Dimensioning, Performance and Optimization of Cloud-native Applications

Jack Henschel

School of Science

Thesis submitted for examination for the degree of
Master of Science in Security and Cloud Computing.
Espoo, 16.7.2021

Supervisors

Prof. Mario Di Francesco
Prof. Raja Appuswamy

Advisor

M.Sc. Yacine Khettab

Aalto University
School of Science

Aalto University,
P.O. BOX 11000,
00076 Aalto,
FINLAND
www.aalto.fi
Abstract of the master's thesis

Author Jack Henschel

Title Dimensioning, Performance and Optimization of Cloud-native Applications

Degree programme Master's Programme in Security and Cloud Computing

Major Security and Cloud Computing **Code of major** SCI3113

Supervisors Prof. Mario Di Francesco
Prof. Raja Appuswamy

Advisor M.Sc. Yacine Khettab

Date 16.7.2021 **Number of pages** 67+7 **Language** English

Abstract

Cloud computing and software containers have seen major adoption over the last decade. Due to this, several container orchestration platforms were developed, with Kubernetes gaining a majority of the market share. Applications running on Kubernetes are often developed according to the microservice architecture. This means that applications are split into loosely coupled services that are distributed across many servers. The distributed nature of this architecture poses significant challenges for the observability of application performance.

We investigate how such a cloud-native application can be monitored and dimensioned to ensure smooth operation. Specifically, we demonstrate this work based on the concrete example of an enterprise-grade application in the telecommunications context. Finally, we explore autoscaling for performance and cost optimization in Kubernetes — i.e., automatically adjusting the amount of allocated resources based on the application load. Our results show that the elasticity obtained through autoscaling improves performance and reduces costs compared to static dimensioning.

Moreover, we perform a survey of research proposals for novel Kubernetes autoscalers. The evaluation of these autoscalers shows that there is a significant gap between the available research and usage in the industry. We propose a modular autoscaling component for Kubernetes to bridge this gap.

Keywords kubernetes, autoscaling, cloud-native, monitoring, performance

EURECOM
Sophia Antipolis

EURECOM,
Campus SophiaTech,
450 Route des Chappes,
06904 Biot Sophia Antipolis,
FRANCE
www.eurecom.fr
Sommaire de thèse

Auteur Jack Henschel

Titre Dimensioning, Performance and Optimization of Cloud-native Applications

Maîtrise Master's Programme in Security and Cloud Computing

Diplôme Security and Cloud Computing **PROMO** 2022

Superviseurs Prof. Mario Di Francesco
Prof. Raja Appuswamy

Superviseur entreprise M.Sc. Yacine Khettab

Date 16.7.2021 **Pages** 67+7 **Langue** Anglais

Sommaire

Le cloud computing et les conteneurs logiciels ont connu une adoption majeure au cours de la dernière décennie. Par conséquent, plusieurs plateformes d'orchestration de conteneurs ont été développées, parmi lesquelles Kubernetes a obtenu la majorité des parts de marché. Les applications fonctionnant sur Kubernetes sont souvent développées selon l'architecture de microservices, qui signifie que les applications sont divisées en services faiblement couplés qui sont distribués sur nombreux serveurs. La nature distribuée de cette architecture pose des défis importants pour l'observabilité des performances des applications.

Nous étudions comment une telle application cloud-native peut être surveillée et dimensionnée pour assurer un fonctionnement sans heurts. Plus précisément, nous démontrons ce travail en nous appuyant sur l'exemple concret d'une application d'entreprise dans le contexte des télécommunications. Enfin, nous explorons l'autoscaling pour l'optimisation des performances et des coûts dans Kubernetes — c'est-à-dire l'ajustement automatique de la quantité de ressources allouées en fonction de la charge de l'application. Nos résultats montrent que l'élasticité obtenue par l'autoscaling améliore les performances et réduit les coûts par rapport au dimensionnement statique.

De plus, nous réalisons une étude des propositions de recherche pour de nouveaux autoscalers Kubernetes. L'évaluation de ces autoscalers montre qu'il existe un écart important entre la recherche disponible et l'application dans l'industrie. Nous proposons donc un composant modulaire de mise à l'échelle automatique pour Kubernetes afin de combler cet écart.

Mots-clés kubernetes, autoscaling, cloud-native, monitoring, performance

Preface

First of all, I want to thank the SECCLO consortium for giving me the opportunity to participate in this exciting degree program and study at two excellent universities. During the course of this programme I was able to explore two countries, meet new friends and in general broaden my intellectual and cultural horizon. Special thanks to **Eija Kujanpää**, **Laura Mursu**, **Anne Kiviharju** and **Gwenaëlle Le Stir** for their administrative support during this time.

I would like to thank **Mario Di Francesco**, my main academic supervisor at Aalto University, for his continuous, comprehensive and honest feedback about my thesis. Also thanks to **Raja Appuswamy** for being my academic supervisor at EURECOM.

Yacine Khettab, my thesis instructor, gave helpful guidance while starting my work at Ericsson and proofread several drafts. I am thankful for **Adam Peltoniemi**, my manager, who hired me for Ericsson and gave me enough time to work freely on my thesis. I am grateful for the help of my colleagues at Ericsson, who provided inputs and guidance for setting up test environments for experiments: **Gábor Kapitány**, **Olli Salonen**, **Tomi Poutanen**, **Bálint Csatári** and **Jussi Tuomela**.

Furthermore, I want to express my gratitude to my good friends **Max Crone** and **Markus Opolka** for their invaluable feedback, discussion and proofreading of my thesis.

Finally, I would like to thank my editor, GNU Emacs, for allowing me to seamlessly develop, write and edit all aspects of the work covered in this thesis.

I warrant that the thesis is my original work and that I have not received outside assistance. Only the sources cited have been used in this thesis. Parts that are direct quotes or paraphrases are identified as such.

Otaniemi, 16.7.2021 Jack Henschel

With the support of the
Erasmus+ Programme
of the European Union

Contents

Abbreviations

API	Application Programming Interface
CA	Cluster Autoscaler (Kubernetes)
CPU	Central Processing Unit
CRD	Custom Resource Definition (Kubernetes)
HPA	Horizontal Pod Autoscaler (Kubernetes)
HTTP	Hypertext Transport Protocol
I/O	Input and Output operations
IP	Internet Protocol
IaaS	Infrastructure-as-a-Service
KEDA	Kubernetes Event-driven Autoscaler
KEP	Kubernetes Enhancement Proposal
PaaS	Platform-as-a-Service
QoS	Quality of Service
SLA	Service Level Agreement
SLI	Service Level Indicator
SLO	Service Level Objective
SSH	Secure Shell
VM	Virtual Machine
VPA	Vertical Pod Autoscaler (Kubernetes)
YAML	YAML Ain't Markup Language

1 Introduction

"A distributed system is one where the failure of a computer you didn't even know existed can render your own computer unusable."
— Leslie Lamport

Cloud computing allows users to access compute, storage and network resources on-demand over the Internet. Resources can be allocated and released whenever needed, thus *elasticity* is one of the most prominent features of cloud computing [1]. Around the same time that this computing paradigm became widely used, the *microservices* software architecture also became popular. In fact, these two trends are correlated, since cloud infrastructure facilitates the development of distributed microservice architectures [2].

To take full advantage of the elasticity in cloud computing, *autoscaling* (sometimes also referred to as *adaptive scaling*) needs to be implemented. It is a technique to automatically scale the application (and the services it is composed of) based on the current demand. In general, *scaling* refers to acquiring and releasing resources while maintaining a certain application performance level, such as response time or throughput [3]. Scaling an application can be achieved in two ways: *horizontal scaling* and *vertical scaling*. Horizontal scaling, also referred to as *scaling out*, refers to creating more instances of the same service. The workload is then distributed across all instances, resulting in a lower workload per instance (*load balancing*). An example here is adjusting the number of web servers according to the amount of incoming website requests. Vertical scaling, also referred to as *scaling up*, refers to giving more resources (compute, memory, network, storage) to a particular instance of the service. By giving more resources to one or multiple instances, they are able to handle more workload. An example for this is providing more memory resources to a database instance: this commonly results in faster response times, because the database can fit more data in memory instead of having to load it from disk.

Due to the higher cost of cloud infrastructure compared to on-premise infrastructure, it is vital to take advantage of its elasticity and implement autoscaling. This autoscaling needs to provision and release cloud resources without human intervention. Overprovisioning leads to paying for unused resources, while underprovisioning causes the application performance to degrade [4]. The scaling logic needs to balance between these two goals: minimizing resource usage (and thereby cost) with an acceptable service quality and minimizing service-level agreement violations by provisioning sufficient amount of resources.

To reliably and consistently achieve this in the first place, extensive monitoring of low- and high-level metrics needs to be set up. These metrics are not only used as inputs for a scaling policy, but also to ensure that the system is not spiraling out of control (e.g., erroneously requesting more and more compute resources, thereby incurring large bills or starving other services for resources). Even then, efficiently operating and scaling a complex mesh of microservices is a task with many challenges [5].

1.1 Contribution and Outline

This thesis investigates how applications running on top of Kubernetes can be dynamically scaled, thereby allowing to take full advantage of the elasticity of cloud computing. In particular, we focus on the prerequisites (metrics-based monitoring) and the challenges of autoscaling (identifying the right metrics and eliminating bottlenecks).

The contributions of this thesis are the following:

- a thorough discussion of Kubernetes concepts and components relevant for autoscaling;
- an overview of generic autoscaling literature and a qualitative comparison of research proposals for Kubernetes autoscalers;
- a proposal for a novel, modular Kubernetes autoscaler with a WebAssembly sandbox;
- the implementation of an extensive monitoring solution for a production-grade application running on Kubernetes (with Grafana, Prometheus and several metric exporters);
- a discussion of which types of metrics are suitable for scaling and how metrics can be used to get a holistic view of application performance;
- the implementation and fine-tuning of autoscaling policies for the target application (with HPA and KEDA);
- a quantitative evaluation of several autoscaling policies according to performance and cost criteria.

The rest of this thesis is organized as follows. Chapter 2 elaborates on the joint rise of cloud computing, containers and microservices, and how Kubernetes unifies these three concepts. This is followed by an introduction of Kubernetes' architecture and components relevant to scaling. Chapter 3 outlines state-of-the-art autoscaling components for Kubernetes. It also presents and evaluates recent research about Kubernetes autoscalers, and presents a solution to close the gap between academia and industry on this topic. Chapter 4 documents the concrete setup of metrics-based monitoring for an application running on Kubernetes and the associated autoscaling infrastructure. In Chapter 5, this infrastructure is used to conduct quantitative experiments about the behavior, performance and cost of different autoscaling policies. It also discusses and validates several policy optimizations. Finally, Chapter 6 provides concluding remarks and outlines future work.

2 Background

> "*Cloud-native is a term describing software designed to run and scale reliably and predictably on top of potentially unreliable cloud-based infrastructure.*"
> — Duncan Winn

This chapter examines the history of the trends towards cloud computing and microservices and highlights the connection to Kubernetes. Afterwards, essential Kubernetes concepts are presented, as they are required for understanding the effects of the changes and configuration settings in the following chapters.

2.1 Cloud Computing and Containers

Cloud computing offers users instant access to a wide variety of processing and storage services, all of which can be accessed over the Internet. One of the most prominent features of "the cloud" is full elasticity of computing resources [1]. In fact, cloud resources can be dynamically requested and adjusted. Unlike buying a fixed amount of physical servers, only the requested resources need to be paid for.

Originally, the term elasticity has been used in physics to refer to the property of material that is capable of returning to its original state after deformation. In economics, elasticity describes the effect that changing one variable has on another variable. More recently, the concept of elasticity has been applied to the context of cloud computing. However, unlike in physics and economics, elasticity in cloud computing does not convey a quantitative meaning and is rarely rigidly defined. More often, it is simply used as buzzword on the frontpage of public cloud providers' websites. The SPEC Research Group has produced a detailed definition of elasticity in the context of cloud computing: it is the ability to scale up and down the number of resources allocated for an application [6].

Cloud computing is the latest incarnation of a continuous trend to make computing resources more flexible, configurable and efficient. The introduction of multi-tasking (*time-sharing*) operating systems allowed multiple users to simultaneously access the same physical hardware. This trend is also reflected in the shift from specialized servers (*mainframes*) to using commodity hardware at a large scale. Virtualization even abstracts away this commodity hardware into a *computing substrate*: an abstract platform for performing computations. Finally, containers further reduce dependency on the underlying operating system and execution environment. System administrators no longer need to decide on which specific machine an application runs or worry if enough resources are available.

The "modern" *container* is an implementation specific to the Linux kernel, but other operating systems have similar concepts. The idea of restricting the access of a particular process goes back to 1982, when *chroot* environments in Unix allowed restricting the accessible filesystem for a particular process to a specific directory. In 2000, FreeBSD introduced *Jails*[1], which builds on the chroot concept and provides

[1] https://docs.freebsd.org/en/books/handbook/jails/

additional isolation and security guarantees (e.g., separate namespaces for process IDs). In 2002, Oracle introduced *Zones*, also known as *Solaris Containers*, as a first-class concept in the Solaris operating system [7]. In 2005, *OpenVZ* was the first operating-system-level virtualization for Linux, but required a modified Linux kernel. By 2008, the Linux kernel natively supported enough features to host *Linux Containers* (*LXC*). Containers allow limiting the resources (CPU, memory, filesystem, network etc.) available to a process – or set of processes – through a Linux kernel feature called *cgroups* (control groups). Like regular processes in an operating system, containers share the same kernel with all other processes on the host. Unlike regular processes, each container only sees and has access to its own, separate environment, which is achieved through namespace isolation. By combining both resource restriction and namespace isolation containers implement *operating-system-level virtualization*. In contrast to full virtualization, where multiple kernels are running on the same host, containers have a lower resource footprint (CPU, memory and storage utilization), thereby enabling higher application performance [8]. Subsequently, this allows a higher density of applications per host and more efficient resource usage by colocating different types of applications [7]. Since the size of container images tends to be an order of magnitude smaller compared to virtual machine disk images [9], they can easily and efficiently be shared online through container image registries. Finally, containers can be started within seconds, as opposed virtual machines which can take minutes to initialize. This allows frequently adding and removing container instances without much overhead, thereby improving elasticity [10].

The Docker project introduced a tool that can manage the entire life cycle of a container: building an image from a set of instructions (*Dockerfile*); sharing this container image over the internet (*DockerHub*); as well as creating, running and deleting containers based on images. This is what is commonly understood when referring to the modern container [7]. All these features mean that containers can not only be used to run applications and their components, but also to package them up in a convenient format alongside their configuration. Thus, containers provide a higher level of abstraction for the application lifecycle, including not only starting and stopping, but facilitating also upgrades and replication in a seamless way [11].

Since Docker's introduction in 2013, the *containerization* of applications has seen widespread adoption. The monitoring company *Datadog* found in their 2018 report that 23.4% of their customers had adopted Docker, both at small (single developers and small startups) and large scales (enterprise software development) [12].

2.2 Microservices

Coincidentally, containers provide a flexible abstraction for composing a collection of microservices, which is a software architecture that has increased in popularity over the last ten years [2]. With the microservice architecture, a single application is decoupled into multiple, distributed services. Each service follows the *Single Responsibility Principle* by providing independent functionality and communicates with other services via language-agnostic *Application Programming Interfaces* (APIs).

The main advantage of microservices is organizational: each service can be developed and operated by a different development team, and therefore each team can make independent organizational decisions (such as software releases) as well as technological decisions (programming languages, frameworks etc.) [13]. As a result of the shift towards microservices, the backend architecture of many applications has seen an increase in complexity as well. Already in 2001, IBM has pointed out that the main obstacle in the IT industry is the growing software complexity [14].

When an application is made up of many different microservices, these services should be distributed across multiple machines for two reasons: fault tolerance and performance. A greater degree of fault tolerance enables high availability. If all services of an application are running on the same machine and that machine encounters a hardware fault (such as power loss), the entire application will be offline (Figure 1b). In the case of loosely coupled microservices distributed across multiple machines, a failure of a single machine only partially affects the availability of the application (Figure 1c). At the same time, distributing the individual services across multiple machines increases the performance of the overall application as it is no longer constrained by the available CPU, memory or storage resources of an individual machine. Additionally, each service can be scaled with fine granularity, which reduces the cost compared to the conventional replication of the entire application. However, such an architecture comes at the cost of increased software complexity, larger amounts of network traffic and susceptibility to network outages.

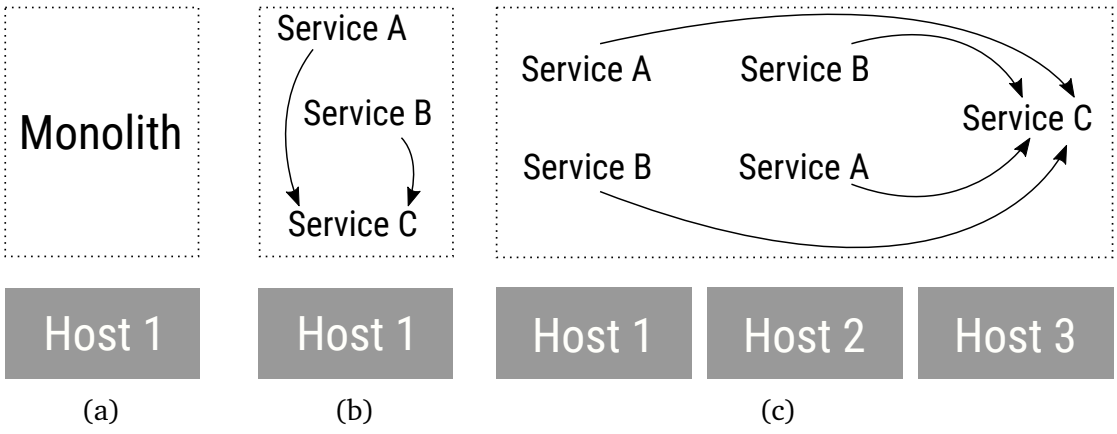

Figure 1 – Comparison of monolithic, local and distributed microservice architectures

Additionally, there is another major challenge: increased operational complexity. While all the services may be simple to install and run with container tools such as Docker, system administrators need to operate many of these applications across a large number of machines (dozens, hundreds or even thousands). These and related tasks are commonly referred to as *orchestration*. More specifically, orchestration is the management of (virtual) infrastructure required by an application during its entire lifecycle: deploying, provisioning, running, adjusting, terminating. This is part of the vision of *autonomic computing* introduced in 2001 [14]: software systems that can manage themselves.

This is exactly where Kubernetes comes in: its main goal is making the orchestration of complex distributed systems easy while leveraging the density improvements offered by containers [7].

