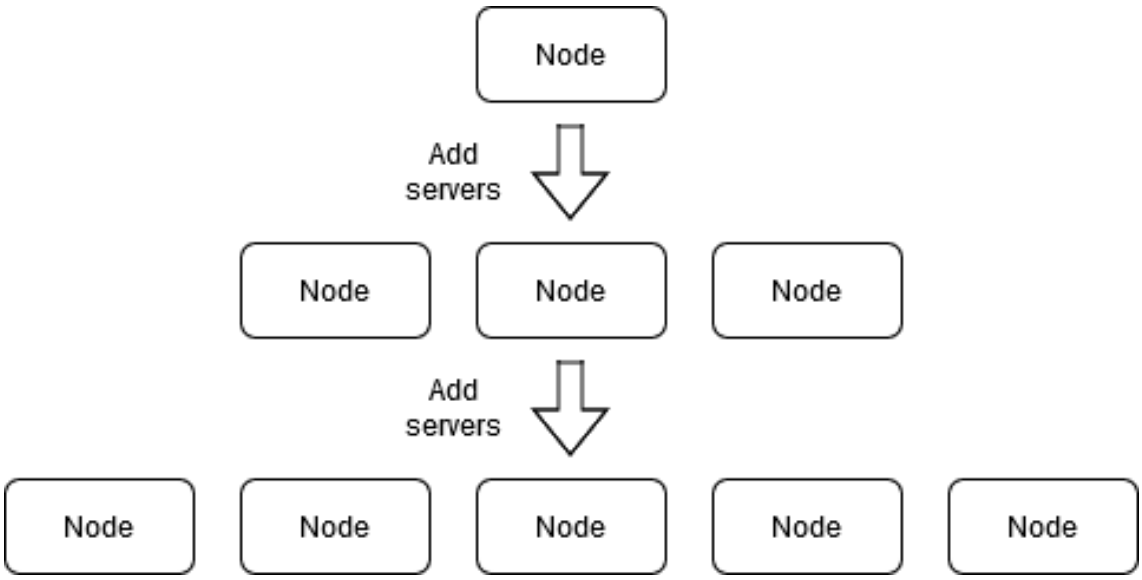

Figure 2.6. *Horizontal scaling*

2.1.3 Uncertainty

Cloud as a multi-tiered modular environment adds a high degree of uncertainty to reliably measuring system qualities. Pahl et. al [5] identify the following different sources of uncertainty.

Uncertainty in adaptation and its models. Choosing adaptation policies can be a subjective process. The policy may require continuous re-evaluation because of unpredictable changes in environment or application demand. Adaptation thresholds are dependent upon knowledge of system behaviour. The also rely on knowledge on how resources are managed.

Uncertainty in dynamic provisioning environment. Managing resources in the cloud is not immediate. During this window that can be minutes for Virtual Machines, the application is at risk to workload rises, causing uncertainty.

Uncertainty in monitoring data. To control one needs to continuously monitor application

states and resources in which applications are deployed to promptly react to variations in load. Monitoring is essentially measuring that is prone to deviations and noise.

Uncertainty in change enactment. Same change can take different times based on uncertainty in underlying resources.

3 MANAGING APPLICATIONS IN KUBERNETES

Kubernetes is a tool that wraps around the paradigms defined in chapter 2.1. It abstracts away the hardware infrastructure to expose the user an entire data center as a single computational resource. It allows the user to deploy software services and run them without having to deal with the actual servers to which the tasks are distributed. When deploying multiple services through Kubernetes, it allocates each service to a server, deploys it, and enables it to find and communicate with the other components of the whole application through an internal network.

Kubernetes is a software system that allows its users to deploy and manage containerized applications. It depends on the Linux container features to run unrelated applications without the need for manually deploying these applications on each possible host and without having to know any internal details of these applications. Because these apps run in isolated independent containers, they have no influence on other apps running on the same servers, which allows for running applications for completely different organizations on the same hardware.

From on-premises datacenters to the largest cloud provider operated datacenters, the larger the datacenters, the more Kubernetes starts to be useful. Increasing complexity handled by a simple platform is very useful for developers to deploy and run any type of application, while the hardware providers can remain completely oblivious to the multitude of apps running on hardware they own. Figure 3.1 illustrates the abstraction for developers working with Kubernetes. [10]

The Kubernetes architecture depicted in figure 3.2 consists of nodes. Nodes represent different hardware units with boundaries. A Kubernetes cluster can have two kinds of nodes: master nodes and worker nodes. Master node controls and manages the whole Kubernetes system through the Control Plane. The applications deployed by developers are run on the worker nodes.

The Master node is made of four vital components that control the cluster and enable it to function. Kubernetes API Server is the main communication endpoint for developers and other control plane components. Etc Distributed (etcd) is a key-value storage that acts as a persistence storage for the API server and holds the state of the cluster. Scheduler assigns a worker to each Kubernetes component of an application. Controller Manager performs cluster-level tasks, such as keeping track of the worker nodes, handling node failures, and replicating components. These four components can also be replicated and