2.3 Kubernetes

Kubernetes is an open-source framework for automating the deployment, scaling and management of distributed applications in the context of a cluster. A *cluster* is a set of worker nodes which are orchestrated by one control plane and appear as a single unit to the outside. Kubernetes' initial design was based on Google's internal *Borg* and *Omega* systems, both cluster management systems that the company uses to schedule workloads across machines in its datacenters [7]. Kubernetes was introduced in 2015 and the name is commonly abbreviated as *K8s*. Since then, it has become a widely used platform for deploying distributed applications. This is made apparent by the fact that all major public cloud platforms offer a managed Kubernetes service (AWS EKS, GCP GKE, Azure AKS, IBM Cloud Kubernetes, AlibabaCloud ACK). Unless otherwise noted, all statements in this thesis regarding Kubernetes refer to version 1.20 (released in December 2020).

Kubernetes itself is implemented as an application with a microservice architecture. This means the individual parts – which are referred to as *components* – are loosely coupled and have distinct functional responsibilities. Each component provides a set of services to the other components through well-defined APIs[2]. This allows the Kubernetes architecture to be open and extensible, which is one of the explicit development goals.

Using a microservice architecture for Kubernetes makes sense for two reasons: the software managing other applications needs to be highly available (fault tolerancy); and the software is developed in a distributed fashion by many *special interest groups*[3] (SIGs). Furthermore, it enables anyone to enhance or replace every single one of the components individually without having to modify the rest of the system. Finally, many of the components are optional and thus do not necessarily need to be used in every environment. One example of the extensibility is the Crossplane project[4], which exposes resources outside of a Kubernetes cluster (such as databases or virtual machines) through the Kubernetes API. The result of this extensibility is that Kubernetes itself is a complex mesh of microservices. It is absolutely necessary to have a firm understanding of its components to be able to effectively operate and optimize it.

Kubernetes refers to individual machines as *nodes* and to a set of nodes controlled by the same Kubernetes instance as a *cluster*. To the user, Kubernetes presents a declarative interface for describing the state of *objects* in the cluster. The most common *core objects* (supported by default) are Pods and Services. As described before, the list of objects is extensible through Kubernetes' microservice architecture.

[2] https://github.com/kubernetes/community/blob/master/contributors/devel/sig-architecture/api-conventions.md

[3] https://sigs.k8s.io

[4] https://crossplane.io/

Declarative means that Kubernetes continuously tries to the converge the current state of the objects towards the desired state, which is defined by a *specification* (or *Spec*) in Kubernetes. Practically speaking, when the user specifies that 5 instances of a web server should be running, Kubernetes creates 5 instances and monitors that they are available. When one of them is no longer available, for example due to a software bug or hardware failure, Kubernetes automatically creates a new instance.

2.3.1 Kubernetes Objects

In Kubernetes, a *Pod* – not a single container – is the smallest deployment unit [15]. A Pod can comprise one or more containers. It has an associated configuration (the *PodSpec*) that determines how exactly the container(s) are run, including attached compute, network and storage resources. All containers within the same Pod share these resources. Furthermore, Kubernetes supports two types of resource declarations: *requests* and *limits*. *Resource requests* define the minimum value of a given compute resource that has to be guaranteed to the Pod. Requests are used by the scheduler to decide on which worker node to place the Pod. *Limits* define the maximum amount of resources available to the Pod. By default, the supported resources are CPU and memory resources [15]. Resource reservations and limits for other metrics (e.g., network or disk usage) can be added by installing third-party extensions — these define so-called *extended resources*[5].

Kubernetes uses *ReplicaSets* to manage the number of concurrently running instances of the same Pod (replicas). A *Deployment* is a higher-level concept that manages the ephemeral ReplicaSets and Pods (Figure 2). It provides many useful features to describe the desired state of an application [15]. For example, if a node fails (hardware fault, power outage etc.), Kubernetes does not "recreate" individual Pods previously running on this node. However, when Kubernetes detects that a Pod which is part of a Deployment is unavailable, it creates a new instance of the same Pod type (a behavior referred to as *self-healing*). Therefore, Deployments provide a declarative orchestration interface for applications running on Kubernetes.

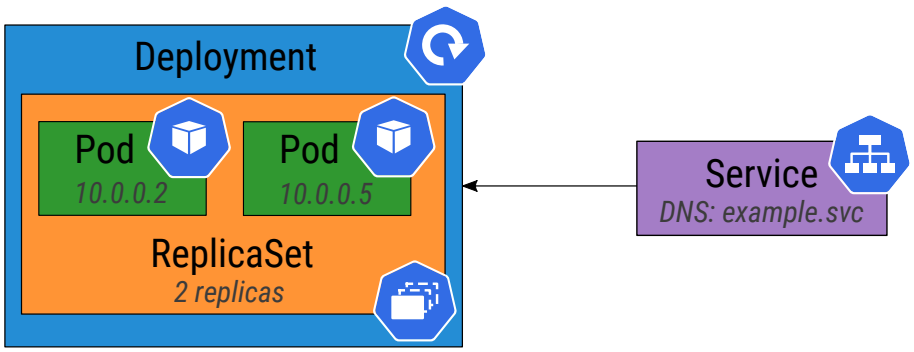

Figure 2 – Overview of built-in Kubernetes objects and their relations

A *Service* is an abstraction that gives a distinct network identity to an application running in one or more Pods (Figure 2). This is necessary because Pods are

[5] https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/

ephemeral, meaning that they can be created or destroyed all the time – alongside their associated IP addresses. When an application communicates with another one through a *Service*, Kubernetes automatically forwards all requests to the *Service* to the associated *Pods*. Thus, a Service provides a mechanism for *service discovery* (each Pod has a unique IP address, but Pods can be created and destroyed) as well as *load balancing* (by default Kubernetes uses a round-robin algorithm distribute requests across all Pods associated to a Service). An example of a Service definition is shown in Appendix A.3. A Service can also describe an entity outside the Kubernetes cluster, such as an external load balancer [16].

A *StatefulSet* is the equivalent of a Deployment but tailored for applications that require guarantees about the ordering and uniqueness of the application instances. In particular, a StatefulSet offers stable, unique IP addresses; unique, stable Pod names; and ordered, graceful deployments [15]. These constraints are vital for the correct and efficient operation of many stateful applications, such as databases or message queues.

This section only covered the most important Kubernetes objects relevant to the work in this thesis. An extended discussion of all objects can be found in [17].

2.3.2 Kubernetes Components

The *control plane* is the layer of components that exposes the API and interfaces to define, deploy and manage the lifecycle of Kubernetes objects [18]. These components are drawn blue in Figure 3. The *data plane* is the layer that provides compute capacity (such as CPU, memory, network and storage resources) where objects can be scheduled. Such capacities are made available through the *kubelet* running on each worker node: the daemon is responsible for the communication with the control plane. The kubelet continuously gathers facts about its host and the workloads running on it (e.g., CPU, memory, filesystem, and network usage statistics), and sends them to the control plane. These statistics are collected with *cAdvisor*[6], which is an tool for container resource usage and performance analysis.

Based on the information provided by the worker nodes, the *scheduler* decides which workloads (i.e., Pods) will be placed on the worker node, subject to predefined constraints and runtime statistics. The default scheduling policy is to place Pods on nodes with the most free resources, while distributing Pods from the same Deployment across different nodes. In this way the scheduler tries to balance out resource utilization of the worker nodes [18].

Then, the kubelet on the corresponding worker node receives the scheduling decision in the form of *PodSpecs* (Pod specifications). It implements the received instructions by launching and monitoring the containers through the *container runtime* (e.g., containerd[7] or cri-o[8]). The kubelet also manages the lifecycle of other host-specific resources, such as storage volumes and network adapters [18].

[6] https://github.com/google/cadvisor
[7] https://containerd.io/
[8] https://cri-o.io/

The *controller manager* (CM) implements the core functions of Kubernetes with control loops (such as replication, endpoints and namespace controller). A control loop is a non-terminating loop that regulates the state of a system. It is commonly found in industrial control systems and robotics. The controller manager monitors the state of the cluster (including all its objects) and tries to converge the state of the system towards the desired state, thereby implementing Kubernetes' declarative nature [18].

The user can interact with the control plane through a well-defined API. The *kubectl* command line tool allows the user to retrieve and define the state of the cluster in a human-friendly manner. Automation tools (e.g., Terraform) can also directly interact with the Kubernetes API for the same purpose.

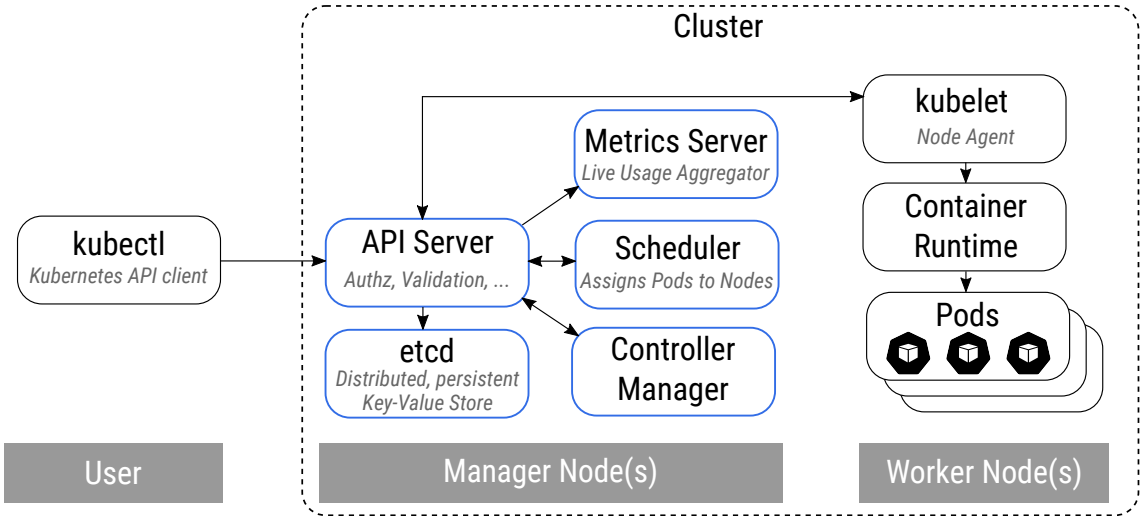

Figure 3 – Architecture of Kubernetes (components in blue are part of control plane)

The *API server* provides the entrypoint to the control plane for internal and external components (Figure 3). Thus, it performs authentication, authorization, versioning and semantic validation functions. It is responsible for communicating with all the other control plane components and is the only component which has direct access to the database [18]. Kubernetes uses the strongly-consistent, distributed key-value store *etcd* as a shared database for the control plane.

The *Metrics Server* is an efficient and scalable cluster-wide aggregator of live utilization statistics. It tracks CPU and memory usage across all worker nodes, as reported by the kubelet's cAdvisor (Figure 3). The Metrics Server implements the Metrics API[9] and is the successor of the deprecated Heapster[10]. Other implementations (e.g., Prometheus) and adapters can be used as an alternative source for the Metrics API. Notably, the Metrics API does not offer access to any historical usage statistics.

There are several other components in the Kubernetes control plane which are not shown in Figure 3, as they are not relevant to the work in this thesis.

[9] https://github.com/kubernetes/metrics
[10] https://github.com/kubernetes-retired/heapster

3 Autoscaling

> "*Theoretically-obtainable efficiencies are often hard to achieve in practice because the effort or risk required to do so manually is too high. What we need is an automated way of making the trade-off.*"
> — Rzadca et al. [19]

This chapter first introduces the autoscaling concept from first principles and provides an overview of relevant cloud autoscaling literature. Next, publicly available autoscaling components for Kubernetes are presented. Finally, we survey research proposals for novel Kubernetes autoscalers and perform a qualitative evaluation.

Elasticity is the ability of a system to increase or decrease the allocated resources on demand [1]. *Autoscaling* (sometimes also referred to as *adaptive scaling*) is the process of adjusting the amount of available resources to the current demand. It has been extensively researched over the last decades [20, 21], also outside of the realm of computer science [22]. In particular the widespread adoption of the cloud computing paradigm has accelerated the pace of development and research in this field.

With the proliferation of container orchestration frameworks over the last five years, the topic of container autoscaling has seen particular attention. In this chapter we specifically focus on autoscaling techniques for the Kubernetes framework. It has seen the largest adoption in industry as well as academia in recent years. A large ecosystem of open-source technologies, startups and business models has evolved around it, making it very likely to remain popular in the future. Other orchestration frameworks are either rarely used (like Mesosphere Marathon) or gradually being phased out by their developers (like Docker Swarm)[11].

3.1 Autoscaling in the Cloud

The following section gives an overview of research on the topic of autoscaling and elasticity of cloud infrastructure and applications.

In general, autoscaling approaches are based on the broadly recognized *MAPE-K* control loop. MAPE-K stands for *Monitor*, *Analyse*, *Plan* and *Execute* over a *shared Knowledge base* [14]. It is an instance of a feedback loop widely used for building self-adaptive software systems [4]. In the monitor phase the execution environment is observed. The observed data (i.e., metrics) is then used in the analyse phase, which determines whether any adaptation of the system is required. In the plan phase the appropriate actions to adapt the system are evaluated and a final selection is made, considering feasible adaptation strategies such as scaling up or down. In the execute phase the system is changed to match the new desired state proposed in the plan phase. The knowledge base is used as a central store of information about the entire execution environment (i.e., a database) [13].

[11] https://thenewstack.io/mirantis-acquires-docker-enterprise/

In 2014, Lorido-Botran et al. [23] published a comprehensive survey on resource estimation techniques for scaling application infrastructure. Their survey is one of the seminal works in this field. It gives an overview of 50 research proposals and classifies them into five categories for autoscaling approaches: threshold-based rules, control theory, reinforcement learning, queuing theory and time-series analysis. We go into the details of these categories later. At that time, containers were not widely used and therefore all of the autoscaling proposals were focused on virtual machines. Furthermore, the landscape of cloud-native applications and infrastructure was still in its infancy, which complicated a direct comparison between different approaches.

In 2015, Galante et al. [24] published a survey on the use of elastic cloud computing for scientific applications. Their work describes several approaches, advantages and drawbacks of running scientific computations in the cloud. Their presentation of elasticity mechanisms of public cloud providers focused on IaaS (*Infrastructure-as-a-Service*) and PaaS (*Platform-as-a-Service*) solutions. They found that most traditional scientific applications are executed in batch mode and have difficulty adapting to dynamic changes in the infrastructure (e.g., addition or removal of one or more worker nodes). Among other challenges, they found that the elasticity mechanisms lacked good support for scientific batch workloads, as they were more focused on server-based applications (web servers etc.). They also recognized the lack of cloud interoperability.

In 2016, Hummaida et al. [25] published a survey that focused on efficient resource reconfiguration in the cloud from the perspective of the infrastructure provider, instead of the user. In this context, they defined *Cloud Systems Adaptation* as "*a change to provider revenue, data centre power consumption, capacity or end-user experience where decision making resulted in a reconfiguration of compute, network or storage resources*." [25]

Another large scale survey on the challenges of autoscaling was published by Qu et al. in 2018 [4]. Additionally, they provided a detailed taxonomy of cloud application autoscaling, which we apply in our work. Also their work focused only on autoscaling proposals for virtual machine based infrastructure, however many of the concepts applied to VMs can also be applied to containers, sometimes even in better ways.

As well in 2018, Al-Dhuraibi et al. [26] published a similar survey of state-of-the-art techniques and research challenges in autoscaling. Their work was the first survey that included both VM- as well as container-based solutions. While their survey included relevant container orchestration tools at the time, none of the research proposals for autoscaling were developed for Kubernetes. Furthermore, their survey also presented a taxonomy for cloud application elasticity, which we partially base our classification on.

More recently, Radhika and Sadasivam [27] provided a study on proactive autoscaling techniques for heterogeneous applications. None of the proposals evaluated in the work was developed for Kubernetes.

3.2 Built-in Kubernetes Autoscalers

The Kubernetes authors have developed three components that enable elastic scaling of clusters: *Horizontal Pod Autoscaler* (HPA), *Vertical Pod Autoscaler* (VPA) and *Cluster Autoscaler* (CA). *Kubernetes Event-driven Autoscaler* (KEDA) is a third-party component for Kubernetes that builds on top of HPA. The purpose and operation of these autoscalers are described in the following sections.

3.2.1 Horizontal Pod Autoscaler

Kubernetes' *Horizontal Pod Autoscaler* (HPA) is one of the control loops integrated in the Controller Manager (Section 2.3.2). It allows dynamically adjusting the number of Pod replicas for a particular Deployment by consuming the Metrics API [28].

By default, HPA scales Pods based on relative CPU utilization. However, memory utilization, custom and external metrics are also supported. CPU and memory utilization on the worker nodes are collected with cAdvisor. Custom or external metrics can be utilized by querying a custom metrics API. The kube-metrics-adapter project[12] offers adapters to convert metrics from popular third-party metric services (such as Prometheus) into a suitable format for HPA.

The core algorithm of HPA is shown in Equation 1 and 2. The current metric value m_i is retrieved for all active replicas in the set R and the mean value $\bar{m}$ is calculated. The target number of replicas $\hat{r}$ is calculated by dividing the mean usage m by the desired usage $\hat{m}$, multiplying the result with the current number of replicas r and finally rounding up [28].

$$\bar{m} = \frac{\sum_{i \in R} m_i}{R} \tag{1}$$

$$\hat{r} = \left\lceil r * \frac{\bar{m}}{\hat{m}} \right\rceil \tag{2}$$

As an example, let us assume that the target memory utilization is set to 50%, the Pod memory limit is set to 1 GB and there are currently three Pod replicas which utilize 600 MB, 700 MB and 800 MB, respectively. In this case, the mean utilization of all Pods is 70%, which is 1.4 times above the target utilization. Multiplying this number with the current number of replicas yields 4.2, which is rounded up to 5. Thus, the HPA indicates to the Kubernetes control plane that two more replicas are needed in order to match the target resource utilization. Of course, a single adjustment might not be enough (due to non-linear scaling of applications and varying load), therefore the HPA runs as a control loop with a default interval of 15 seconds (*sync-period*) [28].

In general, the algorithm has a bias towards scaling up faster and scaling down slower. For example, to avoid *thrashing* (oscillations in the number of Pods) each newly created replica runs for at least one downscale period (by default 5 minutes) before it can be removed again. The HPA supports several other settings: cluster-wide

[12] https://github.com/zalando-incubator/kube-metrics-adapter

settings (e.g., scaling stabilization, scaling tolerance, Pod synchronization period) as well as Deployment-wide settings (e.g., specifying a minimum or maximum number of Pod replicas) [28]. The authors of [29] have developed a formal, discrete-time queuing model of HPA's algorithm, which gives an approximation of the number of Pods deployed by the autoscaler.

Scaling decisions taken by HPA are not immediately reflected in the status of the cluster, but first need to propagate through several control plane components (Section 2.3.2). In particular, Kubernetes needs to perform the following steps before a new Pod is available for handling workloads:

1. The HPA control loop is activated to calculate the new desired number of replicas. The result is saved in the ReplicaSet object.
2. The ReplicaSet controller is activated to pick up the changes in the ReplicaSet. It creates a new Pod object to fulfill the requirement.
3. The scheduler control loop detects that there is a Pod without an assigned node. It selects an appropriate worker node for the new Pod, while taking into account the scheduling policy and cluster status. The kubelet on the selected node is notified about the pending Pod.
4. The kubelet on the worker node initiates the Pod creation process. This includes downloading container images from the registry and unpacking them, launching and initializing containers associated to the Pod and waiting until it becomes *ready* (indicated through a *Readiness Probe*).

3.2.2 Vertical Pod Autoscaler

The *Vertical Pod Autoscaler* (VPA) automatically sets resource requests and limits of a Pod based on historical usage. Thus it allows each Pod to have the necessary resources available while minimizing excess resources (so-called *slack*). It can both down-scale Pods when the initially requested resources were too high as well as up-scale Pods when their usage indicates that they under-requested resources [30]. This means that the resources in the cluster are used and shared efficiently, because each Pod is adjusted to the amount it currently requires, instead of operating with static thresholds. Additionally, this improves Pod scheduling on nodes. VPA can either use the Metrics Server or a Prometheus instance to obtain time-series metrics. Unlike HPA, VPA only supports scaling based on CPU and memory metrics, but not custom (external) metrics. This component is not part of a default Kubernetes installation and the following discussion applies to VPA version 0.9.

The VPA itself is implemented with a microservice architecture consisting of three separate components (Figure 4):

- The *Recommender* monitors the current and past resource consumption of containers and provides recommended values for CPU and memory requests.
- The *Updater* checks which of the managed Pods have correct resources set and, if the active resources diverge more than 10% from the recommendation, it forwards the recommended values to the Admission Plugin. If the state of

the cluster allows, it also evicts Pods (terminates running instances) in order for the new resource values to take effect.
- The *Admission Plugin* intercepts Pod creation requests to set the resource values given by the Updater.

This highlights a major drawback of the current VPA implementation: its operation is disruptive. In fact, resource adjustments are carried out by terminating the Pod and then re-scheduling it with the newly estimated resources. This approach works for stateless services (though they may still experience service disruption), but it is a major impediment for stateful and high-performance applications [31]. As of writing, the Kubernetes authors are working on in-place updates of Pods, which would allow the VPA to operate non-disruptively.[13]

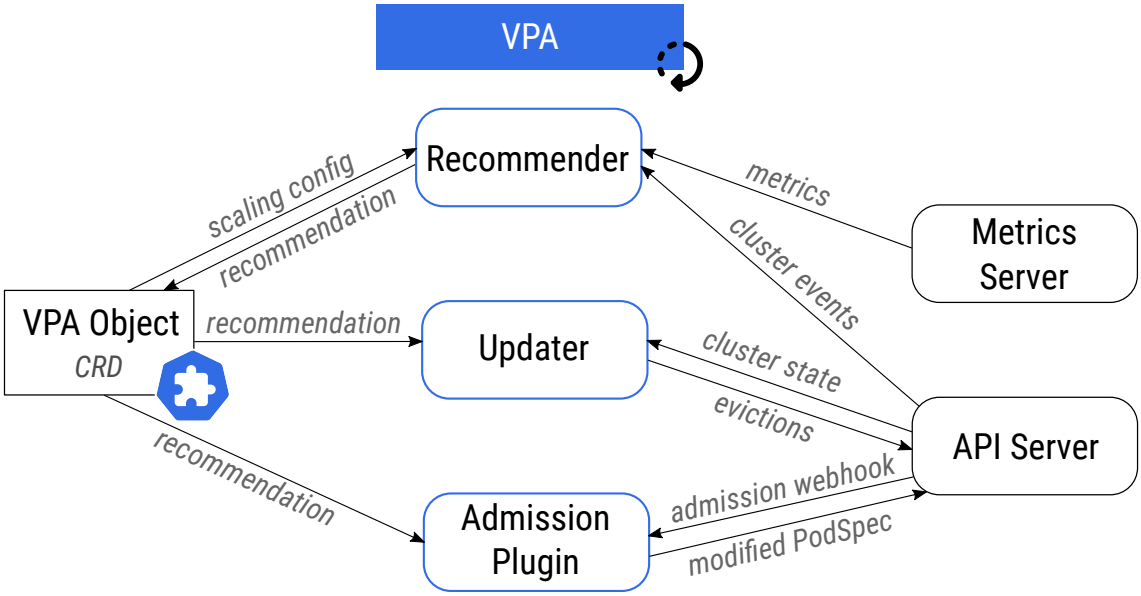

Figure 4 – VPA Architecture

Compared to the HPA, the VPA's algorithms are more complex as they also take into account historic usage and cluster events such as out-of-memory errors. Since the Metrics API only offers access to live usage data, but not historical metrics, the VPA needs to built its own internal model for historical metric values [30]. While this kind of complexity is usually undesired by infrastructure operators, this scaling logic has been used successfully in Google's Borg infrastructure.

For CPU resources, VPA collects historic CPU utilization and samples it into buckets with exponentially growing boundaries, from 0.001 cores up to 1,000 CPU cores with an exponential growing rate of 5%. For memory resources, VPA collects the memory usage peaks of each 24 hour interval during the last 8 days. These 8 samples are put into buckets with exponentially growing boundaries of 5% (e.g., 10-11.2MB, 11.2-12.66MB, ...). Both types of samples then get a decaying weight

[13] https://github.com/kubernetes/enhancements/tree/master/keps/sig-node/1287-in-place-update-pod-resources

with a half-life of 24 hours, meaning that the most recent samples get a weight of (slightly below) 1, 24 hour old samples a weight of 0.5 and so on.

Then, three values are calculated (Equation 3): the lower boundary b_l (50th percentile of historic usage H), the target value t (90th percentile of H) and the upper boundary b_u (95th percentile of H). Exemplary, the 90th percentile describes the boundary below which the resource utilization is for 90% of the time. Each of these bounds is then scaled with a safety margin m of 15% [30].

The target value t is the recommended *resource request* for the Pod. The *resource limit* is either scaled proportionally to the initial ratio between request and limit or set to a specific maximum. For example, when the initial request is 100MB with a limit of 200MB, and VPA recommends the request to be 175MB, then the proportionally scaled limit will be 350MB (unless a LimitRange[14] is specified).

Finally, the calculated upper and lower bounds are multiplied with a confidence interval c which is based on the amount of collected samples (number of days of historic data), i.e., with more historic data the confidence of the estimations is higher (Equation 3).

The lower bound estimate e_l means that any value below this bound is likely not enough for the application. The upper bound estimate e_u means that any value above this bound is likely to be wasteful. These two estimates are used by the Updater to decide if a Pod should be evicted. In this sense, these bounds act as a proxy for determining if it is worth adjusting the resources of a Pod, which avoids overly frequent changes and thereby thrashing [30].

$$\begin{aligned} m &= 15\% \\ b_l &= P_{50}(H) * m \\ t &= P_{90}(H) * m \\ b_u &= P_{95}(H) * m \\ c &= 1 + \frac{1}{days(H)} \\ e_l &= b_l * c^{-2} \\ e_u &= b_u * c \end{aligned} \tag{3}$$

3.2.3 Cluster Autoscaler

The *Cluster Autoscaler*[15] (CA) is the third component that enables scaling on Kubernetes. Specifically, it adjusts the size of the cluster in terms of the number of worker nodes. It can either add nodes when the compute resources in the current cluster are insufficient, or remove nodes when there are unutilized nodes. This is achieved through integration with the APIs for provisioning and deprovisioning virtual machines of several public cloud platforms.

The CA adds new nodes to the cluster when there are Pods in *unschedulable* state, meaning the scheduler was unable to assign a node to the Pod due to insufficient

[14] https://kubernetes.io/docs/concepts/policy/limit-range/
[15] https://github.com/kubernetes/autoscaler/tree/master/cluster-autoscaler

resources. This can happen for several reasons: a new application is deployed to the cluster, the HPA increases the number of replicas of a Deployment or the VPA increases the resource requests for a workload. Conversely, the CA decreases the size of the cluster when some nodes are consistently unneeded for a significant amount of time, i.e., it has low utilization and all of its Pods can be scheduled on other nodes. All of these decisions are made subject to several constraints the cluster administrator can set to prevent the CA from affecting the functionality of the cluster (e.g., eviction policy or Pod disruption budget).

While the interaction between the two Pod autoscalers (HPA and VPA) and the CA is crucial for successfully operating an elastic Kubernetes cluster, the CA is not relevant to the work carried out in this thesis. Thus, a technical study of its internal mechanism is omitted here.

3.2.4 Kubernetes Event-driven Autoscaler

While Kubernetes' HPA can easily be configured to use CPU and memory utilization, scaling based on custom metrics requires several other components: metrics need to be exposed; an external monitoring tool needs to be provisioned and configured; metrics need to be fed into Kubernetes' Metrics API; and finally, those metrics can be used for autoscaling with HPA (these steps are detailed in Chapter 4).

The *Kubernetes Event-driven Autoscaler* (KEDA) [32] addresses this complexity by taking a different approach: instead of reactively scaling based on metrics from a monitoring system, it observes events as they are happening at the source. KEDA supports a wide range of event sources (including message queues, databases, streaming services and monitoring solutions)[16], which can be cluster-internal and cluster-external. With KEDA the end user only needs to configure a simple *ScaledObject* CRD (discussed in Section 4.3.3).

Figure 5 shows KEDA's two components: *agent* and *metrics API server* [32]. The agent is responsible for scaling a Deployment between zero and one replicas. This is a workaround for the limitation that HPA does not scale Deployments to less than one replica. The metrics API server is responsible for listening to the event source and exposing new events through Kubernetes' metrics API (this component serves the same purpose as the metrics adapter we discuss in Section 4.2).

Figure 5 shows the autoscaling operation of KEDA. When idle (i.e., no incoming events) KEDA scales the Deployment to zero replicas. As soon as events are detected, KEDA scales the Deployment to one replica. Further scaling (depending on the number or rate of incoming events) is then carried out by the HPA, which consumes the metrics exposed by KEDA's metrics server [32].

Unlike other event-driven and serverless frameworks (e.g., Knative[17]), KEDA is a single, lightweight component that focuses exclusively on autoscaling. Because Deployments need to be explicitly marked as managed by KEDA, it is a flexible solution safe to run alongside existing Kubernetes applications without major modifications.

[16] https://keda.sh/docs/2.2/scalers/
[17] https://knative.dev/

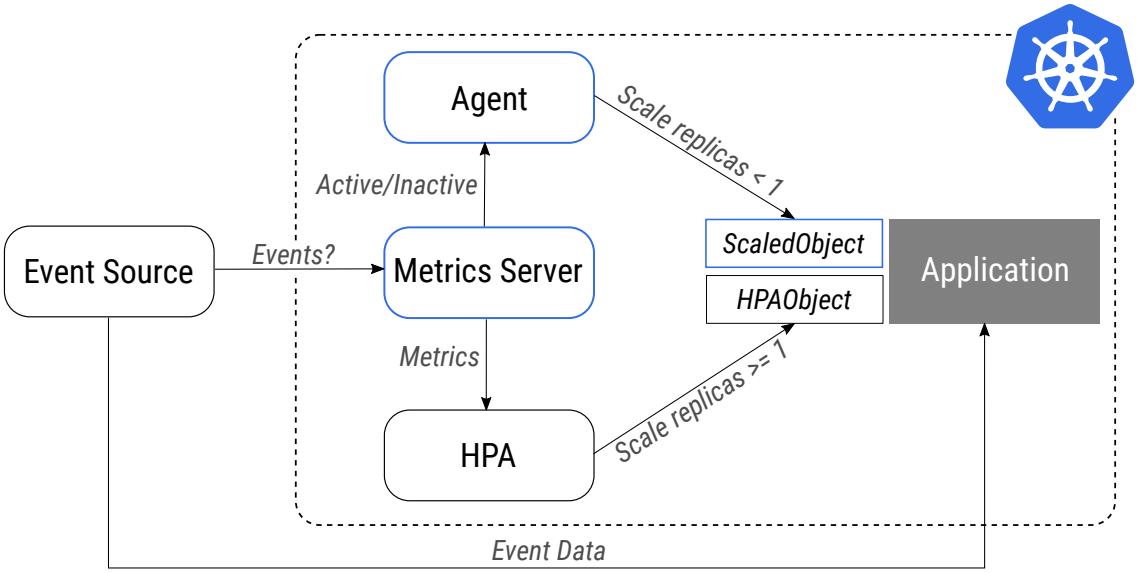

Figure 5 – Overview of KEDA autoscaling. KEDA components marked in blue.

3.3 Research proposals for Kubernetes Autoscalers

While autoscaling has been adequately considered in the literature, the following survey provides an overview and discussion of proposals for novel Kubernetes autoscalers. The survey considers only cluster-internal scaling mechanisms (i.e., vertical and horizontal scaling of Pods), external cluster scaling is outside the scope of this study (e.g., [33, 34]). This choice was made because the nature of autoscaling decisions between these dimensions is quite different. The same reasoning applies to scheduling algorithms. While there have been interesting proposals for improved Kubernetes schedulers [35, 36, 37], scaling and scheduling are two fundamentally different operations. Nevertheless, there are certainly advances to be made by having some amount of coordination between scaling and scheduling.

Table 1 – Comparison of Kubernetes autoscaler proposals

Reference	Architecture	Technique	Approach	Scaling Timing	Metric	Workload Pattern
KHPA-A [38]	single	threshold-based	reactive	horizontal	CPU	unspecified
HPA+ [29]	single	forecast-based	proactive	horizontal	no. of timeouts	predictable burst
Libra [39]	single	threshold-based	reactive	horizontal & vertical	CPU & throughput	unspecified
Microscaler [5]	single	threshold-based	reactive	horizontal	response time	unspecified
Q-Threshold [40]	single	RL-based	reactive	horizontal	response time	predictable burst
RUBAS [31]	single	threshold-based	reactive	vertical	CPU & memory	unspecified
me-kube [41]	multi	threshold-based	proactive	horizontal	SLA	unspecified
Chang [42]	single	threshold-based	reactive	horizontal	CPU	unspecified

The nomenclature in Table 1 mostly follows the taxonomy of [4] and [23]. The *Architecture* refers to the logical application architecture the autoscaler focuses on. By default, an autoscaler scales each service individually based on a set of criteria, in which case the column is labeled *single*. For complex microservice architectures it can be beneficial to use a centralized scaling algorithm that coordinates the scaling of multiple services. This case is labeled as *multi*.

The *Technique* column refers to the model used for determining the need for scaling and estimating the necessary resources. *Threshold-based autoscaling* relies on simple heuristics to identify the need for scaling the application. It is the most popular approach and offered out of the box by many cloud platforms. The performance of this technique depends on the quality of the thresholds and how well the usage scenario can be modeled with static thresholds. Instead of setting static thresholds, reinforcement learning methods can be used to dynamically learn the scaling threshold required to meet a target SLA (*RL-based*).

Forecast-based autoscaling (also called *time series-based*) uses some form of machine learning to model the current workload and predict short-term future workloads. The models can be learned either on-line (at runtime) or off-line (before runtime), though the latter is considered risky as it can have difficulty adjusting to unforeseen workload patterns. The objective function of the machine learning algorithms commonly tries to match an SLA (e.g., application response time), while minimizing the amount of allocated compute resources.

The *Timing* column describes the nature of the algorithm. If the autoscaler works purely based on the current demand, it is considered to be *reactive* because it only reacts to changes in the workload. If the algorithm also partially predicts future demand and scales accordingly, it is considered *proactive*.

The *Scaling Method* refers to the dimension of scaling. Horizontal scaling is characterized by adjusting the number of instances of the same type. Vertical scaling is defined by adjusting the amount of resources allocated to a particular instance. Ideally, both approaches should be combined [19].

The *Metric* column specifies based on which time-series values the autoscaler decides to scale and perform its resource estimation.

The *Workload Pattern* column indicates if the authors developed or tested their algorithm for a particular usage scenario. *Predictable burst* refers to a cyclical usage pattern with a large difference between minimum and maximum utilization. This is a common pattern for news and social media websites, which have large amounts of traffic during the day and very little at night. In case of *unspecified*, the authors did not indicate a specific workload pattern.

Unfortunately, none of the implementations in Table 1 have been made publicly available. This is absolutely necessary to effectively and objectively benchmark different algorithms against each other. Thus, the following focuses on a qualitative evaluation of the algorithms.

3.3.1 KHPA-A

Casalicchio [38] and Casalicchio and Perciballi [11] studied the relation between absolute and relative usage metrics for CPU-intensive workloads. Relative usage measures refer to the utilization reported as a percentage of the allocated resource capacity. These relative measures are exposed through the `cgroup` kernel primitives and are used by most container tools, such as Docker and cAdvisor. They are commonly used because they provide an intuitive notion for horizontal scaling and defining usage quotas [11]. For example, two containers with maximum CPU utilization running on the same host are reported to consume 50% CPU resources, while the host system is consuming 100% CPU resources. However, if just one application is running and the host system is at 50% load, the response time from that application would be quite different. Absolute usage measures refer to the actual, system-wide utilization of resources on the host system. The authors studied this discrepancy and found that the required capacity tends to be underestimated with relative usage metrics, which makes them unsuitable for determining the necessary resources needed to meet a service level objective [38].

In particular, their findings revealed that there is a linear correlation between relative and absolute usage metrics. Using this linear correlation allows transforming relative metrics (such as container quotas and limits) into absolute metrics. Accordingly, the authors developed a scaling component similar to Kubernetes' HPA, but based this one on absolute metrics. The correlation coefficients are the only additional input parameters to the algorithm. This *KPHA-A* performed significantly better in single- and multi-container use cases with high load on the system (more than 80% total CPU load). Specifically, for single instance workloads the application response time with standard HPA was 50% higher compared to using KHPA-A. With concurrent workloads the response time was more than 3 times higher, depending on the particular workload. The KHPA-A was able to achieve this by scaling the number of Pods about 10% higher than standard HPA [38].

The authors argue that the results show that relative usage metrics cannot be reliably used if consistent QoS levels (e.g., application response time) are desired. With their proposed algorithm, QoS metrics can be translated into CPU usage metrics. It should be noted that the workloads in the study (`sysbench` and `stress-ng`) were highly artificial and solely CPU bound. Real-world applications will certainly behave differently when utilizing CPU, memory, storage and network resources. Furthermore, the linear coefficients required for the transformation from relative to absolute usage metrics are specific to the workload and execution environment. Thus, they need to be constantly re-evaluated and it is unclear if this linear correlation also holds true in real-world scenarios with more diverse workloads.

3.3.2 Libra

Balla et al. [39] argue that solely horizontal autoscaling is insufficient for enabling a service to be fully elastic. Therefore, they propose an autoscaler called *Libra* which performs both vertical scaling (determining the appropriate CPU limit) as well as

horizontal scaling (determining the correct number of replicas).