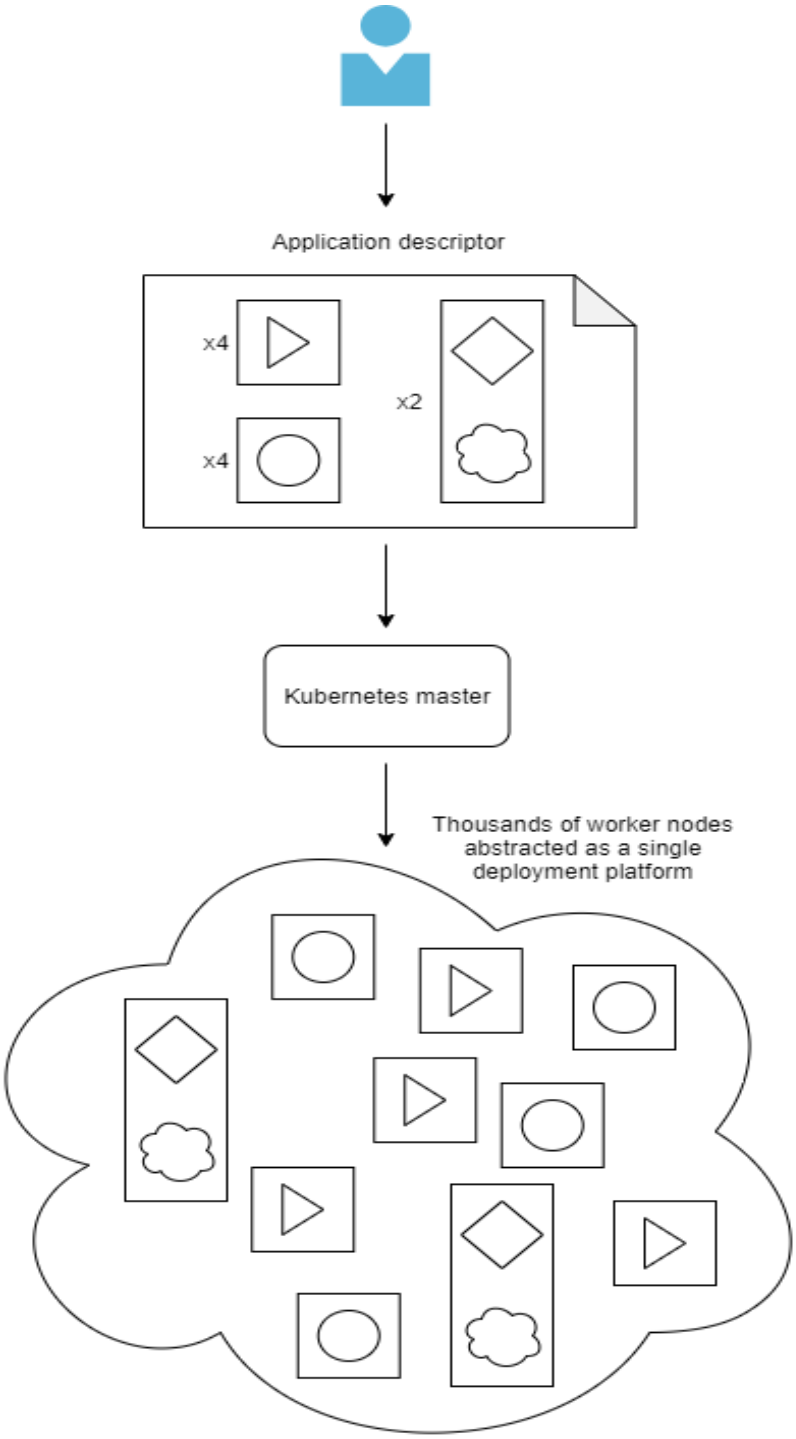

Figure 3.1. *Kubernetes use-case illustration [10]*

split to multiple master nodes to ensure high availability.

The worker nodes run the containerized applications. To run, monitor and provide services to applications the worker nodes use three components. A Container run-time runs the containers. The most common ones include Docker and rkt. Kubelet manages containers on its node by communicating with the API server. Kubernetes Service Proxy load-balances network traffic between application components. [10]

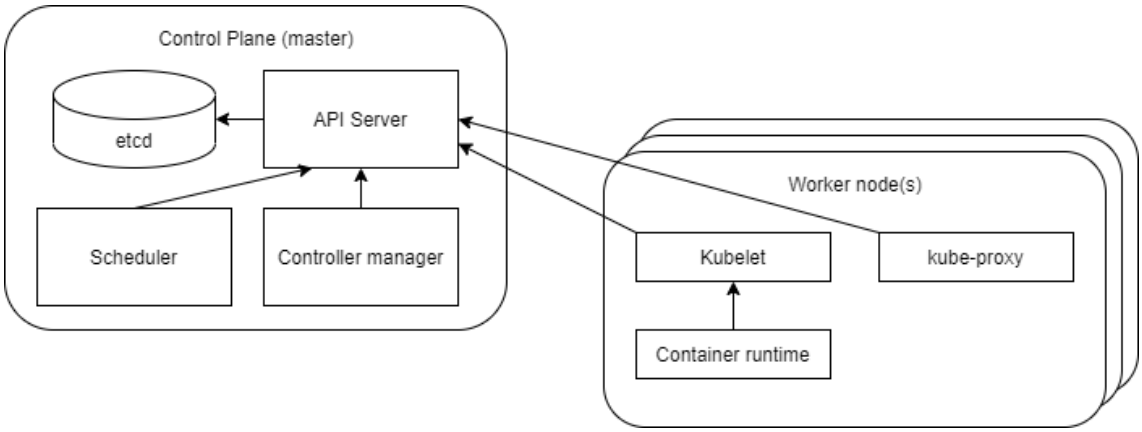

Figure 3.2. *Kubernetes cluster architecture [10]*

When looking at how Kubernetes controls the state of the cluster it is important to note that the API Server does not do anything except store resources on *etcd*, and notify clients about changes in these resources. Etcd acts as a persistence layer for the API Server. A component can request to be notified on a resource creation, modification or deletion. Clients watch for these events through an HTTP connection to the API server. The actual state of the system is ensured by controllers running in the Controller Manager. It combines multiple controllers for different kinds of resources. These controllers are not aware of each other and only communicate with the Kubernetes API.

3.1 Configuration Patterns For Kubernetes

This chapter compares different configuration patterns in Kubernetes. The chapter introduces four different configuration patterns. Applications need configuration for accessing external services, production-level tuning and data sources. Instead of hard-coding configuration in applications the configuration should be externalized so that it can be changed even after building the application.

3.1.1 Environment Variable Configuration

This pattern describes the simplest way to configure applications. Putting configuration into universally supported environment values is the easiest way to externalize configuration for small sets of configuration values. [11]

Using environment variables for storing the application configurations is recommended by The Twelve-Factor App. The approach is simple and works for any platform or environment. There is no operating system that cannot define environment variables or make them available to applications. [11]

Environment variables are easy to use and widely known. The concept fits well with containers and all the runtime platforms support environment variables. But they are not secure and for the management of these values to not become too hard, they shouldn't

exceed a decent amount. With a large number of environment variables a level of indirection is often used. The configuration is put in various configuration files, one file conforming to a single environment. A single environment value is then used to select one of these files. An example of this approach are profiles from Spring Boot. This couples the configuration tightly with the applications and causes image rebuild for every change in any environment. **[11]**

As environment variables are universally applicable, they can be set at various levels. This makes it hard to track where a given environment variable comes from because of the fragmentation of the configuration definitions. If defining environment variables is not constricted to one place, it might be hard to find the definition of a variable. And even if it is a clear location, it may be overridden in another location. For example, it is possible to replace environment variables defined within a Docker image during runtime in a Deployment resource in Kubernetes. Without a central place where all the environment variables are defined, the debugging of the configuration becomes an issue. Environment variables, while simple to use, are mainly applicable for simple use cases and face difficult limitations when the configuration is more complex. [11]

3.1.2 Configuration Resource

Kubernetes provides native resources for configuring ordinary and confidential data. This allows for decoupling the lifecycles of configuration and application. The Configuration Resource pattern describes the ConfigMap and Secret resources, their usages and limitations. [11]

Configuration resources are introduced as a more flexible option for keeping all the configuration data in a single place instead of distributing it around to various resource definition files, or putting the whole file into an environment variable. ConfigMaps and Secrets provide storage and management for key-value pairs. The keys of a ConfigMap holding data can be used in two ways: as a reference for an environment variable, key being the name, or as a filename, as files are mapped to a volume mounted in a Pod. With the file mode mounts are updated when the ConfigMap is updated via the API, making hot reload of configuration possible. Environment variables can't be changed after a process has been started, thus hot reload is not possible in environment variable mode. [11]

Secrets are a secure version of the ConfigMap and can be consumed as environment variables. They are only distributed to nodes running Pods that need access to them. They are stored in memory in a temporary file-system and removed with the Pod. They are never written to physical storage. They are stored in encrypted form to Etcd. [11]

ConfigMaps and Secrets are used to store configuration information in dedicated resource objects easily managed with the Kubernetes API. The main advantage of using ConfigMaps and Secrets is decoupling the usage of configuration data from its definition. It allows to manage the objects by using configurations independently from the applications. Some limitations of the Configuration resources include a 1 MB size limit

for Secrets and an individual quota on the number of ConfigMaps that can be used per project or namespace. [11]

3.1.3 Default Values

Default values make developing through an interface easier, as they abstract away configurations that one might not need or know to exist. The *convention over configuration* paradigm, which encourages decreasing the number of decisions the developer using the framework is required to make, is further enabled by using default values. [11]

If changing default values is made a difficult task they might become an anti-pattern for an evolving application. Stakeholders relying on defaults will always be surprised when a default value changes. The change must be communicated. A change in default values should always be considered a major change and with semantic versioning a change in default values should always be an increase in the major version number. [11]

If a given default value does not satisfy the requirements it's better to remove the default value and compel the user to provide a configuration value. This avoids silent unexpected behavior and makes the application break as early as possible. [11]

Default values should only be used for values that can be reliably predicted. A reasonable default should last for a long time. Using defaults prompts the use of a clear and indicative documentation. [11]

3.2 Design Patterns For Kubernetes

This sections explains few of the main software design patterns used around the subject of configuration an application management on Kubernetes. The chapter introduces three different design patterns.