The autoscaler starts out by determining the adequate CPU limit. Libra deploys at least two Pods to avoid affecting the QoS too much while performing these measurements. The *production* Pod is deployed with high CPU limits and serves 75% of the incoming traffic, while the *canary* Pod is used to find the appropriate CPU limit and serves the residual 25% of traffic. The latter is assigned a low initial limit, which is then gradually increased by Libra, until the average number of served requests and the response time converge towards a stable value.

Afterwards, Libra updates the production Pod with the newly determined CPU limit and acts as a horizontal autoscaler. In particular, it increases the number of Pods when the average response time is double the value determined in the previous phase or when the amount of served requests approaches 90% of the value associated to CPU limit. For example, if the appropriate CPU limit has been determined to be 70% and it has been empirically measured that the Pod can serve 1,000 requests with that value, Libra starts adding another Pod when each of the running instances is serving more than 900 requests. Conversely, if the requests per Pod fall below 40%, Libra removes Pod replicas [39].

The authors' experiments showed that Kubernetes' default HPA did not scale the Deployment enough, which led to 40% lower throughput (requests per second), while Libra scaled the Deployment to double the number of Pods. Determining the appropriate resource requests and limits for a service by deploying different kinds of Pods is an excellent idea: it enables accurate live measurements on real-world data without a large impact on clients using the service. Unfortunately, the conducted benchmark was quite simplistic: a web server that simply returns the string "Hello" (and does not perform any other computations or I/O operations). Thus, from these experiments it is unclear whether the same results could be obtained for more sophisticated applications and workload scenarios.

3.3.3 RUBAS

Rattihalli et al. [31] have designed a vertical autoscaler that aims to eliminate the dependency on resource thresholds set by human operators entirely. The *Resource Utilization Based Autoscaling System* (*RUBAS*) is a vertical scaling component that can scale Pod resource limits non-disruptively. Unlike Kubernetes' VPA, which terminates running Pods and creates new ones from scratch (see Section 3.2.2), RUBAS uses *CRIU* (*Checkpoint Restore in Userspace*) to save the application state and restore it when launching a new Pod. This is an excellent achievement that is particularly important for stateful workloads, i.e., those where recreating the service state from scratch would incur a significant performance penalty. An example here would be a database, which needs to have complex and large data structures in memory to operate efficiently.

In contrast to VPA, which uses the peak resource consumption for estimating the new resource requirements, RUBAS bases the estimates on the median of past observations. The authors argue that a temporary peak in resource consumption by the application should not be used for estimating future resource demands.

Their experiments showed that the non-disruptive migration can significantly reduce the execution time of applications (16% runtime improvement) when the initial resource thresholds specified by the user were too low. In this case both VPA and RUBAS need to update the resource thresholds several times before converging on the optimal solution. This result is particularly relevant for one-off batch jobs, not so much for long-running services. They also found that due to resource allocation based on average utilization, RUBAS had to perform fewer migrations (stopping and restarting of Pods) compared to VPA. Conversely, this also increased the CPU (72% compared to 82%) and memory utilization (76% compared to 86%). Higher utilization means that tasks on the cluster will complete faster overall.

A major drawback of RUBAS is that it exists entirely outside of Kubernetes, instead of integrating itself as a control-plane component. This violates the design philosophy of Kubernetes[18] and requires managing and provisioning additional infrastructure. Moreover, RUBAS requires access to both the cluster-level management API (to control workloads in the Kubernetes cluster) as well as underlying infrastructure resources (direct access to worker nodes for creating snapshots; shared volumes for storing snapshots). To this end, the presented architecture is unlikely to find adoption in the Kubernetes ecosystem.

3.3.4 HPA+

Toka et al. [29] proposed a proactive autoscaling approach based on demand forecasting with machine learning. Unlike most scaling algorithms which are reacting only based on the current load of the system (such as the default HPA), their autoscaling engine *HPA+* takes into account a larger window of time and future demand forecasts based on machine learning models.

Under the hood, the HPA+ utilizes several models for scaling: an auto-regression (AR) model, a supervised Hierarchical Temporal Memory (HTM) neural network and an unsupervised Long Short-Term Memory (LSTM) neural network. In [43] the same authors also added a fourth, reinforcement learning-based (RL) model.

Because each model performed poor or well depending on the particular usage pattern, the authors combined all models in a single autoscaling engine. This engine continuously runs all models, but only the model with the best performance on the most recent input (last two minutes) is considered for scaling decisions.

While the underlying algorithms are quite complex, HPA+ packages them into a single parameter which the end user can tune: *excess* describes the trade-off between lower loss (amount of unserved client requests) and higher resource utilization. This is achieved through resource over- and under-provisioning. Compared to the original HPA, the HPA+ only scaled the Pods about 3-9% more (depending on the excess parameter), but had significantly lower request loss. In their follow-up article [43], the authors confirmed these findings with more extensive benchmarks. They generated synthetic data based on real-world traces of Facebook.com website visits on a university campus. While the use of recent real-world data for their tests is

[18] https://kubernetes.io/docs/concepts/extend-kubernetes/

commendable, it should be noted that the artificially-generated traffic pattern was much more spiky (meaning large, positive outliers) than the original input traces, which can be considered as an artifact in the data.

With their autoscaling solution, the authors tried to focus on ease of usability by introducing a parameter which controls the trade-off between resource usage and QoS level violation. However, since the proposed models need to be trained and fitted to the application at hand, setting up such a system is a non-trivial task. In addition, it requires a sizable amount of clean data that closely mirrors the usage scenario of the application, because not every application follows the usage patterns of a social media website.

3.3.5 Microscaler

Yu et al. [5] presented *Microscaler*, a horizontal autoscaler which combines an online learning approach with a heuristic approach for cost-optimal scaling of microservices while maintaining desired QoS levels. The authors introduced a criterion called *service power* to determine the need for scaling and to estimate the appropriate scale. Service power represents the ratio between the average latency of the slowest 50 percent of requests (P_{50}) and the average latency of the slowest 10 percent of requests (P_{90}) during the last 30 seconds. When the service power is close to or above 1 (i.e., $P_{90} \approx P_{50}$), the application can handle most of the requests within the desired QoS level. When the service power falls significantly below 1, it means that the service quality is degraded.

Microscaler mainly considers the QoS experienced by the user, instead of QoS of individual services. As a consequence it scales not just because a service has high CPU utilization or increased response time, but only when a user-facing SLA is violated. This way, it can avoid detecting false-positive scaling events. In their case, response time SLAs can either be violated by falling below the minimum threshold T_{min} or by rising above the maximum threshold T_{max}. The service power criterion is then incorporated into a Bayesian Optimization approach, which allows minimizing an objective function (i.e., cost) while obeying constraints (i.e., SLA bounds).

Unfortunately, the results presented in their work are inconclusive: while Microscaler converges slightly faster to the desired QoS level than other autoscaling approaches, the difference is not significant. Also, the mathematical model carries complexity and several parameters which need to be adjusted depending on the application [5].

3.3.6 Q-Threshold

Horovitz and Arian [40] proposed an algorithm called *Q-Threshold* for dynamically adjusting horizontal scaling thresholds. It is important to note that this solution is not a full autoscaler by itself: instead it is a machine learning algorithm that only learns and suggests the ideal thresholds for scaling. For example, “to not raise response time above 100ms, add more replicas when CPU utilization is above

78.5%". This makes the operation more transparent to the cluster administrator and requires less trust, as the administrator remains in full control.

As the name suggest, Q-Threshold leverages a Q-Learning algorithm and is enhanced with several optimizations for faster convergence. Q-learning is a model-free reinforcement-learning algorithm that learns the optimal actions in state space through a reward function. In the context of horizontal scaling the goal of Q-Learning is finding the optimal autoscaling policy while obeying a specified SLA. Therefore, the reward function needs to trade-off SLA violations (in their case response time, which should be as low as possible) for resource utilization (which should be as high as possible). When the SLA is violated, the reward is negative.

The authors have conducted extensive simulations with different variations of their algorithm and found satisfying results: Q-Threshold completely avoids SLA violations and has a stable behavior when scaling due to workload changes. They also compared their algorithm against a static scaling threshold of CPU utilization (25% lower bound, 75% upper bound). Unsurprisingly, the proposed algorithm performed better than a static threshold in this comparison. Unfortunately, the authors neither gave a detailed description of the system implementation nor how the tests were conducted. Therefore, it is ultimately unclear how well the Q-Threshold algorithm would work in a real-world scenario, despite promising simulation results.

3.3.7 me-kube

Rossi et al. [41] presented *Multi-Level Elastic Kubernetes* (*me-kube*), an autoscaler that coordinates the horizontal scaling of microservice-based applications. It is based on a two-layered control loop. On the lower layer, the *Microservice Manager* controls the scaling of a single service by monitoring and analyzing metrics with a local policy. This local policy can either be proactive (in this case reinforcement learning-based) or reactive (application metric-based). On the higher layer, the *Application Manager* controls the scaling of an entire application (which can be composed of multiple services) by observing the application performance. When the Microservice Manager detects a need to scale the service, it sends a *proposal* to the Application Manager. A proposal contains the desired number of replicas for the service as well as a *score*: it describes the estimated improvement of the proposed adaptation.

Conversely, when the Application Manager detects an SLA violation, it requests proposals from the Microservice Managers. The Application Manager then coordinates the scaling of several microservices to avoid interfering scaling decisions (e.g., service B is scaled down because it is not receiving traffic from service A, which is under high-load). To evaluate scaling decisions, the Application Manager considers all submitted reconfiguration proposals and chooses the one with the highest score, since it considers that one to have the highest impact on the overall application SLA. This process is repeated iteratively until the target response time is met again or until there are no more proposals. Once a decision has been made, the scaling action is communicated down to the Microservice Managers.

In their experiments the authors tested several scaling approaches. They found

that Q-Learning performed the worst in a moderately complex deployment scenario, which drastically increased the state space and thereby made it hard to learn for the model. Kubernetes' HPA produced several policy violations, mainly because the workload was not necessarily CPU bound but HPA still scales based on CPU utilization by default. The hierarchical (i.e., centrally coordinated) scaling approach combined with a local predictive policy (ARIMA) performed the best, since it only caused SLA violations at the very beginning of the stress test.

In summary, the central coordination of scaling decisions in microservice-based applications is a promising approach. The authors have not documented how they deployed or configured the additional scaling components, therefore it is somewhat unclear which impact these aspects would have on a real-world deployment. Moreover, an additional Microservice Manager is required for each microservice, which can have significant compute requirements, depending on the model and amount of data used for the scaling algorithm.

3.3.8 Chang

Chang et al. [42] proposed a generic platform for dynamic resource provisioning in Kubernetes with three key features: comprehensive monitoring, deployment flexibility and automatic operation. All three are essential for operating a large, distributed system on top of Kubernetes (or any other orchestration platform).

Their monitoring stack is based on Heapster (for collecting low-level system metrics), Apache JMeter (for generating application load and measuring response time), InfluxDB (time-series database for storing metrics) and Grafana (as a visualization tool). It needs to be pointed out that the authors decided to inject artificial load into the system to measure application performance, instead of collecting "native" system metrics. In this sense, they are measuring and collecting performance data at the client side, instead of at the server-side. This approach has the advantage that it captures the service level experienced by clients more accurately [3]. However, it also introduces additional stress on the system, which might be undesirable when the system is already under heavy load.

The authors described their *Resource Scheduler Module* and *Pod Scaler Module*, which adjust the number of running Pods based on CPU utilization with static thresholds. The application (or the application developer) cannot directly set these thresholds, instead they need to be set by the cluster operator. So while it is true that their setup eases "automatic operation", it does not allow any configuration of scaling parameters at runtime. However, this is somewhat understandable as this was one of the first articles giving an extensive coverage of a custom autoscaling infrastructure in Kubernetes.

3.4 Summary

The official autoscaling components for Kubernetes (HPA, VPA, CA) are widely and successfully used. On one hand, their fundamental algorithms are simple (no complex mathematical equations or machine learning algorithms), giving cluster

operators the ability to intuitively understand and trust them. On the other hand, there are several parameters that can be tweaked for scaling, which gives operators the ability to adjust the scaling behavior to their application and use case. Most importantly, these components come with a well-defined API (in form of Kubernetes objects) and have been *battle-tested* in production systems, meaning they are proven to be reliable and effective.

The considered research articles have explored several novel directions for Kubernetes autoscalers. Most of the authors focused on horizontal scaling, as scaling in this dimension tends to be less error-prone compared to vertical scaling. Additionally, horizontal scaling works well at small as well as large scales, whereas vertical scaling is better suited for larger systems (due to Kubernetes limitations discussed in 3.2.2). However, work from Google [19] has shown that vertical autoscaling has two important benefits. First, vertical autoscalers adjust resource reservation and limits much better to actual usage compared to static allocation by developers, which results in cost savings due to increased resource utilization. Delimitrou and Kozyrakis [44] found that 70% of the time developers request more resources than their application actually requires, while in 20% they under-request resources. Second, these autoscalers alleviate the developer's burden of having to accurately set reservations and limits on their services. This increases productivity, because humans are not only slow and inaccurate at estimating required resources, but they also need to continuously perform this task while the software is developed and updated.

Most of the solutions looked at scaling each service individually; only one (me-kube [41]) implemented centralized (*hierarchical*) scaling of several services. While coordinated scaling can offer significant benefits, it is difficult to generalize this approach from simple test cases to large, complex meshes of production systems. Bauer et al. [45] have also considered hierarchical scaling research, but they have not implemented and validated their proposal with Kubernetes.

From the research it seems clear that proactive autoscaling (i.e., scaling not only based on current load, but based on predicted future load) is beneficial for aggressive up- and down-scaling. The major impediment here is the complexity of the algorithms. More complex algorithms take more time to train and evaluate. Potentially they need to be fed with large amounts of data, too. Additionally, cluster operators are no longer able to understand why the system is behaving in a certain way, which is highly undesirable. Thus, a balance needs to be struck: simple predictive models such as ARIMA appear to work well for this [41, 46].

No conclusion has been reached about whether a service should be scaled based on low-level (e.g., CPU, memory utilization) or high-level metrics (SLA). While several of the previously presented articles found that high-level metrics work much better, Rzadca et al. [19] advise against directly optimizing application metrics based on their experience at Google. They argue that this job should be left to the service developers to allow them to reason about the performance of their service. It also separates concerns between infrastructure and services. Thus, the choice of scaling metrics depends on the development context and application usage scenario.

One common theme among all works reviewed above is the lack of details about

implementation. Not only the underlying algorithms are important when setting up a production-grade system, but also how the system is configured and operated. Kubernetes provides excellent tools for managing resources inside a cluster. Yet none of the proposed autoscalers have been integrated into Kubernetes the way HPA, VPA and KEDA are: once installed, they can be easily configured for each individual Deployment by attaching *Custom Resource Definitions* (CRDs).

3.5 Modular Kubernetes Autoscaler

We believe there is a gap between industry and academia in the area of autoscaling research. On the one hand, researchers want to test and evaluate their novel algorithms for autoscaling without having to worry about the integration with other Kubernetes components. This can be a challenging task due to the high development velocity of the Kubernetes ecosystem. On the other hand, industry practitioners value reliability and interoperability: Kubernetes has been designed from the ground up with a strong focus on reuseable and well-defined APIs. Thus, any new autoscaling component should use – as well as expose – these APIs.

We propose a modular autoscaling component for Kubernetes that combines these two goals: it provides a higher-level abstraction for the autoscaling algorithm by placing it inside a sandbox with simple interfaces. This provides a separation of concerns: the authors of the algorithm can focus only on the scaling logic, while the autoscaling component takes cares of the interaction with Kubernetes and other external systems. In the following, we outline its key design and architectural aspects. The autoscaling component should be implemented using the Go programming language to align with other Kubernetes components and make use of the official Kubernetes client libraries[19]. The architecture of the autoscaling component (shown in Figure 6) is similar to HPA and VPA: it fetches metrics from the Metrics API or an external monitoring system, and is configured with Kubernetes CRD objects. The CRD object is the interface between the user and Kubernetes cluster since it contains the workload-specific scaling configuration. Appendix A.9 shows a preliminary example of this CRD.

At runtime, the autoscaling component passes the metrics and scaling configuration to a WebAssembly sandbox, which runs the core autoscaling algorithm and returns scaling results. Afterwards, the autoscaling component performs the actions described by the results of the algorithm (e.g., increase the amount of replicas to a certain number) by communicating with the Kubernetes API.

WebAssembly[20] is a portable binary instruction format that can be used as a compilation target for many programming languages. Its main features are speed, memory safety and debuggability [47]. Running the core scaling algorithm inside a WebAssembly sandbox has several advantages for researchers:

- the algorithm can be implemented in any programming language (Python, JavaScript, Rust etc.) and then compiled to WebAssembly bytecode;

[19] https://github.com/kubernetes/client-go
[20] https://webassembly.org/

- the code runs at near-native speed (faster than interpreted languages), allowing the implementation of complex and resource-intensive algorithms (e.g., neural networks);
- the sandbox provides simple interfaces for data input and output (i.e., no need to interact directly with the complex and evolving Kubernetes API), which simplifies development, testing and simulation.

Cluster operators gain the following advantages from the WebAssembly sandbox:

- memory safety avoids security bugs and strictly bounds the potential impact of errors (if the algorithm fails it might produce garbage output, but will not affect the stability of other system components);
- the WebAssembly bytecode can be replaced on the fly, thus allowing upgrading or changing the algorithm dynamically;
- the autoscaler can host multiple WebAssembly sandboxes, which allows different services to be scaled with individual algorithms;
- the same scaling algorithm can be used across different hosts and environments because the bytecode is agnostic in terms of processor architecture and operating system.

For similar reasons, the Kubewarden project[21] uses WebAssembly sandboxes to implement modular security policies in Kubernetes.

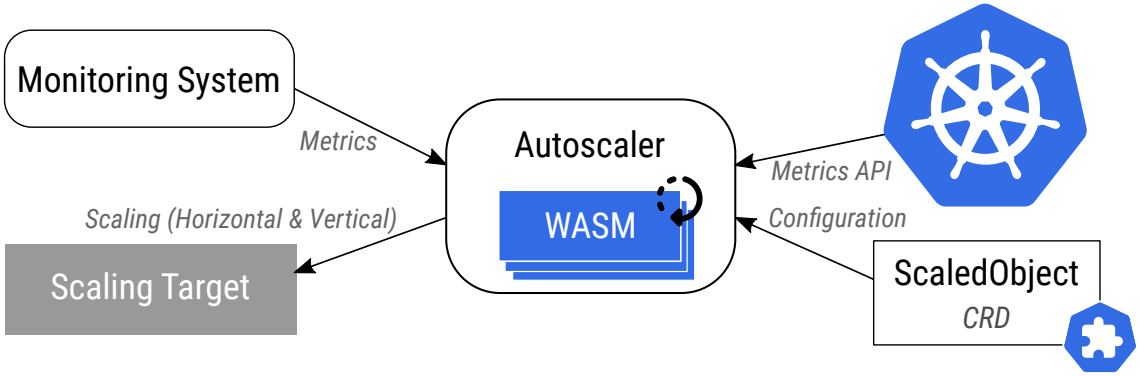

Figure 6 – High-level operational diagram of modular Kubernetes autoscaler with WebAssembly (WASM) sandbox

Additionally, the modularity of this autoscaler allows one important distinction from HPA and VPA: it can combine the decision for horizontal and vertical scaling into one algorithm and one component. HPA and VPA are separate, uncoordinated components which cannot be used to scale on the same metric (otherwise race-conditions might occur and lead to unstable behavior[22]). The proposed modular autoscaling component does not have this issue since horizontal and vertical scaling decisions are generated from the same component – and are therefore conflict-free.

The implementation of this modular Kubernetes autoscaler is left as future work.

[21] https://www.kubewarden.io/

[22] https://github.com/kubernetes/autoscaler/blob/master/vertical-pod-autoscaler/README.md#known-limitations

4 Implementation

"If you are not monitoring stuff, it is actually out of control."
— John Wilkes

This chapter presents a monitoring infrastructure and autoscaling policies for a Kubernetes-based application in a production-grade environment. It details the technical implementation necessary to identify and expose relevant metrics from the target application and execution environment; aggregate and visualize those metrics with a modern monitoring solution; install VPA, HPA and KEDA autoscaling components; and configure the autoscaling policies.

The application in which we introduce and evaluate autoscaling capabilities is *Ericsson Security Manager* (ESM). Among other features, it provides policy-based security automation, compliance monitoring and security analytics functions for telecommunication infrastructure. It aids network operators to automate security controls and maintain them in the desired state. As the complexity of modern telecommunication infrastructure grows, it is crucial for these systems to be configured securely and remain that way.

One part of the target application is responsible for connecting to the external systems, checking their security settings and if necessary re-configuring them. The architectural design of the application which covers this functionality is shown in Figure 7. The API server accepts commands from the user, such as *"check security settings of system A and B"*. It then fetches the necessary connection details from the database and forwards detailed task instructions via the message queue to an executor. To minimize the security risk of connecting to external services, the executor acts as a "dumb client" for these tasks. This means that it does not have access to any other parts of the application and only executes the given tasks. Once finished, it returns the results of the operations to the API server through the message queue. Finally, the API server stores the results in a database and makes them available to the user. As connecting to external systems and running these tasks takes time, this entire process is asynchronous.

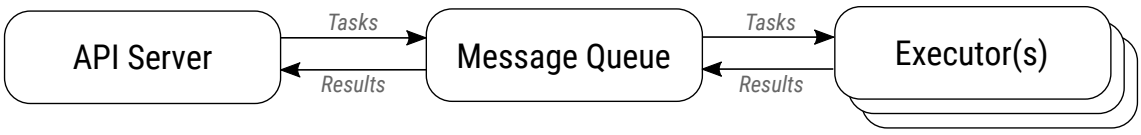

Figure 7 – Partial software architecture of target application

The goal of our work in this chapter is to dynamically scale the executor component based on the current load. This includes setting up monitoring for the system, identifying relevant metrics and setting up autoscaling components based on those metrics. While we focus our implementation on the target application, the methods and findings are generalizable to any application that uses batch- and queue-based processing.

4.1 Monitoring

Monitoring is the first phase of the *MAPE-K* control loop (Section 3.1). It refers to observing the state of the execution environment to detect failures, trigger alerts and provide information about overall system health.

Modern monitoring systems are based on metrics. A *metric* is a numeric value of information represented as a time-series, i.e., each value is associated with a unique timestamp. A *service-level indicator* (SLI) is a metric which measures a specific dimension of the *quality of service* (e.g., response time, error rate). A *service-level objective* (SLO) defines the range of acceptable values for an SLI within which the service is considered to be in a healthy state. A *service-level agreement* (SLA) can be based on an SLO and is a formal commitment from a service provider towards its users. It usually also specifies responsibilities and compensation when an SLO is violated [3].

To collect low- and high-level metrics about the target application, we use an industry-standard monitoring stack consisting of Prometheus and Grafana (shown in Figure 8). Prometheus is a time-series database developed to collect and store numeric values (indeed "metrics") [48]. Grafana is a visualization tool that can use Prometheus as a data source for plotting graphs, heatmaps and diagrams. This means Grafana itself does not store any data, but fetches the relevant data from Prometheus on the fly using the PromQL query language.

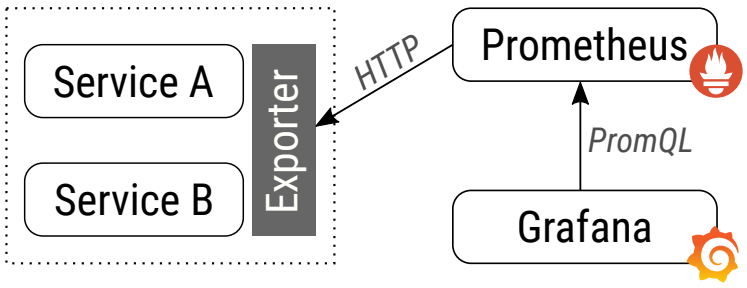

Figure 8 – Logical view of monitoring infrastructure with Prometheus and Grafana

Prometheus itself does not extract metrics from the system or application, but rather relies on so-called *exporters*[23]. These exporters expose relevant metrics through an HTTP endpoint in a plaintext format[24], which then gets queried periodically (according to the *scrape_interval*) — this process is referred to as *scraping*. For many commonly used services (databases, message queues, operating systems etc.) open-source exporters already exist[25]. A *custom* exporter needs to be developed to expose metrics from a proprietary or novel application.

We install and configure Prometheus and Grafana with Helm Charts as shown in Appendix A.1 and A.2, respectively. Helm Charts[26] provide a convenient and declar-

[23] https://prometheus.io/docs/instrumenting/exporters/
[24] https://prometheus.io/docs/instrumenting/exposition_formats/
[25] https://exporterhub.io/
[26] https://helm.sh/

ative way to install complex applications into a Kubernetes cluster. This declarative installation makes our research reproducible in any Kubernetes environment.

It is important to keep in mind that Prometheus is fundamentally a pull-based monitoring solution. This means the Prometheus server opens a connection to the exporters, not the other way around (push-based). Thus, the network topology and firewalls must allow this. In Kubernetes, this is configured through Network Policies[27].

Prometheus offers built-in support for Kubernetes service discovery, therefore *scraping targets* (the endpoints from which Prometheus collects metrics) do not need to be defined statically in a configuration file. Instead, they can be enabled and configured with Kubernetes annotations. Annotations are small pieces of key-value metadata which can be attached to Kubernetes Pods, Services or Ingresses. Prometheus then automatically discovers the objects with appropriate labels and starts scraping the associated endpoint. An example of such a Service annotation is shown Appendix A.3. This behavior is an instance of a self-configuring system according to the concept of autonomic computing [14].

4.2 Prometheus Exporters

Since Prometheus itself does not extract metrics from an application, installing and configuring special exporters is necessary. This section documents which exporters have been tried and deployed to collect metrics as well as the purpose they serve. We start with low-level metrics and gradually move towards higher-level metrics. It is important to note that only when metrics from different sources (exporters) are combined, it is possible to obtain a comprehensive view of the application behavior and performance.

The *kube-state-metrics* exporter[28] exposes details about the state of objects managed by Kubernetes' control plane. For example, it reports the number of Deployments and Pods as well as configuration of these objects (e.g., resource requests and limits) to Prometheus. It should be noted that these metrics only describe the state of virtual objects. To get real-time information about the state of the Kubernetes worker nodes (CPU, memory, I/O utilization), Prometheus is configured to scrape metrics from *cAdvisor* (refer to Section 2.3.2). Combining these two data sources yields insight into how individual Pods and containers are behaving, as shown in Figure 9.

Next, we looked into extracting metrics from *RabbitMQ*, the message queue used in the target application. RabbitMQ comes with a plugin that only needs to be enabled for exposing a metrics endpoint[29]. After setting up a dashboard and observing the metrics, we had to realize that these metrics are useful for accessing the status and health of RabbitMQ itself, but are not detailed enough for our purposes. In particular, it was not exposing statistics about individual channels and queues (e.g.,

[27] https://kubernetes.io/docs/concepts/services-networking/network-policies/
[28] https://github.com/kubernetes/kube-state-metrics/tree/v1.9.8
[29] https://www.rabbitmq.com/prometheus.html

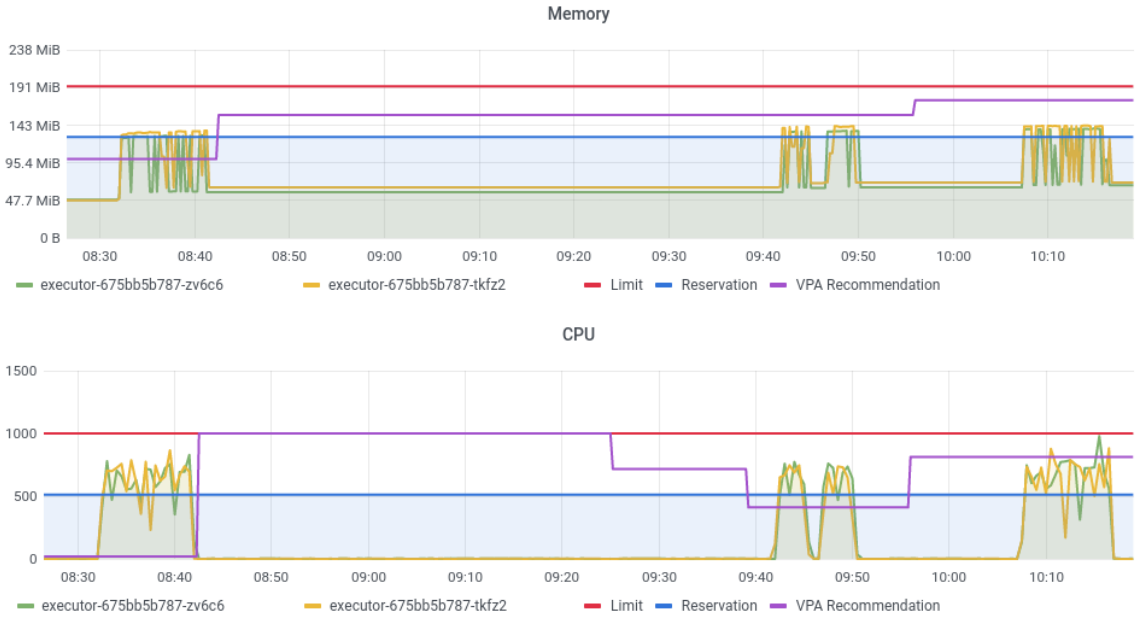

Figure 9 – Grafana screenshot with Container CPU and Memory Details. Red lines indicate resource limits, blue lines resource requests. Yellow and green lines are actual utilization (two replicas). Purple lines indicate VPA recommendations.

queue length). Thus, it was decided to use a third-party exporter for RabbitMQ[30]. This exporter gives access to detailed statistics, which are necessary to differentiate the messaging activities of different microservices.

Afterwards, we proceeded to a higher level and looked at logical tasks. The target application uses the *Celery* Python framework[31] for sending tasks between microservices. Celery is using RabbitMQ as a message broker, but provides higher level concepts to the application (e.g., scheduling, rate limiting, persistence) and decouples it from the specific message broker implementation. Two open-source Prometheus exporters for Celery[32] [33] are available. Fundamentally, they both work in the same fashion, but expose slightly different metrics. We found that the metrics these exporters provide give more useful insights than the raw metrics from RabbitMQ, because they differentiate between tasks of different types (not just messages on a queue) and higher-level semantics (e.g., task cancelled).

Still, these metrics did not give us a full picture, because any moderately complex application is more than just the sum of its components. To accurately measure (and possibly predict) the application performance, also the state of the application and business logic (that connect the individual components) needs to be captured. Therefore, we decided to build a custom exporter for our target application. This allows us to extract high-level application metrics which are not accessible otherwise. For example, our target application allows running tasks according to a schedule.

[30] https://github.com/kbudde/rabbitmq_exporter
[31] https://docs.celeryproject.org/en/stable/
[32] https://github.com/danihodovic/celery-exporter
[33] https://github.com/OvalMoney/celery-exporter

These tasks are then stored in a database until a certain condition is met, then they are sent to the executor over the message queue. In this case, simply looking at the tasks in the message queue would give an incomplete picture.

As the name suggests, where and how a custom exporter obtains metrics is highly specific to the application. Nevertheless, the following section documents a brief example of how to conduct such an endeavor. Our implementation is written in Go and uses the official Prometheus client library[34]. The library provides a framework for exposing metrics via HTTP. Specifically, each time a client connects to the HTTP endpoint, the `Collect` method of the associated collectors is called (Listing 1).

The management interface of our target application exposes all running, finished and scheduled tasks in JSON format. The exporter shown in Listing 1 queries the relevant API endpoint (line 3), parses the data and walks through the list of tasks (line 7). It categorizes the tasks based on their status (running, finished, failed etc.) and counts the number of tasks in each category (line 8). This number is then published through the HTTP endpoint of the exporter (line 13). In this case, the type of metric is a *Gauge* (line 15): an arbitrarily increasing and decreasing value, e.g., number of active tasks [48].

Listing 1 – Sample code of custom exporters

```
1  func (c *Collector) Collect(channel chan<- prometheus.Metric) {
2      // query management interface for task information
3      tasks, _ := c.queryConfigurationApi()
4
5      // analyze all tasks
6      results := make(map[string]float64)
7      for _, task := range tasks {
8          results[task.Result] += 1
9      }
10
11     // publish metric via HTTP endpoint
12     for status, count := range results {
13         channel <- prometheus.MustNewConstMetric(
14             c.tasks, // contains metric name and description
15             prometheus.GaugeValue, // type of metric
16             count, status) // metric value and metric label
17     }
18 }
```

The code in Listing 1 produces the partial output shown in Listing 2 on the `/metrics` HTTP endpoint.

Listing 2 – Sample metric output

```
# HELP esm_tasks_total Total number of tasks labeled by status.
# TYPE esm_tasks_total gauge
esm_tasks_total{status="failed"} 0
esm_tasks_total{status="running"} 42
esm_tasks_total{status="finished"} 7
```

[34] https://github.com/prometheus/client_golang/tree/v1.10.0

Additionally, we implemented several other metrics with types Counter and Histogram. A *Counter* is a monotonically increasing value, which is only reset when restarting the service [48]. Exemplary, it can be useful for describing the total number of tasks created by the application. A *Histogram* samples observations into buckets with pre-configured sizes [48]. It can be used to describe the duration of requests, for instance 0-10ms, 10-100ms, 100ms-1s, 1-10s etc. A histogram provides a balance between tracking the duration of each task individually (which has high *cardinality*, i.e., expensive in terms of bandwidth and storage resources) and aggregating into mean, minimum and maximum values (which loses information about distribution and outliers).

While developing this custom exporter, we followed the best practices and conventions for writing Prometheus exporters[35] and metric naming[36]. The custom exporter was packaged into a container image and deployed alongside the target application. We were able to confirm that it provides useful high-level metrics about the application by setting up a Grafana dashboard and visualizing the metrics as time-series graphs. The exposed metrics allow reasoning about the application behavior and making appropriate scaling decisions based on the collected metrics.

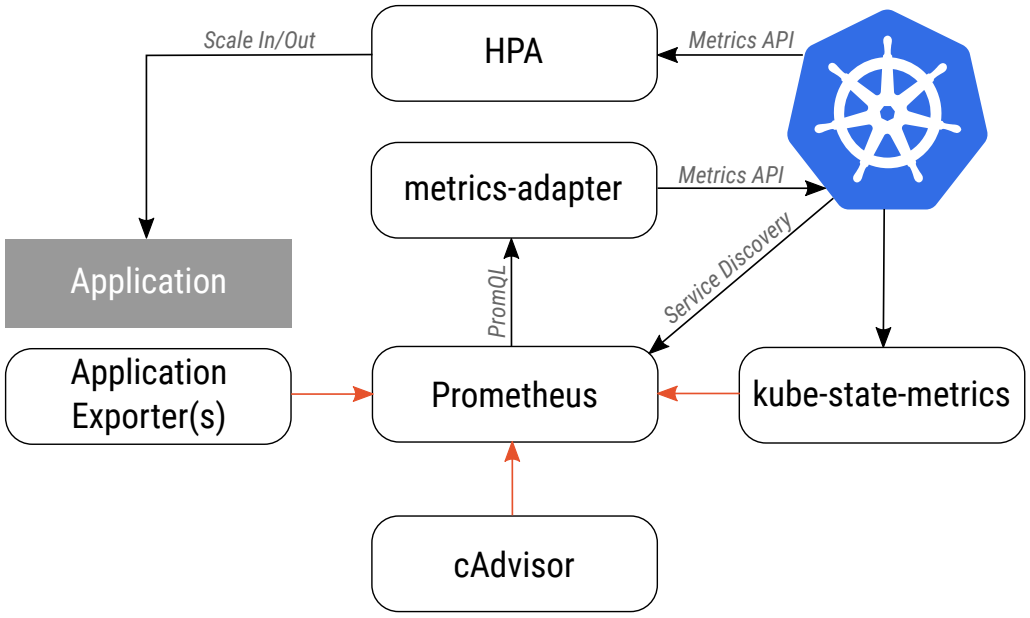

Figure 10 – Flow of metrics used for scaling. Arrows denote the logical flow of data. Orange arrows symbolize raw HTTP metrics.

To use these custom metrics with Kubernetes HPA (Section 3.2.1), another component needs to be installed into the cluster: a *metrics adapter* (Figure 10). This component is responsible for translating the metrics from Prometheus into a format compatible with the Kubernetes metrics API (Section 2.3.2). We choose the *prometheus-adapter* project[37] for this purpose as our use case focuses solely on

[35] https://prometheus.io/docs/instrumenting/writing_exporters/
[36] https://prometheus.io/docs/practices/naming/
[37] https://github.com/kubernetes-sigs/prometheus-adapter

Prometheus metrics. Another project with a similar goal is *kube-metrics-adapter*[38] which allows utilizing a wider range of data sources, for example InfluxDB or AWS SQS queues. The installation of the adapter was performed with a Helm Chart and is detailed in Appendix A.4. In essence, the adapter is configured with a PromQL query it should execute. The query can be parameterized with several labels and parameter overrides. The result of this query is exposed as a new metric through Kubernetes' metrics API (Figure 10).

4.3 Autoscaling Setup

The previous section outlined the setup of the entire monitoring pipeline. This section details how Kubernetes' autoscalers need to be configured to use the collected metrics. As outlined in Chapter 3, scaling can be performed in two dimensions: horizontally and vertically.