3.2.1 Declarative Deployment

Declarative deployment is a design pattern that allows for using different strategies for updating applications on Kubernetes. This pattern is encapsulated in a Kubernetes concept called Deployment. With one it's possible to use different strategies and tune the process of updating an application. This makes sense when updates are required to be made considering the frequency of the updates and the efficacy of the scheduler. The scheduler requires containers with adequately defined resource policies, appropriate placement policies and sufficient resources on the host system. [11]

For the Deployment to be able the start and stop a group of Pods in a predictable manner, containers need to listen for lifecycle events e.g. SIGTERM, and to provide health-check endpoints to signal that they have started successfully. Should they do so, the platform can shut down containers cleanly and start updated instances to replace them. When

these concepts have been satisfied, everything else in the upgrade process can be defined in a declarative way. The update can be executed with predefined steps and an expected outcome making it a one atomic action. [11]

The Deployment turns the manual process of updating applications into a declarative task that can be automated. Ready-made deployment strategies, rolling deployment and plain recreation govern the replacement of expired containers by new ones, and the release strategies, blue-green deployment and canary deployment govern how a new version of the service is brought available to the consumers. [11]

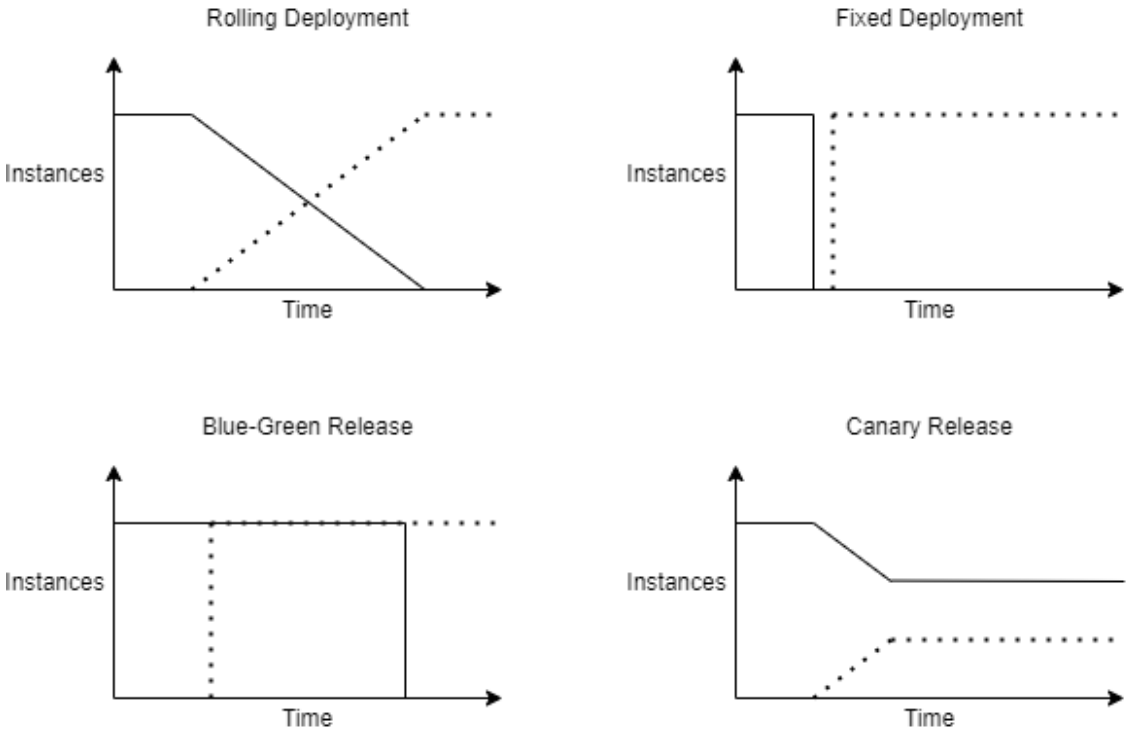

Figure 3.3. *Deployment update strategies [11]*

Rolling update removes downtime from the update process. This requires the application to have replicas, or multiple pods running. One by one the old pods are terminated and after a termination a new pod is spun up onto its place. This, compared to the fixed deployment where old pods are first shut down after which the new ones are started, ensures that a replica of the application is always running. [11]

Blue-Green release is used in production environments to minimize downtime and reduce the risk of for example data loss in a data processing application. There is no out-of-the-box automated way for Blue-Green releases in Kubernetes. The idea is to apply a second e.g. Deployment on side of the existing one. Once the deployment is up and running the traffic is directed from the old one to the new one. Then it is safe to delete the old deployment, since it is not serving any consumers anymore. [11]

Canary release deploys a new version of an application softly. In a canary release only a small subset of old service instances are replaced with new ones. The strategy let's only some of the consumers reach the updated version. After the new version has proven to

be adept, can the rest of the instances be replaced. [11]

3.2.2 Managed Life Cycle

Containerized applications managed by cloud-native platforms adapt their lifecycles by listening to the events emitted by the managing platform. They do not have any control over their own lifecycle. The Managed Lifecycle pattern describes the best practices on how applications should react to these lifecycle events. [11]

Kubernetes as an example of a cloud-native platform provides the ability to run and scale applications predictably and reliably on top of sometimes unreliable cloud infrastructure. Applications running on these platforms conform to a set of constraints and contracts. The applications honor these contracts so that they are able to benefit from the capabilities of the platforms. Graceful start up and shutdown without widely impacting the consumer services is achieved as a result handling and reacting to the lifecycle events. Application lifecycle should not be considered to be in the control of the user, but fully automated by the platform. [11]

There are different events for controlling a container. Events such as SIGTERM, and SIGKILL signals are emitted by the platform. In addition to these process signals, lifecycle hooks such as postStart and preStop are provided. A container can listen to these events and react to them. [11]

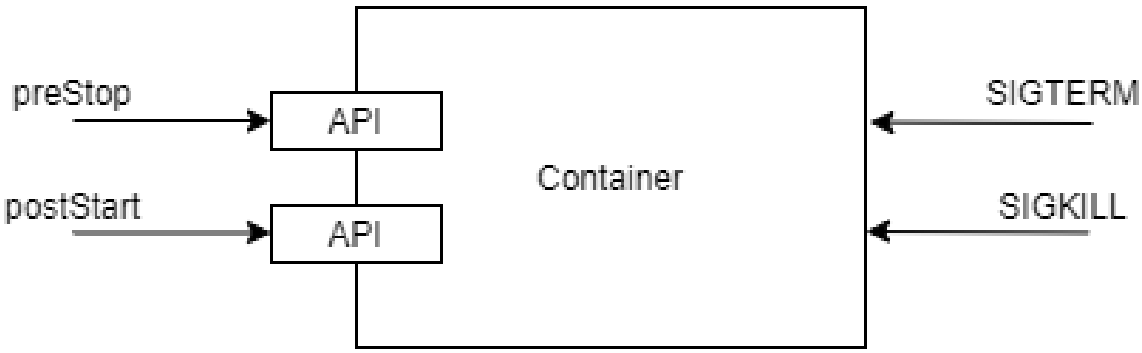

Figure 3.4. *Managed container lifecycle [11]*

The SIGTERM signal is the correct moment for a container to shut down in a clean way. The container can receive the signal if Kubernetes has decided to shut it down. It can be because of a Pod that the container belongs to is ordered to shut down, or it can be due to a failed liveness probe. The signal acts as a gentle nudge for the container to shut down as quickly as possible. This can mean different things for different containers. [11]

If a container has not shut down after the SIGTERM signal, or the following default grace period of 30 seconds, a SIGKILL signal is issued which shuts down the container forcefully. This motivates designing and implementing applications with swift startup and shutdown processes. [11]

The postStart hook comes into play after the container has been created. It is

asynchronous with the container's process. It is defined as a 'blocking call' which means that the container status remains Waiting until it completes. This keeps the Pod status in the Pending state. Different use cases include delaying the startup state of the container, and preventing container from starting before fulfilling a set of preconditions. [11]

The preStop hook comes into play before the container is terminated. It follows the principles of SIGTERM signal and should be used for a graceful shutdown when it's not possible to react to the SIGTERM signal. As in the hook is executed before triggering the SIGTERM. [11]

There are more lifecycle controls on the Pod level. An init-container is an initialization mechanism where a container runs sequentially, until completion and before any of the application containers in a Pod start up. They can be used for Pod-level initialization tasks. [11]

3.2.3 The Operator Pattern

In a microservice architecture the systems comprise of multiple services that need to be configured to work in conjunction to achieve the goal of the business process. The cloud sets requirements for the systems to be adaptive and elastic.

In their paper [12] Cosmo R. D. et al. bring forth the problem of writing distributed and adapting software systems in the cloud. They especially emphasize the challenge of maintaining and reconfiguring such systems. They present frameworks that address the issue such as CloudFoundry [13], Canonical's Juju [14] and Fractal's FraSCAti [15]. In all the approaches the user assembles a working system out of components that have been particularly designed, or adjusted, to work in conjunction.

The user is required to select and connect the components and in the case of reconfiguration, a manual reassembly of the system is required. The user can also write code to reassemble the system. Cosmo R. D. et al. argue that automation is a key concept in tackling this challenge. When the need to reconfigure appears frequently, or the number of components and services involved increases it becomes necessary to specify configurations at a certain level of abstraction. This requires developing tools that enable the transformations that lead the state of the system configuration to a desired one declared by the user.

Kubernetes architecture makes it possible to extend the cluster with custom controllers such as the one in figure 3.5 without modifying the code of the platform itself. To do so, a custom client for the Kubernetes API, and a Custom Resource to control are created. This design is called the Operator Pattern. Result is a state controller that defines a flexible single interface for application configuration in the form of a Kubernetes Custom Resource. These controllers are called Operators.

The operator pattern is a documented pattern for software extensions to Kubernetes that automate tasks that Kubernetes itself does not provide. The automation happens by

building a custom controller. A controller is essentially a control loop watching the state of the cluster through the API server and reacting to events with the goal of moving the current state towards a desired state. A simple control loop is illustrated in figure 3.5. Operators are clients to the Kubernetes API. Operator is a controller that handles Custom Resources. [16]

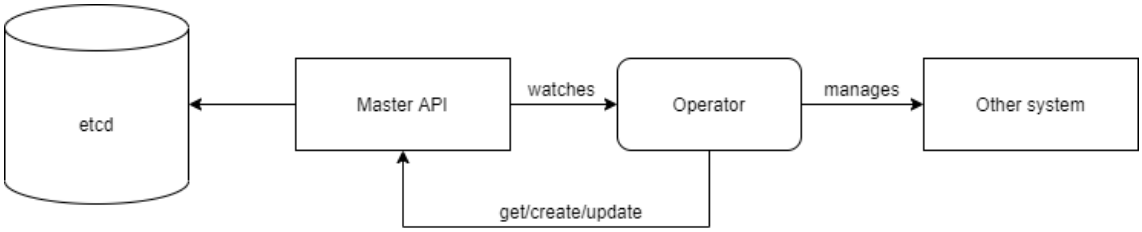

Figure 3.5. *Operator control loop [17]*

In Kubernetes, a resource is defined to be an endpoint in the API that stores a collection of API objects of a certain kind. Custom Resources are user-defined resources that extend the Kubernetes API. To get the Kubernetes API to handle user-defined resources, a Custom Resource Definition is required. Custom Resource Definition defines the resource's attributes and template for the API. The controller should only be able handle Custom Resources that conform to the CRD. [18]

3.3 Operator Reconcile Loop

Reconcile loop is a programmable concept where a controller watches the state of the cluster and determines the necessary actions to change it accordingly. It is a function, called reconcile loop, triggered by an event to watched resources. An Operator can for example watch changes to a Custom Resource and then create and maintain a child resource according to the state described in the Custom Resource. Figure 3.6 represents this use case.

When the Custom Resource is created the reconcile loop will create an object representing the Kubernetes resource. Kubernetes API is then requested to find out if the resource already exists. If it doesn't it gets created. Had the Custom Resource been updated instead of created, a check is made whether the child resource has been updated. If so, an update is triggered to the API. If any errors happen in execution there's a choice of requeueing the request for reconcile again.

3.4 Operator Practices for Production

There are great options for getting started with basic operator pattern programming such as the Operator Software Development Kit [19]. Although much of the development has been automated, the generated chassis covers only the very basic functionality, as in creation and deletion of resources. In the context of this thesis a production ready controller required some additions to the reconciling logic.