Horizontal scaling (scaling in and out) refers to creating more replicas of the same Pod. Assuming that the workload is automatically distributed across all instances, scaling out results in a lower workload per replica on average. It should be noted that this assumption does not always hold true. Special attention needs to be paid to (partially) stateful services. Nguyen and Kim [49] performed an investigation of load balancing stateful applications on Kubernetes. They found that especially distribution and load balancing of leaders throughout the cluster are important for maximizing performance when scaling horizontally.

Two other aspects need to be considered when implementing horizontal scaling on Kubernetes: microservice startup and shutdown. These aspects are particularly important when using autoscaling, since individual Pods are frequently created and removed at all times.

If the Pods of a microservice are exposed with a Kubernetes Service (see Section 2.3.1), the Pods should have *readiness probes* configured. Based on this probe Kubernetes determines if the application is ready to handle requests (e.g., after it has finished its startup routine). Only then Kubernetes starts routing network traffic to a newly started Pod [17]. This way the new replicas handle part of the incoming load as soon as possible – but not too early – while scaling out.

Any application running as a distributed system should implemented *graceful shutdown* (or *graceful termination*): when a Pod is shut down, Kubernetes stops routing new traffic to the replica and sends the Pod a SIGTERM signal; the application should finish serving the outstanding requests it has accepted and terminate itself afterwards [17].

Vertical scaling (scaling up and down) refers to adjusting requested resources (compute, memory, network, storage) allocated to a service based on the actual usage. By giving more resources to one or multiple instances, they are able to handle more workload. While most industry practitioners only focus on scaling up (allocating more resources), the opposite is actually far more desirable: scaling down. The *Autopilot* paper from researches at Google shows that significant cost

[38] https://github.com/zalando-incubator/kube-metrics-adapter

savings can be realized by automatically adjusting the allocated resources, i.e., vertical scaling [19].

Some of the research proposals discussed in Section 3.3 have shown potential to be effective and cost-efficient autoscalers, but none of them offer a publicly available implementation. For this reason, HPA (Horizontal Pod Autoscaler, Section 3.2.1), VPA (Vertical Pod Autoscaler, Section 3.2.2) and KEDA (Kubernetes Event-driven Autoscaler, Section 3.2.4) were chosen for autoscaling: they are widely deployed in the industry and their implementations are battle-tested. Furthermore, they feature a plethora of configuration options to adjust the scaling behavior. This allows developers and administrators to fine-tune the scaling behavior to their use cases and goals. These options – as well as their effects – will be explored in the following sections.

4.3.1 Vertical Scaling with VPA

Since VPA is an external component not included in a standard Kubernetes distribution, it needs to be installed separately. The installation process with a third-party Helm Chart as well as the configuration options to connect VPA to Prometheus are shown in Appendix A.7. Notably, we only configured the *Recommender* component, but not the *Updater* or *Admission Controller* (see Section 3.2.2). This decision was made because as of Kubernetes 1.20, the resource allocation of a created Pod is immutable. Since only the *PodTemplate* can be modified, the Pod needs to be deleted and created from scratch for new resource requests and limits to take effect [17]. Thus, it has potential for service disruption, especially when the number of Pod replicas is small (removing 1 out of 100 Pod replicas does not make a significant difference, but removing 1 out of 3 replicas can impact overall service health). Nevertheless, the VPA Recommender can be a useful tool for determining appropriate resource requests and limits, as we show in the following section.

After the component is installed into the cluster, VPA needs to be instructed to monitor our application so that it can build its internal resource usage model and produce an estimate. VPA is enabled and configured for each application running on Kubernetes individually. This is done through a special object called *Custom Resource Definition*, short *CRD*. CRDs act just like Kubernetes core objects, however they are not implemented by the *Controller Manager* (Section 2.3.2), but through an external component — in this case: the VPA.

Listing 3 shows the CRD for configuring VPA to monitor the `executor` Deployment (line 6-9). VPA is instructed to only provide resource recommendations, but not change the configuration of running Pods (lines 10-11). We configure VPA to monitor a specific container in the Pod (this avoids interference with sidecar containers) and the types of resources (lines 13-16). CPU and memory are the only resources supported by VPA. The CRD is added to the cluster with `kubectl apply` in the same namespace as the target Deployment.

Listing 3 – Vertical scaling configuration CRD for VPA

```
apiVersion: "autoscaling.k8s.io/v1"
kind: VerticalPodAutoscaler
metadata:
  name: executor
spec:
  targetRef:
    apiVersion: "apps/v1"
    kind: Deployment
    name: executor
  updatePolicy:
    updateMode: "Off" # Recommendation only
  resourcePolicy:
    containerPolicies:
      - containerName: "executor"
        mode: "Auto"
        controlledResources: ["cpu", "memory"]
```

Once the VPA has been able to collect metrics for a while, the resource request and limit recommendations can be retrieved as shown in Listing 4. Section 3.2.2 explains the meaning and calculations behind these values. For our use case, only the upper bound and target recommendations are relevant. The upper bound can be used as the resource limit, the target value can be used as a resource request for the container.

Listing 4 – VPA Resource Recommendations

```
$ kubectl describe vpa/executor
  [...]
  Recommendation:
    Container Recommendations:
      Container Name: executor
      Lower Bound:
        Cpu:     15m
        Memory:  163355301 # 163 MB
      Target:
        Cpu:     763m
        Memory:  163378051 # 163 MB
      Uncapped Target:
        Cpu:     763m
        Memory:  163378051 # 163 MB
      Upper Bound:
        Cpu:     1045m
        Memory:  174753812 # 175 MB
```

Thanks to the extensive monitoring setup deployed in Section 4.2, the operation of the VPA can be visualized with Grafana. The purple lines in Figure 9 indicate the VPA recommendations for the memory and CPU resource. It can be seen that over time VPA is adjusting the recommendations based on the actual usage of the Deployment. The longer the Deployment is running with a production workload, the more accurate the VPA estimates are.

As mentioned previously, we are only using the VPA as a tool to provide resource recommendations during development, not for setting the resource values at runtime

in production environments. We plan to use this VPA deployment in a stability and performance test environment, where it can observe the application over a long period of time during a high-load scenario. This will help guide the developers to make educated decisions about the appropriate resource requests and limits for their containers. A tool like *Goldilocks*[39] can provide a dashboard to visualize the recommendations and make them easily accessible through a web interface.

4.3.2 Horizontal Scaling with HPA

This section details the configuration of Kubernetes' Horizontal Pod Autoscaler (HPA) [28]. As outlined Section 3.2.1, HPA is implemented in the Controller Manager and therefore part of every Kubernetes installation. Therefore, no installation is required, HPA just needs to be configured for each scaling target.

The goal of the HPA in our scenario is to give the application similar performance to a static overprovisioning of resources, while keeping the cost at a minimum. The exact metrics for quantifying these dimensions are discussed in Chapter 5. Additionally, the autoscaler should be able to find this optimum trade-off with varying workload sizes. Mathematically, the resulting system can be described as a queuing system where the number of workers is adjusted dynamically based on the queue length [50].

The most minimal horizontal scaling policy could be applied with the command `kubectl autoscale executor --cpu-percent=50`. This would scale the number of replicas based on the average CPU load across all Pods in the Deployment. However, as discussed at the beginning of this chapter, our workload is neither purely CPU nor memory bound, but also by the throughput of external systems. Thus, we need to scale this Deployment with a high-level metric which we exposed in Section 4.2.

Based on empirical observations and experiments, we identified the current queue length (used in Figure 11) as a meaningful autoscaling metric. Thanks to the metrics adapter installed in Section 4.2, we can configure the HPA to scale based on external metrics, as shown in Listing 5. The object structure is similar to the CRD of the VPA. It specifies a *target* – the Kubernetes object which should be scaled (line 6-9) – and based on which metric it should be scaled (line 13-16). The goal of the HPA is to make the metric value equal to target value (line 17-19) by adjusting the number of Pods. When the metric value is above the target, it creates more instances. When the metric value is below the target, it removes instances. For details about the algorithm refer to Section 3.2.1. Additionally, we specify safety bounds: the Deployment must have at least 1 replica (line 10) and at most 20 replicas (line 11). This is an important engineering practice to guard against bugs and misconfiguration (e.g., the unit of the metric value changes from seconds to milliseconds), which could lead to automatic creation of large numbers of replicas.

After applying the `HorizontalPodAutoscaler` object (shown in Figure 5) to the cluster, the current configuration as well as operation of HPA can be observed on the command line (Appendix A.6) as well as visually with the monitoring setup (Figure 11).

[39] https://goldilocks.docs.fairwinds.com/

Listing 5 – Initial HPA Scaling Policy (`hpav0`)

```
1  apiVersion: autoscaling/v2beta2
2  kind: HorizontalPodAutoscaler
3  metadata:
4    name: executor # name of the autoscaling object
5  spec:
6    scaleTargetRef: # Deployment to be scaled
7      apiVersion: apps/v1
8      kind: Deployment
9      name: executor
10   minReplicas: 1 # safety bounds
11   maxReplicas: 20
12   metrics:
13   - type: External
14     external:
15       metric: # scale based on this external metric
16         name: esm_tasks_queued_total
17       target: # scale until this value is reached
18         type: Value
19         value: 1
```

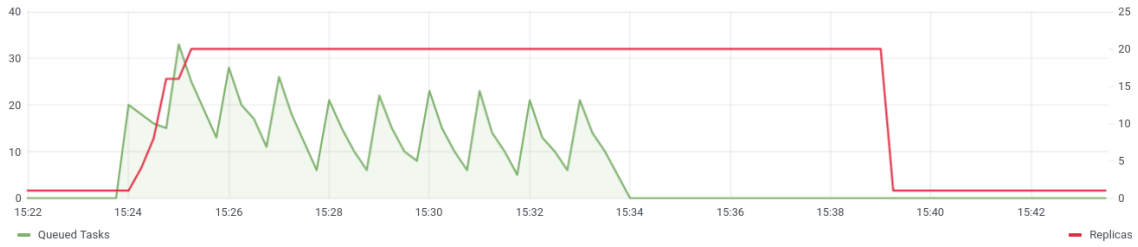

Figure 11 – Grafana screenshot of scaling behavior with initial HPA policy (`hpav0`). Task queue length in green, number of replicas in red.

4.3.3 Horizontal Scaling with KEDA

This section describes the setup and configuration of the *Kubernetes Event-driven Autoscaler* (KEDA). As noted in Section 3.2.4, KEDA significantly reduces the number of components required to use custom metrics for autoscaling in Kubernetes. A monitoring tool, exporters and a metrics adapter (as documented in the previous sections) are not required for using KEDA. The installation procedure with KEDA's Helm Chart is shown in Appendix A.8.

Similar to the previous section, we will use the RabbitMQ message broker as a trigger for horizontal scaling (Listing 6, line 14). Specifically, KEDA is configured to scale the `executor` Deployment based on the number of messages in a specific queue (line 18-20). Alternatively to scaling based on queue length, the rate of messages could also be used.

Listing 22 in Appendix A.8 shows the status of KEDA's `ScaledObject`, the HPA object (created by KEDA) and the Deployment after applying the CRD from Listing 6. Of particular note is that the HPA object has a minimum Pod count of one, but the

KEDA agent scales the Deployment to zero replicas anyway. This allows saving resources when there are no tasks for the system. In the following chapter we evaluate the effectiveness of this *scale-to-zero* behavior and which side-effects it has.

Listing 6 – ScaledObject CRD for KEDA autoscaling

```
apiVersion: keda.sh/v1alpha1
kind: ScaledObject
metadata:
  name: exec-so
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: executor
  pollingInterval: 15
  cooldownPeriod: 60
  maxReplicaCount: 50
  triggers:
  - type: rabbitmq
    metadata:
      protocol: http
      host: 'http://user:password@rabbitmq.namespace.svc:15672/'
      queueName: executorTasks
      mode: QueueLength
      value: '1'
```

5 Evaluation

"*Sometimes magic is just someone spending more time on something than anyone else might reasonably expect.*"
— Raymond Joseph Teller

The previous chapter documented the necessary steps for setting up a metric collection system and implementing an autoscaling mechanism on top of it. This chapter focuses on a quantitative evaluation of the performance and cost improvements made by autoscaling. Our findings demonstrate that the target application is able to achieve maximum performance with the autoscaling policies, while having only minor variances in performance. At the same time, significant cost-savings (more than 19%) can be realized thanks to downscaling during times of low load. We follow the guidelines on scientific benchmarking by Hoefler and Belli [51] to ensure reproducibility and interpretability of our results.

As outlined in the introduction, autoscaling always needs to make a trade-off between optimizing for application performance and optimizing for cost, i.e., allocated resources. If cost is not an issue, one could simply allocate a fixed, large number of resources and always keep those running — known as dimensioning for peak load. However, this is not cost-effective, especially not in a world where cloud resources (such as virtual machines and containers) can be allocated and are billed by the second. Thus, the goal is to always keep the number of allocated resources as low as possible — with some reasonable safety margin to compensate unforeseen deviations.

5.1 Benchmark Setup

To measure the performance of the system we use one of the application-level metrics we previously exposed in Section 4.2: the duration of configuration runs. A configuration run refers to the configuration of a fixed set of systems being checked and updated. This process is commonly triggered by the user through a web interface, thus it is deemed a relevant metric for measuring the performance of the application.

To measure the cost of horizontal scaling we use *replica seconds*: the number of running Pods per second integrated over the time period of the benchmark. For example, if the benchmark lasts one minute and two replicas are running the entire time, this would result in 120 replica seconds. Prometheus is not suitable for such time-accurate measurements [48], therefore we implement our own tool that fetches this data directly from the Kubernetes API.

We set up a performance benchmark for the application with a constant workload size, which allows us to measure the cost-reductions that can be achieved through autoscaling. Thus, our experiment evaluates the *strong scaling* behavior [51] of the application and – by extension – the autoscaler. In the context of computing performance, strong scaling refers to the addition of more processors while the problem size is constant. An individual benchmark run is structured as follows:

1. Set up and initialize target application from scratch.
2. Start the benchmark timer.
3. After 60 seconds, start a configuration run. Sequentially repeat ten times.
4. Wait until all configuration runs finish (this marks the *time to completion*).
5. Continue running the application for 30 minutes after starting the benchmark timer. Then, stop the *replica seconds* counter.
6. Terminate the application.

The entire benchmark procedure is repeated three times. We verified the benchmark results do not contain outliers or other inconsistencies. Between each benchmark the target application is completely removed from the cluster (the Kubernetes namespace is deleted) and re-installed into a new namespace. This ensures the benchmark runs are completely isolated and no transient side effects (such as warm caches) are present. The setup phase described above is entirely scripted to minimize potential for variation.

Configuration runs are started exactly 60 seconds apart from each other. Since they take longer than 60 seconds (see Figure 13), the application needs to process multiple configuration runs in parallel. Each configuration run contains the same amount of work. The *time to completion* marks the point when all configuration runs have finished processing. The application continues to run afterwards (until a fixed timeout) to demonstrate the cost-savings made possible by autoscaling. If the benchmark was stopped as soon as the workload was completed, only the performance (but not the cost) could be compared. Because of the fixed benchmark duration, the cost-savings can be extrapolated to one day, week or month.

The benchmarks are performed on a Kubernetes cluster consisting of one control plane node and two worker nodes. The worker nodes have a combined capacity of 136 CPU cores and 1134 GiB memory.

5.2 Functional Verification

This section focuses on testing the functionality of the autoscaling setup, as there are many components involved and several parameters to be tweaked. For this reason an artificial task (`openssl speed`) is given to the executor component of the target application (see Chapter 4). This allows us to iterate quickly and test out different configuration settings and scaling metrics. In Section 5.4 we set up a production-like environment for performing the benchmarks with a real workload, which connects to and configures external systems.

As explained at the beginning of Chapter 4, the number of executors in the target application can be adjusted. To get a baseline for the application behavior, we tested several static values for replicas. The graph in Figure 12 shows the results with the cost (replica seconds) on the x-axis and the performance (time to completion, i.e., time until the simulated user has all desired results) on the y-axis. Naturally, the lowest cost (4.000 replica seconds) is achieved when using 2 replicas, which also has by far the largest time to completion (2.000 seconds). The highest cost (14.000 replica seconds) is recorded with 20 replicas, which also has the lowest time to

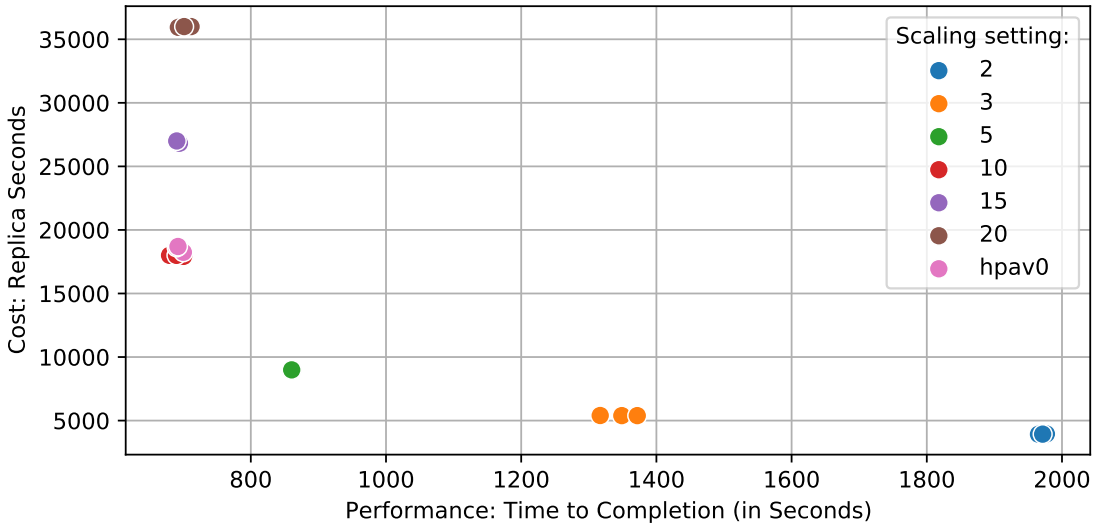

Figure 12 – Results of scaling benchmark. Scaling setting indicates static number of replicas or autoscaling policy.

completion (4.000 seconds). Due to the constant workload size, the benchmarks with 10, 15 and 20 replicas have almost the same time to completion, while having significantly more replicas and therefore replica seconds. This means that for this workload size, more than 10 replicas are inefficient, since most of the replicas are idle (unused) during the execution. In this scenario, the goal of the autoscaler is to optimize towards the bottom left corner (low cost and high performance), irrespective of the type and size of the given workload.

Data from the monitoring system shows that with 3 replicas the average queue duration (time before tasks are actually executed) continuously rises during the benchmark up to a value of 360 seconds. With 20 replicas, the average queue time quickly converges to a value of 30 seconds. This highlights the effect of not having enough executor replicas available which leads to significant delays, since each executor may only process one task at a time.