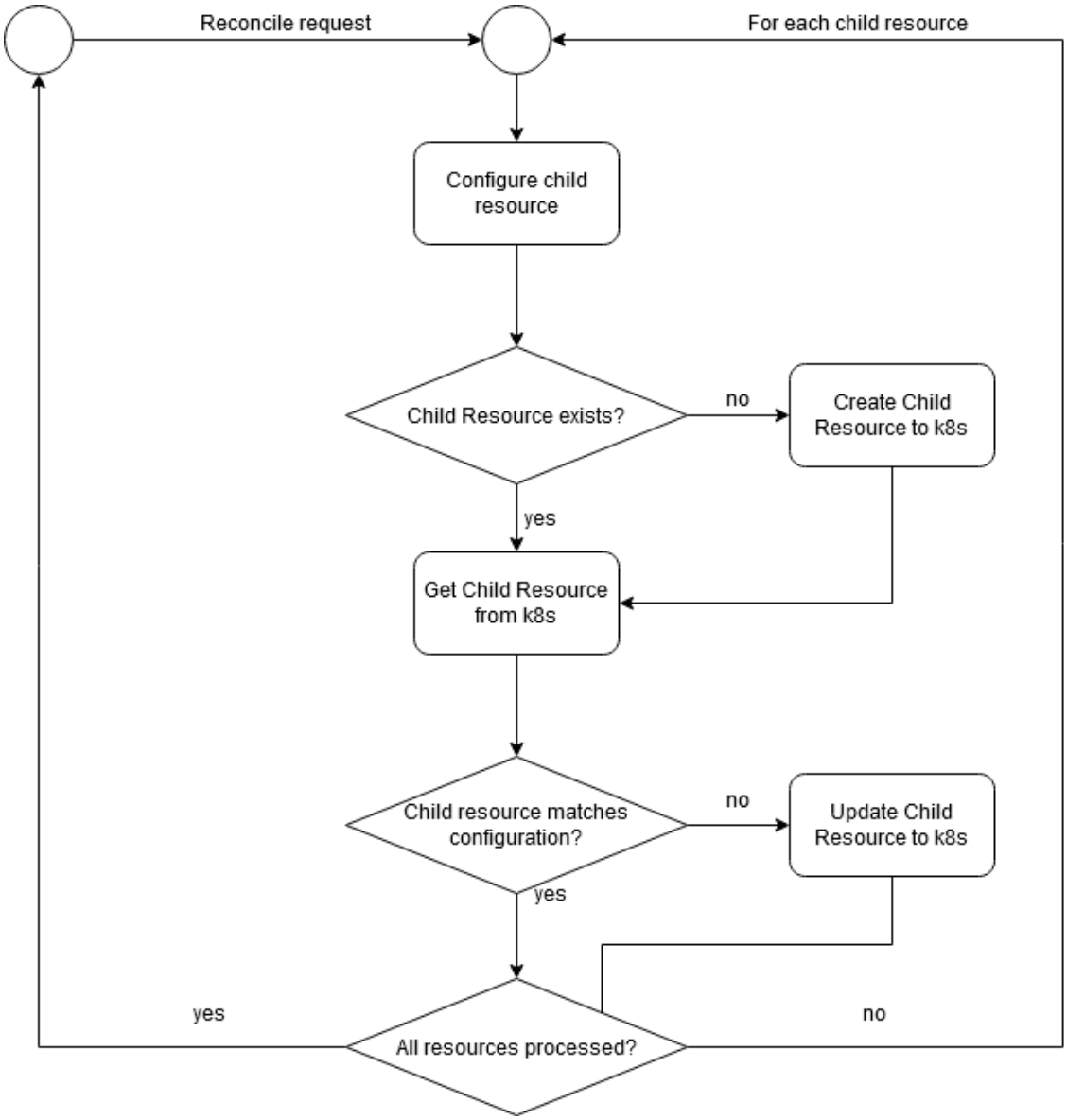

Figure 3.6. *Example of a reconcile loop*

3.4.1 Filtering Reconcile Requests

Reconcile request is a request that activates the execution of the reconcile loop in the controller. In some use cases the reconcile loop is inconvenient to be run on every event that happens on the watched resource. For example updating the status object of a Custom Resource may not need to trigger a reconcile request.

To filter out reconcile requests a *predicate* can be used. Predicates are filters programmed for the watch function to allow only specific events to trigger a reconcile event.

3.4.2 Handling Custom Deletion Tasks

If resource deletion does not require additional logic, deletion of the child resource is fairly simple and does not require code in the reconcile loop. An owner reference can be set of the child resource for the main resource. Then a deletion request to the main resource will result in all the referenced resources in its ownership to be deleted.

Sometimes deleting a custom resource will require logic that the basic owner reference will not cover. For example when there is a need to delete objects in some other API than the Kubernetes'. A solution is to catch a Custom Resource deletion event in the reconcile loop by using a Kubernetes finalizer. In the reconcile loop a finalizer is added after the object has been created. Once a resource with a finalizer is issued to be deleted, it gains a deletion timestamp. This in turn triggers a reconcile loop during which a check is made to look for the deletion timestamp, perform deletion related tasks, and finally delete the finalizer. A Kubernetes resource will get deleted only after its finalizers have been deleted.

3.4.3 Upgrading Custom Resource Definition

To upgrade a Custom Resource Definition simply deleting the old one and replacing it with a new one is not advisable. Deleting a Custom Resource Definition results in the deletion of all related Custom Resources from the cluster. Upgrading a CRD requires a thought out process where the CRD, existing resources, and the controller have been considered.

Upgrading the CRD version requires the controller to handle both the new and the previous versions. Then the old versions of Custom Resources are migrated to the new CRD version. After this can the CRD be updated again to drop the old version.

4 REQUIREMENTS FOR ABSTRACTION

Abstraction is a powerful tool in developing an understanding of complex phenomena. Abstraction is a result of recognizing similarities between certain objects, processes, or situations, and the act of concentrating on these similarities while ignoring the differences for the sake of ephemeral simplicity. The relevant similarities to the control and prediction of becoming events form the fundamentals of the abstract concept. The differences are trivial in the context. The abstract concept can be represented by a word or a picture or any other symbols. [20]

Abstraction can be seen as simplification. At every stage of the abstraction the amount of information needed to handle reduces. This makes abstraction a relation, where the developer of the abstraction is able to choose the level of simplification and reduction. [21]

4.1 Abstraction Layers

The top-most abstraction fundamental in the context of this thesis can be described as transforming streams of data in a controlled manner. This merely brings forth the idea that we have an influx of data and that we want to have a different outcome than what was ingested. What the data contains, what is done to process it, or where it's then stored or mediated to is not information that is defined on the top level. To deeper develop the understanding of this concept, a closer look is taken to identify more practical levels of abstraction. The business requirements and current implementation details are considered. This requires starting at the lower levels to build and identify the abstraction levels in play. To succeed in transforming streams of data in a controlled manner, three things are needed: computing power, control over the computing power, control over data ingestion and transformation.

The computing power is provided to us by the cloud platform, be it public, hybrid or private. Hence the cloud platform that we operate on represents the first abstraction layer. It abstracts away all the details about how to utilize and maintain the hardware which provides the computing power.

The second layer is offered by Kubernetes. Kubernetes gives us control over our computing power. As explained in chapter 3 Kubernetes takes multiple server instances and exposes them as a single computational resource. It gives us tools to harness the computing power and the networking of the servers through documented resources.

These resources and their configuration are abstractions of allocating resources and opening communication channels between the services in the cluster.

The third layer is control over data. Apache Spark is used as a tool that provides means of ingesting and transforming data streams on Kubernetes. The logic for data ingestion and transformations can be written in programming languages such as Scala, Java and Python using the libraries that Spark offers. Spark abstracts away the act of distribution of the data computing to all the nodes in a Kubernetes cluster. The code is written in one place as if the transformations would happen on a single computer, but the computing tasks are under the hood distributed to all the available and configured computing nodes at the cluster's disposal. [22]

As Apache Spark requires specific expertise to use, further abstraction is required. Thus the fourth layer is control over the ingestion and transformation logic. In the context of this thesis a general configurable framework exists to offer different kinds of transformation collections without the need for reprogramming the Spark application. This configuration is an XML file.

The configuration file can be built by using a specific User Interface. This UI allows to visualize and draw to generate a configuration for the fourth layer. The UI can be seen as the fifth layer of abstraction.

Between the fourth level and the fifth one, the amount of information needed to handle does not currently substantially reduce. It is still required for the user to familiarize the framework that is being configured since it requires knowledge of low level programming skills to fully conduct a configuration.

4.2 Requirements and Preconditions

This thesis in its context defines a successful abstraction with the following statement: A domain expert should be able to create and modify data processing workflows without low level programming skills. To achieve this the domain expert would need no knowledge of the framework that is being configured to provide a sound product configuration. Also, an argument should be considered whether deploying the application should even require knowledge of Kubernetes. Actors should be able to configure and deploy the application in Kubernetes from an abstracted layer. The concept can be likened to the specialists having access to a driveable car, as opposed to just the engine. Whether this is fully achieved also depends on other requirements of the system. The abstraction will likely fall to some point on the abstraction spectrum by Rosso et. al in figure 4.1. [23]

The main goal of the successful abstraction is to make domain experts successful service consumers in the cloud platform. With the spectrum, a choice can be made to make Kubernetes completely transparent to the developing parties. Other option is to expose as much of the flexibility as Kubernetes allows to a developer. [23] Third, with default values, an operator and a multi-layered development process the abstraction does not

Figure 4.1. *Abstraction spectrum [23]*

necessarily have to be a static choice. By setting levers in the automation a complete flexibility can be allowed for the developer to determine how deep in the layers one wishes to go. It can be possible to just use the final layer of abstraction, or, still do some modifications on the next layer. This means that the abstraction provides simplicity but does not force it.

The abstraction should be able to expose key features of Kubernetes. If not, over-time the abstraction can become as complicated as the system that it's abstracting. The developers of the underlying *engine* can not consider the delivered engine to be a static environment that is used for decades by the domain experts. Rosso et. al [23] argue, that a constant communication between developers and the domain experts is needed to determine increases in development velocity due to missing features or issues.

Versioning interfaces can structure this communication and make it a less strenuous process. Novakouski et. al [24] concluded that key artifacts created during the development of a service-oriented system should be included in a versioning solution to avoid potential conflicts and promote proper change management by leveraging versioning tools and recommended practices. For example, the company that ordered this thesis has adopted the *Semantic Versioning 2.0* [25] tool.

However not only the domain expert can be considered. The development work and troubleshooting of the underlying framework should not suffer from the abstraction. The performance, test-coverage and other quality measuring aspects of the existing software should be at least preserved or enhanced. While not the focus in the context of this thesis it should be made sure that no new quality issues arise due to the architecture that enables our abstracted interface. The following set of requirements for the architecture and interface are derived:

- The interface is intuitive and does not require low level programming skills
- Deploying a new workflow is automatic and efficient
- Updating an existing workflow is automatic and efficient
- The architecture follows cloud native principles

  - Data processing workflow scales
  - Data processing workflow adapts to uncertainty in the cloud
- Development workflow of the engine does not suffer from the architecture
- The functionality and quality aspects such as performance, or test-coverage of the existing engine are not worsened by the architecture

5 PROPOSED CONSTRUCT

The goal of this chapter is to provide an architecture that enables our abstracted interface by fulfilling the requirements described in the chapter 4. The proposed model is divided into three sections: *Development*, *Abstraction* and *Automation*

5.1 Development

Development of the data pipelines happens with a service specific GUI (Graphical User Interface) designed for building the configuration for the data pipeline engine. Until now the GUI has only produced a single configuration file that maps the data transformations, but the arrangement and deployment of the application requires knowledge about Helm, Kubernetes, The code base and the underlying technology stack.

Figure 5.1 introduces an enhanced workflow. A packaging automation is introduced to the GUI to produce the resources required by the abstracted interface. This poses a requirement for the interface to be as simple as possible so that the packaging automation does not introduce unwanted complexity.

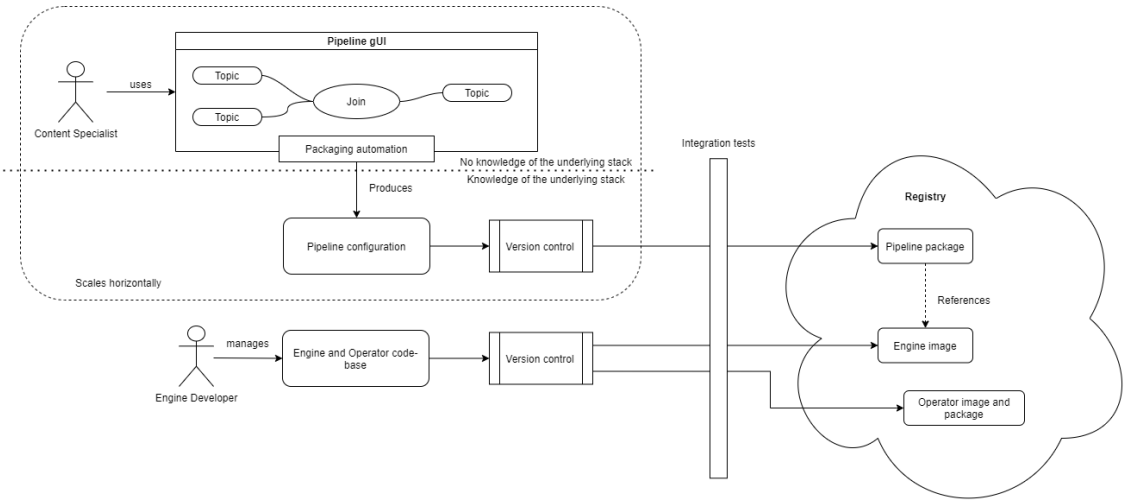

Figure 5.1. *Development roles and workflow*

Deployment through the GUI can be automated within the software supply chain. By packaging automation the GUI resources would be downloaded to a version controlled project with a single click of a button. The GUI generates a job package with all the necessary resources. Then the project is pushed to version control and ran through a

generic Continuous Integration process (CI) to parse arguments and to arrange resources into a deployable package. After passing functional tests the artifacts are deployed to a staging environment for end-to-end testing.

The enhanced workflow decouples development of the data pipelines from development of the engine. This enables horizontal scaling in development of the pipelines. More customers can be on-boarded and pipelines for existing customers created as long as a domain expert is available to interpret the specification of the case. In order for the development of product configurations to scale horizontally, the development of the engine has to scale vertically, meaning that constant maintenance, support, and possibly new features must be considered in the development of the engine.

5.2 Configuration

For the development process to be as simple as possible the amount of information passed to the deploying interface must also be as little as possible. To make the Development process possible the underlying technology stack must be abstracted. The prerequisite for the interface that abstracts the underlying technology stack is the use of Kubernetes as the platform. In the development of the interface the requirements for automation must be taken into account for the users be able to benefit from declarative deployment in the stages of product deployment.