Figure 13 shows the execution time of individual configuration runs during the benchmark. It confirms our intuition that with an overall lower time to completion (as it was the case in Figure 12), the time of individual workloads is also smaller. Furthermore, it shows that a low number of replicas has a significant effect on the variance of the configuration run's execution times. While each configuration run has the same workload, with a low number of replicas a significant increase in variance (in addition to an increase of the average) can be observed. This is explained by the fact that with few replicas, the same executor needs to run multiple workloads sequentially, which slows some of them down drastically.

After establishing the performance and cost characteristics of static configurations, we evaluate the characteristics of a basic autoscaling setup. The first version of the autoscaling policy was shown in Listing 5. We now evaluate the behavior of this policy, which is labeled as `hpav0` in the figures. As Figure 12 shows, the benchmark

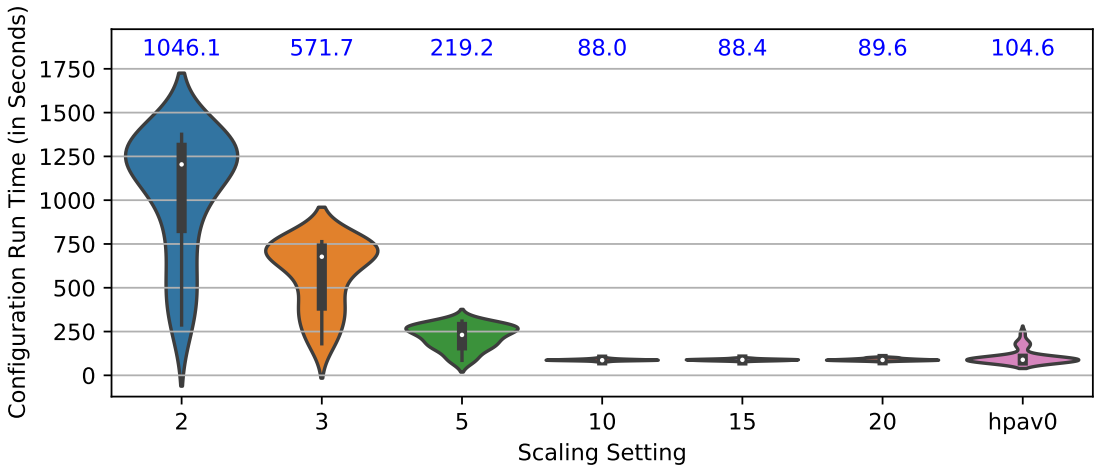

Figure 13 – Execution time of individual configuration runs (3 benchmarks, 10 configuration runs per benchmark). Numbers in blue indicate the mean value.

with this policy had a similar performance and cost as a static configuration with 10 replicas: the application used around 18.600 replica seconds during the benchmark and the total time to completion was approx. 700 seconds.

This result highlights the strengths and weaknesses of this autoscaling policy: it enabled the application to achieve the same performance as with a static configuration of 10 replicas. As Figure 12 shows, this is the maximum amount of performance the application is able to deliver for this particular workload. At the same time, the autoscaling policy was just as costly as a static configuration of 10 replicas, even though for significant periods the application was running with only 1 replica. This is explained by Figure 11: the policy overscaled the number of replicas (more than the necessary value established previously) and even reached the replica limit (`maxReplicas` from Listing 5).

Furthermore, the variance in execution time between different configuration runs was slightly larger than with 10 replicas. This can be explained by the fact that the autoscaler gradually needs to scale up the Deployment at the beginning of the benchmark. Thus, the executor Deployment does not have the optimal resources immediately available and some tasks need to wait longer in the queue. This behavior can be observed in Figure 11.

In summary, the autoscaling policy has performed well (no performance loss, minor introduction of variance), but has not realized any cost-savings in our experiments due to drastic overscaling (provisioning more replicas than required for the workload).

5.3 Cost Optimization

While the initial scaling policy successfully scaled the Deployment during the benchmark, we identified several aspects for improvement. This section addresses these issues by fine-tuning the scaling policy.

- **Delayed scale-down**: the number of replicas is not reduced soon enough after the workload has finished, as is apparent from the number of queued tasks compared to the number of replicas (Figure 11).
- **Potential for premature scale-down**: if a task has a long execution time, the autoscaler might reduce the number of replicas too early because the scaling metric only depends on queued tasks.
- **Overscaling of replicas**: our previous experiments with static replica configurations have shown that provisioning 20 replicas is ineffective, since it does not increase performance (as shown in Figure 12).

The delayed scale-down can be tackled by adjusting the *stabilization window* of the scaling policies (also referred to as *cool-down period*). This setting (shown in Listing 7) specifies how soon HPA starts removing replicas from the Deployment after it detects that the scaling metric is below the target value [28]. The default value is 5 minutes (observable in Figure 11); we adjust the value to 1 minute (line 7). Decreasing the downscale stabilization window can lead to thrashing (continuous creation and removal of Pods) when workload bursts are more than the specified window apart, but offers better elasticity [10]. In our case this is an acceptable trade-off because these containers start quickly, as this application component is lightweight and does not hold any internal state. Additionally, this risk is partially mitigated by only allowing HPA to remove 50% of the active replicas per minute (Listing 7, line 4-6). By default, HPA is allowed to deprovision all Pods at the same time [28], as it is illustrated in Figure 11 (rapid decrease from 20 to 1 replica).

Listing 7 – Improved Downscale Behavior for HPA

```
1   behavior:
2     scaleDown:
3       policies:
4         - type: Percent
5           value: 50
6           periodSeconds: 60
7       stabilizationWindowSeconds: 60
```

The potential for premature scale-down is reduced by not only considering the queued tasks as a scaling metric, but also the currently running tasks. Just because all tasks have been taken out of the message queue by the executors does not mean that the number of executors can be reduced, as they might still be processing the tasks. For this purpose, HPA allows specifying multiple metrics (Listing 8): it calculates the desired replica count for all specified metrics individually and then scales the Deployment based on the maximum results.

Finally, the issue of overscaling replicas can be mitigating by switching to a different baseline scaling metric, shown in Listing 8 (line 5). This metric immediately

represents the number of tasks available for the executor, unlike all future tasks as before. This distinction is important because future tasks might have interdependencies (e.g., if task #1 fails, task #2 and #3 does not need to be executed). Additionally, until now we have been using an absolute value as a target, e.g., the total number of tasks in queue. It makes more sense to use a metric that incorporates the current number of replicas as a ratio. This is necessary because – with the change explained previously – the scaling metric contains the number of available tasks, which are by definition distributed across all executors. Instead of using the scaling metric directly, the raw value is averaged: it is divided by the number of active Pod replicas (Listing 8, line 7-8) [28].

Listing 8 – HPA scaling based on multiple metrics

```
1   metrics:
2   - type: External
3     external:
4       metric:
5         name: esm_tasks_queued_total
6       target:
7         type: AverageValue
8         averageValue: 1
9   - type: External
10    external:
11      metric:
12        name: esm_configuration_runs_processing
13      target:
14        type: AverageValue
15        averageValue: 1
```

To find the appropriate value for `averageValue`, we perform several benchmarks, the results of which are shown Figure 14 to 17. The different `averageValues` have been labeled as `hpav1`, `hpav2`, `hpav3` and `hpa4` for the values 1, 2, 3, and 4, respectively. These new benchmarks also include the other optimizations outlined above. For comparison, the following figures also show the previously discussed benchmark results of the initial autoscaling policy (`hpav0`) and a statically scaled Deployment with 5 and 10 replicas.

Due to the downscaling optimization outlined above, all of the scaling policies were able to reduce the cost (Figure 14) by 50% as they allow scaling the Deployment down sooner. This behavior is illustrated in Figure 15. Additional cost savings can be realized by reducing the replica count to 0 when there are no tasks to be processed. In our application architecture this is feasible because the executors are taking the tasks out of a message queue, which acts as a buffer when no executors are available (yet). However, HPA does not support this by default [28]: setting `minReplicas` field (see Figure 11) to 0 is only possible when changing the configuration of the Controller Manager, which requires re-deploying the cluster. KEDA works around this HPA limitation by having an agent that scales the Deployment to zero when no tasks are active.

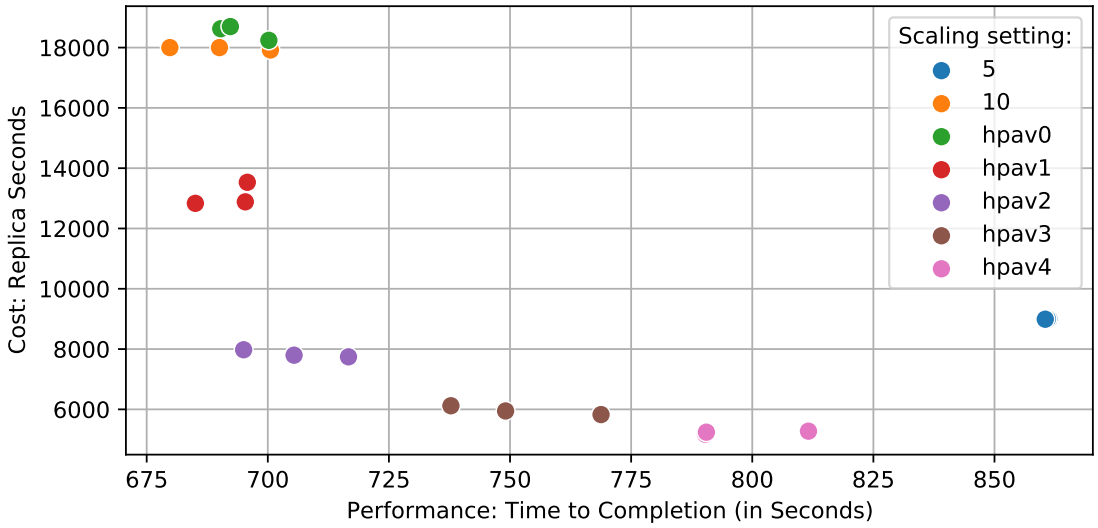

Figure 14 – Results of autoscaling benchmark. Scaling setting indicates static number of replicas or autoscaling policy.

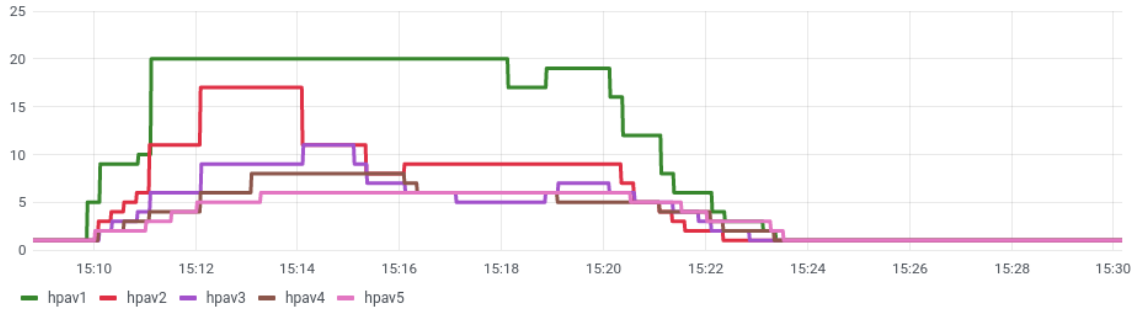

Figure 15 – Grafana screenshot of different horizontal scaling policies. Y-axis represents the number of active replicas, X-axis represents time.

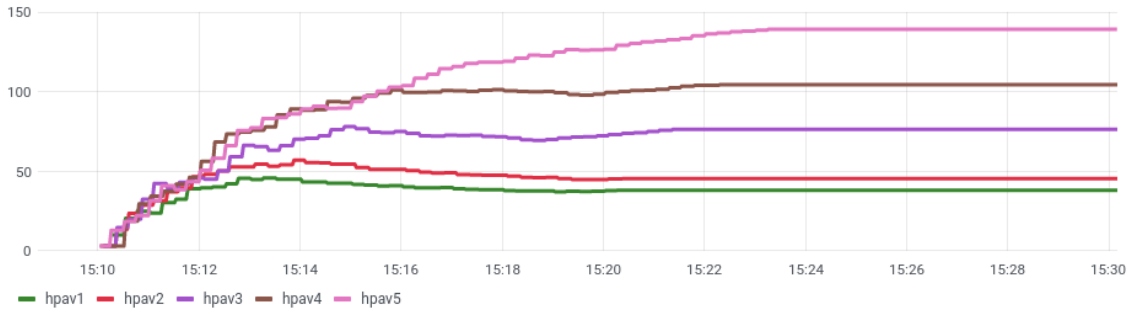

Figure 16 – Grafana screenshot of queuing behavior with different autoscaling policies. Y-axis shows average time tasks are queued in seconds.

Despite the significant cost reduction, all scaling policies performed nearly as well as the fastest configuration (with 10 replicas). This establishes the effectiveness of the autoscaling policies, as confirmed by comparing the average time tasks spend in queue (Figure 16): there is a gradual increase from 38.1 seconds with `hpav1`, 45.4 seconds with `hpav2`, 76.4 seconds with `hpav3` to 104.0 seconds with `hpav4`. A comparison against static dimensioning shows that the values are quite stable (i.e., the average value is not continuously rising) and are almost as low as the best performing static scaling configuration (30 seconds with 20 replicas). Therefore, different parameters for `averageValue` can be used to tweak the trade-off between application performance and cost. With our specific test scenario a value of one provides excellent performance while already realizing major cost savings. Clearly, different user preferences (performance-cost trade-off) will require different values.

Figure 17 shows the impact of the `averageValue` parameter on the duration of individual configuration runs. A larger value effectively allows more tasks to be waiting in the queue without triggering any scaling. Since configuration runs are made up of many individual tasks, the waiting time of these tasks affects the duration of the entire configuration run. All of the autoscaling policies exhibit higher variance in configuration run durations than a static replica number of 10. Between the autoscaling policies, the `averageValue` does not seem to have a major impact on variance.

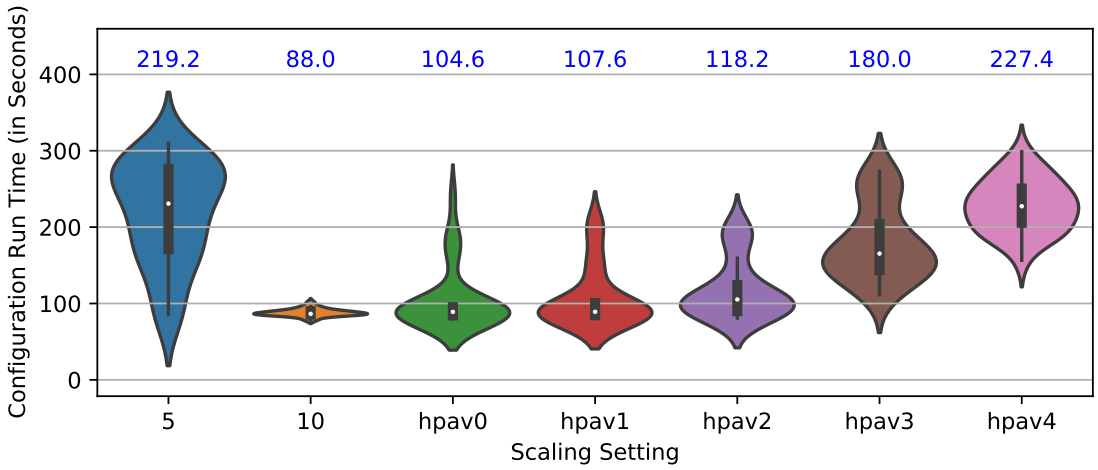

Figure 17 – Execution time of individual configuration runs. Numbers at the top (in blue) indicate the mean value.

5.4 Real-world test scenario

The previous sections validated the functionality of the autoscaling setup as well as evaluated several scaling metrics and parameters based on an artificial workload. In this section we set up production-like target systems, consisting of 25 virtual machines, and configure the application to connect to these systems via SSH. In addition, the configuration scripts used to interact with the systems are representative of tasks carried out in production environments: each script consists of 42 checks for system security settings such as administrator access, login retry interval etc.

We repeat the benchmark scenario described in Section 5.1 with adjustments for the production-like scenario: the overall benchmark time (75 minutes), number of configuration runs (50) and maximum replicas (50) are increased to accommodate workload. Thus, the results from Section 5.1 cannot be compared in absolute terms to the results presented in these sections, though we expect to confirm the trends from our previous findings.

Appendix A.5 (Listing 16) shows the full horizontal autoscaling policy used in HPA benchmarks. The KEDA benchmarks use the scaling policy shown in Listing 6.

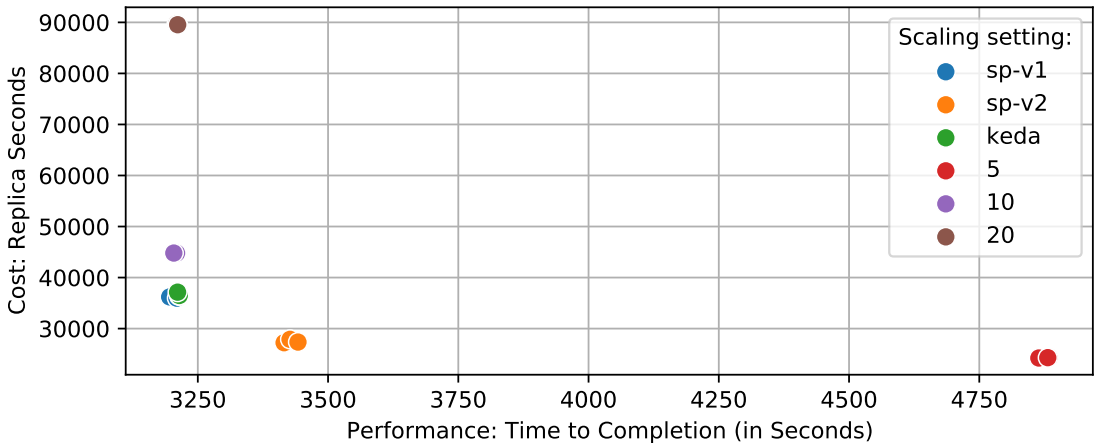

Figure 18 – Results of autoscaling benchmark in production-like scenario with constant workload. Scaling setting indicates static number of replicas or autoscaling policy.

The benchmark results shown in Figure 18 confirm our previous experiments: the autoscaling policy `sp-v1` (blue) as well as KEDA (green) achieve the same performance as a static replica number of 10. This is the maximum performance the application is able to achieve in this scenario, because even with higher replica counts (e.g., 20 in Figure 18) the performance remains the same. At the same time, all scaling policies are able to consistently reduce the cost during the benchmark: `sp-v1` has 19.3% lower cost and `keda` has 18.1% lower cost while maintaining the same performance as 10 replicas. The same performance is explained by the fact that internally KEDA uses HPA for autoscaling and in this case the same target metric and value was specified for KEDA and HPA. Logically, the `keda` autoscaling

policy should have a lower cost than `sp-v1` because KEDA has the ability to scale the Deployment down to zero replicas (as opposed to `sp-v1` which has a minimum of one replica). These cost savings did not manifest themselves in the benchmarks because most of the time there is load on the system. Scaling the Deployment to zero has larger benefits when there are significant periods where a particular service is completely idle. `sp-v2` has 38.6% lower cost while having worse performance than 10 replicas. This is due to the fact that `sp-v2` allows more tasks to be in queue compared to `sp-v1` and `keda`, thereby increasing the time to completion.