As Kubernetes is used as the platform it makes sense for the interface to rely on Kubernetes as well. There are means of extending Kubernetes with the Operator pattern defined in subsection 3.2.3. The Custom Resource Definition is introduced as the main interface for a data pipeline. With it, we can define the properties needed for configuring the engine which, in turn, interprets the configuration and produces a pipeline. Figure 5.2 showcases the architecture choices

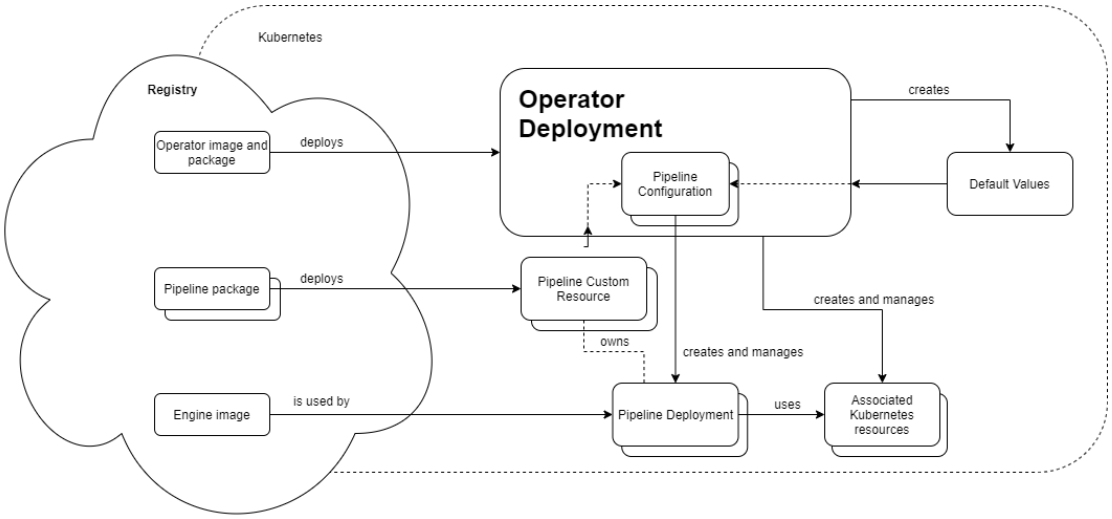

Figure 5.2. *Deployment and configuration of the pipeline with Custom Resource and Operator*

As pipelines utilize Apache Spark they require multiple configurations and Kubernetes resources. Managing all these values by the environment values configuration pattern is not recommended for reasons explained in subsection 3.1.1. A lot of the configuration can be abstracted by utilizing default values along with the Operator. Default values are agreed and controlled by the engine developers, and the possibility is given for the Custom Resource to override these values. Default values do not contain any business logic, but information about the cluster, and tuning parameters for Apache Spark. The implementation of the configuration uses the Kubernetes Configuration Resource design.

5.3 Automation

As described in chapter 2, cloud software lives in a highly dynamic environment. To achieve the characteristics explained in the chapter such as adaptivity and elasticity, the construct employs the Kubernetes Operator pattern. The operator is responsible for managing the Kubernetes resources. It utilizes the Kubernetes API to watch for events in etcd. Upon a Custom Resource creation event, the Operator picks up the event, reads the Custom Resource properties and starts to execute a Reconcile loop described in section 3.3. The Reconcile loop creates all the necessary child resources Spark needs to be able to start a pipeline. Resources in this example include Kubernetes Configuration resources, Network Services and Storage claims. Pipelines also report metrics during processing which are monitored by an external service to which the Operator provides the configuration by API. Message queue topics are also handled by the Operator.

On an upgrade the operator performs updates on all the child resources based on the Custom Resource. The Custom Resource is the only resource it watches for changes. Filtering the rest by predicates is necessary since on an update the operator does not benefit from multiple cascading reconcile results. The control needs to be idempotent and atomic. This means if a node crashes with a metrics service on it, the operator does not know it is gone. To fix this a periodic reconcile can be introduced to ensure adaption.

When a Custom Resource gets deleted all child resources will be deleted due to them being owned by the Custom Resource. Deletes to external API's are handled with custom deletion mechanisms, such as described in subsection 3.4.2

Any resource malfunction that will abrupt the data pipeline will result in the pipeline crashing and the operator creating a new one. If the Operator dies or the node it operates on crashes, the pipeline will still live should its critical components reside on another node. Thus, custom scheduling the Operator on a different node than the Spark driver is required to ensure guaranteed adaptation. Also, it is possible, by adapting the declarative deployment pattern from subsection 3.2.1 to upgrade the operator via fixed deployment while the pipelines are running. To prevent data loss a Blue-Green release is recommended strategy for the pipelines, but since the pipelines can be very resource heavy, in on-premises cloud solutions with limited quota, fixed deployment with minimized container stop and start-up times should be used.

Scaling is possible statically by updating the Custom resource or dynamically by enabling the Spark dynamic allocation. The dynamic scaling benefits from the Managed Lifecycle Pattern described in subsection 3.2.2 that Kubernetes as a platform implements. Spark worker instances, which are scheduled Kubernetes Pods on the platform can be deleted and scheduled on the fly via API calls, based on ingested traffic. Enabling dynamic allocation sets a requirement for the engine algorithms to be able to scale dynamically. The benefits can be questioned in private cloud environments with limited quota if the full capacity can be used.

6 EVALUATION

The thesis set a desired state of the design-oriented constructive research by the following statement: a domain expert should be able to create and modify data processing workflows without low level programming skills. This was to be done through an abstracted interface so that no low level programming skills would be required. A graphical user-interface exists which is used to produce all the configuration components of a pipeline. Packaging some of the configurations require skills that are not expected to be possessed by the domain experts not allowing to reach the desired state. To simplify and reduce the information needed to produce a data pipeline, the packaging automation and further deployment automation aspects were covered.

What the thesis does not consider however, are the details of the GUI used in the configuration to actually provide the simplest, most expressive way to define the actual data pipeline. This would need collaborating and interviewing the domain experts and defining their workflows with the GUI through carefully thought out processes. What should be considered further is the functionality of the GUI. The GUI needs to be expressive enough for domain experts to be able to create evolving data pipelines. The GUI needs to have a good enough user experience and proper workflows for the domain experts to be willing to use it. Since it was determined that abstraction requires a substantial reduction in information about the system between the final layers, the thesis overlooked the fact that on the final layer information needs to evolve in order to enable a new user group. The GUI development and enhancements were left out of the thesis as out of scope and the thesis concentrated more on how the foundation of the abstraction must be built for us to be able to start working on matters of content.

To enable this, the requirements listed in section 4.2 were defined. These only answer the first research question with a broad non-functional requirements, but do not provide the actual functional requirements. As of now the GUI focuses more on the system than on the content which directed the improvement work more towards the system. Hence, the thesis focused more heavily on the system requirements. The thesis actually answered the question: *What requirements are there for the system to abstract away unnecessary information to create and modify data processing workflows*

The domain of the construct was restricted to Kubernetes in this thesis. It introduces Kubernetes design patterns that enable simple resources for an interface, that can consequently enable the implementation of automation required to fulfill the set requirements. The domain was restricted to Kubernetes so other options were not

Requirement	*Fulfilling Construct*
The interface is intuitive and does not require low level programming	Kubernetes Custom Resource and Engine Configuration
Deploying a new workflow is automatic and efficient	Custom Resource and Kubernetes Operator
Updating a new workflow is automatic and efficient	Custom Resource and Kubernetes Operator
The architecture follows cloud native principles	Kubernetes Operator and Apache Spark enable scaling and adaptive control
Development workflow of the engine does not suffer from the architecture	Decoupled Development Process for Products and the Code Base
The quality of the existing engine is not worsened by the architecture	Decoupled Development Process for Products and the Code Base

Table 6.1. *Requirements and their constructs*

considered. The thesis introduced the two sure choices for implementing the interface which were the deployment resource and operator pattern. The Operator Pattern was chosen, because when working with deployments, the additional Kubernetes resources that the pipelines utilize would have needed to be configured before the deployment. A Kubernetes Custom Resource could be used to describe the state of all the resources in a single artefact. With the operator the description would automatically be transformed into the state on the cluster creating all additional resources based on the description. Since the use of a Custom Resource and a Custom Controller is becoming a standard industry practice they were trivial to choose for implementing the interface. As the domain was well restricted the scope for different design choices was quite narrow. However, answering the second research question made it clear that using ready made resources from the Kubernetes toolkit would not give us a good enough abstraction.

Kubernetes consists of an API which is used to declare a state on the cluster. Controllers then achieve the state in the cluster by deploying resources defined by Kubernetes. The nature of evolving and frequently changing configuration was answered with automation of the service composition and configuration. Automation was also used to ensure high availability of the system as per common cloud requirements. The thesis explored the relevant tools in our disposal for achieving the design of automated service composition via the Operator pattern, comprised of a Custom Controller and a Custom Resource.

A construct was proposed as the architecture of the system in chapter 5. With the requirements and preconditions characterized as constructs the thesis proposes a Kubernetes interface that implements the desired abstraction. Table 6.1 lists the requirements and the constructs that fulfill them. With the use of Kubernetes Custom Resource, Operator and Default Values design patterns, enough information could be abstracted away to enable a more simple configuration to enable a packaging automation. Everything after the deployment event is automated to the point of keeping the system healthy assuming a proper configuration. The deployment event happening on an already

running system results in an automated update process where the state defined by a proper configuration is achieved in a declarative manner. Managed Lifecycle and Declarative Deployment patterns were properly implemented in the design. A level of cloud native status was achieved by utilizing the Kubernetes design patterns.

7 CONCLUSION

This paper introduced a cloud controller architecture which enables decoupling the development processes for an underlying code base, and product configurations. The architecture facilitates an abstraction in the form of a configuration interface, that enables domain experts to take over the customer-centric configuration work, freeing software developers as resources to feature development. After some additional work, clear boundaries between these two concepts can now more easily be established.

There are possible identified caveats for the system. The learning curve of the interface might prove to be too steep in the sense that even with development, administrating and monitoring capabilities time resources are still needed from the developers for instructing and supporting customer production deployments. Also, what's left for future to show is whether maintaining the increased complexity of the underlying system will need as much resources as potentially freed.

What was left out of the thesis is the administration, monitoring and troubleshooting of the system. The capability for these is required to be available for service care personell who support customer deployments. Configuration for logging and monitoring systems provided by the cloud platform are included to be installed by the deployment automation. A comprehensive documentation is written for the interface for on-boarding of the domain experts in order to maximize the benefits of the unevolved interface. Future work could investigate the expressive qualities of an interface and its user experience in order to evolve it to an even more simple product configuration.

REFERENCES

[1] Järvinen, P. and Järvinen, A. *On research methods*. Opinpajan kirja, 2018.

[2] *Who Coined Cloud Computing*. Oct. 2, 2020. URL: https://www.technologyreview.com/2011/10/31/257406/who-coined-cloud-computing/ (visited on 10/02/2020).

[3] *Innovations in Computing Sciences and Software Engineering*. eng. Dordrecht: International Conference on Systems, Computing Sciences and Software Engineering Corporate Author.

[4] Duan, Y., Fu, G., Zhou, N., Sun, X., Narendra, N. C. and Hu, B. Everything as a Service (XaaS) on the Cloud: Origins, Current and Future Trends. *2015 IEEE 8th International Conference on Cloud Computing*. 2015, 621–628.

[5] Pahl, C., Jamshidi, P. and Zimmermann, O. Architectural Principles for Cloud Software. eng. *ACM Transactions on Internet Technology (TOIT)* 18.2 (2018), 1–23. ISSN: 1533-5399.

[6] *PCI DSS Cloud Computing Guidelines*. Aug. 3, 2021. URL: https://www.pcisecuritystandards.org/pdfs/PCI_DSS_v2_Cloud_Guidelines.pdf (visited on 08/03/2021).

[7] Shroff, G. *Enterprise cloud computing : technology, architecture, applications*. eng. Cambridge ; Cambridge University Press, 2010. ISBN: 9780521760959.

[8] *Definition of Cloud Native*. Oct. 10, 2020. URL: https://www.infoworld.com/article/3281046/what-is-cloud-native-the-modern-way-to-develop-software.html (visited on 10/10/2020).

[9] Psaltis, A. G. *Streaming Data: Understanding the real-time pipeline*. Manning Publications, 2018.

[10] Luksa, M. *Kubernetes in action*. eng. 1st edition. Shelter Island, NY: Manning Publications. ISBN: 1-61729-372-5.

[11] Ibryam, B. and Huß, R. *Kubernetes Patterns: Reusable Elements for Designing Cloud-Native Applications*. eng. Sebastopol: O'Reilly Media, Incorporated, 2019. ISBN: 9781492050285.

[12] Di Cosmo, R., Zacchiroli, S. and Zavattaro, G. Towards a Formal Component Model for the Cloud. *Software Engineering and Formal Methods*. Ed. by G. Eleftherakis, M. Hinchey and M. Holcombe. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, 156–171. ISBN: 978-3-642-33826-7.

[13] *Cloud Foundry, The Proven Development Platform For Cloud-Native Applications. Cloud Foundry provides a highly efficient, modern model for cloud native application delivery.* Apr. 18, 2022. URL: https://www.cloudfoundry.org (visited on 04/18/2022).

[14] *Deliver, maintain, secure and sustain. open source from cloud to desktop and devices.* Apr. 18, 2022. URL: https://canonical.com (visited on 04/18/2022).

[15] Seinturier, L., Merle, P., Fournier, D., Dolet, N., Schiavoni, V. and Stefani, J.-B. Reconfigurable SCA Applications with the FraSCAti Platform. eng. *2009 IEEE International Conference on Services Computing*. IEEE, 2009, 268–275. ISBN: 9781424451838.

[16] *Operator Pattern*. Nov. 30, 2020. URL: https://kubernetes.io/docs/concepts/extend-kubernetes/operator/ (visited on 11/30/2020).

[17] *Kubernetes Operators Best Practices*. Aug. 3, 2021. URL: https://www.openshift.com/blog/kubernetes-operators-best-practices (visited on 08/03/2021).

[18] *Custom Resources*. Sept. 2, 2021. URL: https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/ (visited on 09/02/2021).

[19] *Operator SDK*. Nov. 4, 2021. URL: https://sdk.operatorframework.io/ (visited on 11/04/2021).

[20] Hoare, C. A. R. Chapter II: Notes on Data Structuring. *Structured Programming*. GBR: Academic Press Ltd., 1972, 83–174. ISBN: 0122005503.

[21] Bjørner, D. (*Software engineering. 1, Abstraction and modelling*. eng. 1st ed. 2006. Texts in theoretical computer science. Berlin ; Springer, 2006. ISBN: 3-540-31288-9.

[22] *Apache Spark - Unified engine for large-scale data analytics*. Aug. 21, 2022. URL: https://spark.apache.org (visited on 08/21/2022).

[23] al, R. et. *Production Kubernetes*. eng. O'Reilly Media, Incorporated, 2019. ISBN: 9781492092308.

[24] Novakouski, M., Lewis, G. and Anderson, W. *Best Practices for Artifact Versioning in Service-Oriented Systems*. eng. 2012.

[25] *Semantic Versioning 2.0*. Nov. 4, 2021. URL: https://semver.org/ (visited on 11/04/2021).