Looking at the execution time of individual configuration runs (Figure 19), we see that the autoscaling policies have a higher variance compared to a static over-provisioning of resources. While the means of `sp-v1` and `keda` are just slightly elevated compared to the result of 10 static replicas, the mean of `sp-v2` is double that of `sp-v1`. Logically this is correct because scaling policy `sp-v2` allows twice the number tasks in queue. Additionally, we can also see that the variance of `keda` is slightly higher than that of `sp-v1`. This can be explained by the additional latency that KEDA has when scaling the Deployment up from zero to one replica.

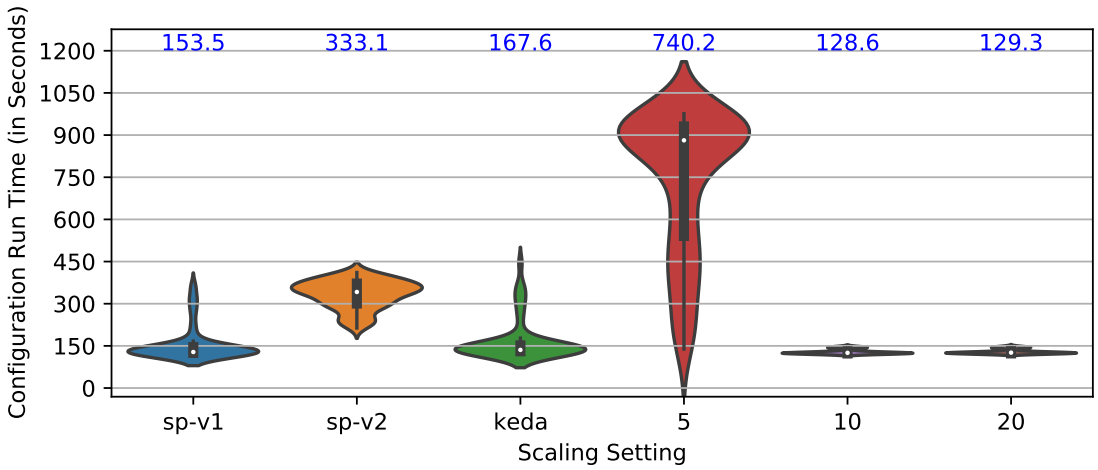

Figure 19 – Execution time of individual configuration runs in production-like scenario with constant workload. Numbers at the top (in blue) indicate the mean value.

Exemplary behaviors of the autoscaling policies during the benchmark are shown in Figure 20. In blue it shows the replica count of `sp-v1`, orange the replica count of `sp-v2` and green the replica count of `keda`. The red peaks indicate when new work has been submitted to the system (their height does not have any significance). From this example it is clear that scaling policy `sp-v2` suffers from *thrashing*: scaling operations are frequently made and then reverted shortly afterwards again. `sp-v1` exhibits a much smoother line, which is expected with a constant workload, and takes 13 minutes to converge to a stable number of replicas (11). Similarly, the scale up of `keda` was slightly delayed and converged to the same number of replicas after 18 minutes. Overall, `sp-v2` scales the Deployment to a slightly lower number (and thereby higher cost, Figure 18), as it allows for more tasks to be in queue.

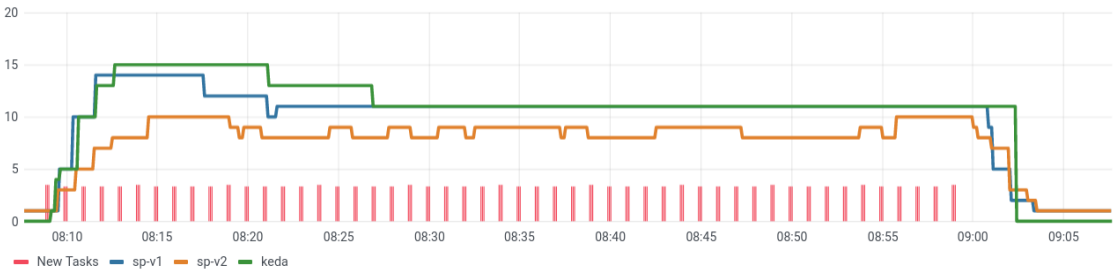

Figure 20 – Grafana screenshot of scaling activity during production-like scenario with constant workload. Red bars indicate the launch of new configuration runs.

In general, the results in Figure 18, 19 and 20 show that the monitoring systems, autoscalers and scaling policies described in the previous sections are able to scale the executor Deployment with satisfying results. The scaling policy is able to maintain overall application performance (the time until the user is shown the result) while reducing the cost (amount of allocated cluster resources) and also incurs a minor penalty in the variance of the duration of individual configuration runs.

In addition, we want to investigate how these autoscaling policies behave under a highly variable workload. For this purpose, the workload of individual configuration runs is increased and their schedule is adjusted: eight configuration runs are launched (in one minute intervals), followed by a three minute period without new work. This pattern is repeated five times, resulting in a total of 40 configuration runs. This scenario tests the elasticity of the autoscaling polices.

Naturally, the results of these experiments are more variable than the previous ones. Figure 21 shows the performance-cost trade-off and in particular the lower cost achieved by the autoscaling policies, while almost reaching the maximum performance (total time to completion). This illustrates that the autoscaling policies `sp-v1` and `keda` are able to adjust to the varying workload and deliver competitive results. However, Figure 22 highlights a minor flaw: on average individual configuration runs are much slower compared to the best case (10 static replicas) and there is a large variance in their durations (even though each configuration run contains the same amount of work). We believe this is at least partly due to the frequent scaling operations required during this benchmark (Figure 23). In the scenario with varying workload, especially `sp-v1` and `keda` had trouble converging to a stable number of replicas (compare blue and green lines in Figure 20 to 23). As discussed previously, a trade-off has to be made between stability of the system (in this case less scaling operations by increasing the cool-down period) and agility (more frequent scaling operations allow for more cost-savings).

One limitation that is not reflected in our test scenario is the *noisy neighbor* problem. While CPU and memory resources can be isolated with Kubernetes by using resource requests (Section 2.3.1), this is not supported for storage I/O and network resources [52]. Thus, when creating more replicas of a Pod, it is possible that these instances compete for the same shared, non-isolated resources, and subsequently fail to achieve maximum performance. In our benchmarks we minimized the effect of this issue by provisioning enough cluster capacity of these resources.

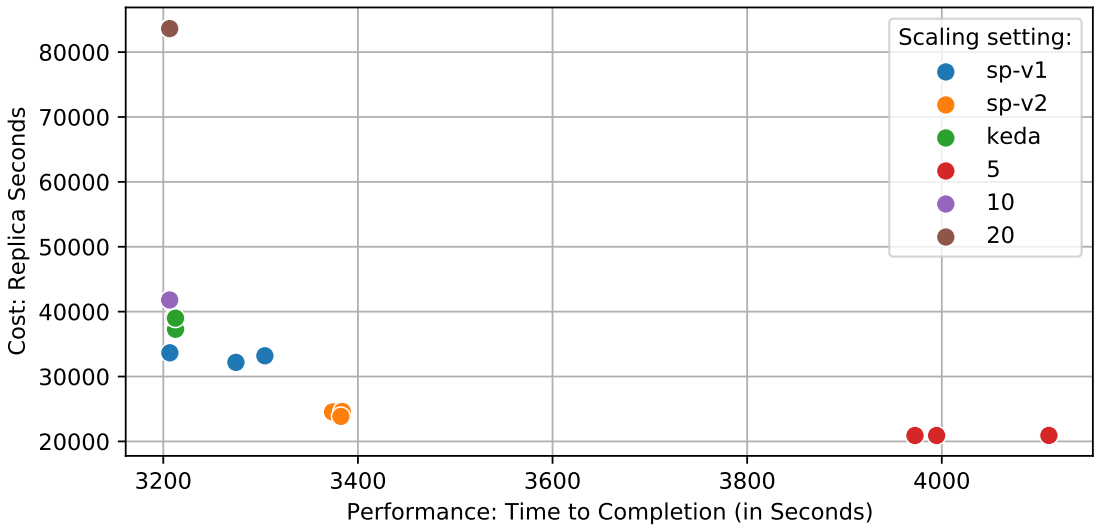

Figure 21 – Results of autoscaling benchmark with varying workload. Scaling setting indicates static number of replicas or autoscaling policy.

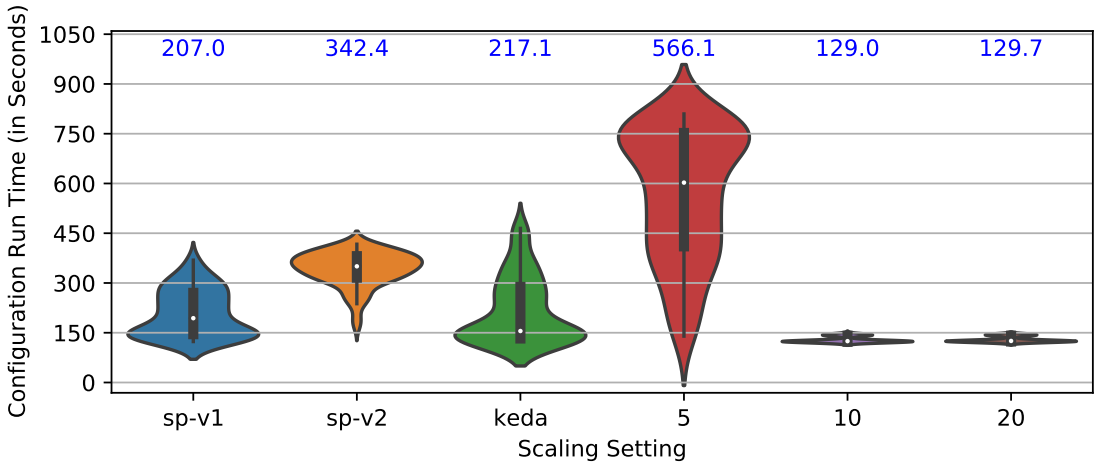

Figure 22 – Execution time of individual configuration runs with varying workload. Numbers at the top (in blue) indicate the mean value.

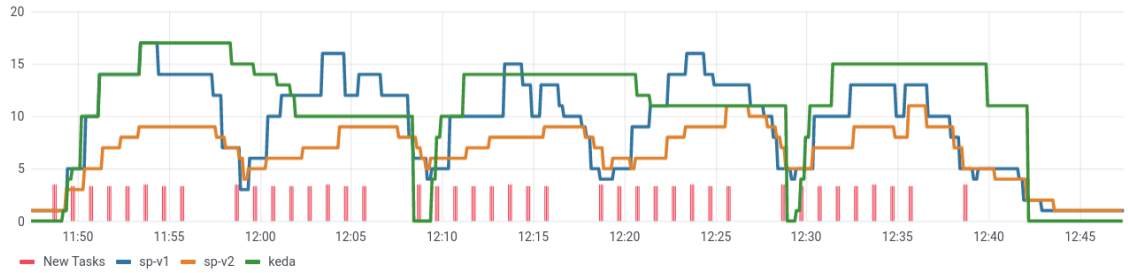

Figure 23 – Grafana screenshot of scaling activity during production-like scenario with varying workload. Red bars indicate launch of new configuration runs.

6 Summary and Future Work

"While developing these systems we have learned almost as many things not to do as ideas that are worth doing."
— Burns et al. [7]

This thesis tackled the questions of how to effectively dimension cloud-native applications and how to assess their performance. While modern cloud platforms allow developers to allocate arbitrary amounts of resources, operating a production-grade service on Kubernetes requires deep insights into the performance behavior of the application [6]. Once this question had been answered, we moved on to the challenge of scaling our application based on the current workload, i.e., *autoscaling*.

We gave an overview of the available literature on the subject of autoscaling applications in the cloud. This revealed that while there have been numerous articles and surveys about VM- and container-based autoscaling, only recently have researchers started investigating specifically Kubernetes. A comprehensive review of the algorithms and technical architectures of publicly available autoscaling components for Kubernetes (HPA, VPA, CA, KEDA) was performed to understand the technologies currently used in the industry. Finally, a survey of research proposals for novel Kubernetes autoscalers was conducted and the proposals were evaluated qualitatively. This research made it clear that proactive autoscaling (i.e., scaling not only based on current load, but based on predicted future load) is beneficial for aggressive scaling. However, this leads to more complex algorithms (which require more time to train and potentially large amounts of data) as well as system behavior that is more opaque to cluster operators. Thus, these two aspects need to be balanced.

The literature is inconclusive about whether a service should be scaled based on low-level (e.g., CPU and memory utilization) or high-level metrics (such as response time). Ultimately, the choice of scaling metrics depends on the development context and application usage scenario. For this purpose, our work outlined the necessary steps to expose and identify metrics relevant for scaling an application running on Kubernetes.

Unfortunately, none of the reviewed proposals have a publicly available implementation. This is problematic because it prevents evaluating the technical soundness of the implementation and its integration with Kubernetes. In the end, not only the underlying algorithms are important when setting up a production-grade system, but also how the operators need to configure and interact with it.

For this reason, we proposed the design and architecture of a novel Kubernetes autoscaler: its main characteristic is modularity by using a WebAssembly sandbox for running the core scaling algorithm. The modular autoscaler gives cluster operators safety and reliability guarantees. At the same time, it offers flexibility and ease-of-use to researchers looking to implement and test their scaling algorithms. The implementation of this Kubernetes autoscaling component is left as future work.

We then presented the necessary monitoring infrastructure and autoscaling policies for an application in a production-grade environment. We believe that this is highly relevant for industry practitioners looking to get started with monitoring an application running on Kubernetes. Prometheus was chosen as a monitoring tool as it is the industry-standard for metrics-based monitoring in the cloud. Grafana was used as a visualization layer to get an intuition for the behavior and correlation of metrics from different system components. The discussion then provided a reference point for which kinds of metrics should be collected by monitoring system to allow the operators to have a complete picture of the application behavior: low-level metrics (e.g., CPU and memory usage), high-level metrics (e.g., response time), platform-level metrics (e.g., Kubernetes Pods), service-level metrics (e.g., message queue status) and application-level metrics (e.g., number of users). Additionally, we described how to identify metrics relevant for scaling and how to configure Kubernetes autoscaling components (HPA, VPA, KEDA) based on these metrics. While the implementation we have shown is specific to the target application, the principles and methodologies can be applied to any cloud-native application. Since we provided detailed documentation about our setup, industry professionals and researchers are able to replicate similar setups in their own environments.

Finally, we performed a quantitative evaluation of several autoscaling policies. Our findings showed that the target application is able to achieve maximum performance with the autoscaling policies, while having only minor variances in performance. At the same time, we achieved significant cost-savings due to downscaling during times of low load. Despite the benchmark results being specific to our target application, other researchers and professionals can reuse the same benchmarking procedures for any queue-based cloud application. Furthermore, the discussed scaling optimizations (delayed scale-down, overscaling etc.) are applicable to any system leveraging autoscaling. In particular, the criteria for evaluating the performance (*time to completion*) and cost (*replica seconds*) dimensions are valuable for anyone carrying out performance-and-cost optimizations with container-based infrastructure.

Concerning future work, we think it would be valuable to compare the current implementation of the target application with an event-driven implementation. Event-driven architectures are at the core of popular *serverless* or *functions-as-a-service* offerings of public cloud providers (e.g., AWS Lambda, GCP Cloud Run, Azure Functions). In this architecture there are no constantly running workers that are listening. Instead, for each *event* (e.g., a task in a message queue) a new instance of the worker is created and once the worker finishes processing the task, the worker is terminated. This frequent creation and termination of workers is made possible by running stateless services with extremely lightweight isolation. Subsequently, such an architecture has the potential for even more elasticity. Mohanty et al. [53] published a survey about open-source event-driven platforms, which can be used as a basis for this future work. More recently, the SPEC research group conducted a general review of use cases for serverless architectures [54].

Overall, this thesis provided foundational and relevant knowledge on the topic of autoscaling for researchers and industry practitioners alike. While not all software

architectures and deployment models were discussed in our work (in particular stateful applications), the reader should have gained insights into tackling the challenging tasks of dimensioning, optimizing and scaling their cloud-native applications. The key takeaway is that a solid foundation of metrics (collected from several components) allows effectively dimensioning and scaling any application with state-of-the-art cloud-native solutions.

References

[1] P. Mell and T. Grance, "The NIST definition of cloud computing," pp. NIST SP 800–145. Available online: https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-145.pdf

[2] S. Newman, *Building Microservices: Designing Fine-Grained Systems*, 2nd ed. O'Reilly Media. Available online: https://go.oreilly.com/university-college-london/library/view/-/9781492034018/

[3] R. Vitillo, *Understanding Distributed Systems*, 1st ed. Available online: https://understandingdistributed.systems/

[4] C. Qu, R. N. Calheiros, and R. Buyya, "Auto-Scaling Web Applications in Clouds: A Taxonomy and Survey," *ACM Computing Surveys*, vol. 51, no. 4, pp. 1–33. Available online: https://dl.acm.org/doi/10.1145/3148149

[5] G. Yu, P. Chen, and Z. Zheng, "Microscaler: Automatic Scaling for Microservices with an Online Learning Approach," in *2019 IEEE International Conference on Web Services (ICWS)*. IEEE, pp. 68–75. Available online: https://ieeexplore.ieee.org/document/8818401/

[6] N. Herbst, R. Krebs, G. Oikonomou, G. Kousiouri, A. Evangelinou, A. Iosup, and S. Kounev, "Ready for Rain? A View from SPEC Research on the Future of Cloud Metrics," p. 43. Available online: https://research.spec.org/fileadmin/user_upload/documents/rg_cloud/endorsed_publications/SPEC-RG-2016-01_CloudMetrics.pdf

[7] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, Omega, and Kubernetes," *Communications of the ACM*, vol. 59, no. 5, pp. 50–57. Available online: https://dl.acm.org/doi/10.1145/2890784

[8] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and Linux containers," in *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 171–172.

[9] R. Morabito, J. Kjällman, and M. Komu, "Hypervisors vs. Lightweight Virtualization: A Performance Comparison," in *2015 IEEE International Conference on Cloud Engineering*, pp. 386–393.

[10] F. Zhang, X. Tang, X. Li, S. U. Khan, and Z. Li, "Quantifying cloud elasticity with container-based autoscaling," *Future Generation Computer Systems*, vol. 98, pp. 672–681. Available online: https://linkinghub.elsevier.com/retrieve/pii/S0167739X18307842

[11] E. Casalicchio and V. Perciballi, "Auto-Scaling of Containers: The Impact of Relative and Absolute Metrics," in *2017 IEEE 2nd International Workshops on*

Foundations and Applications of Self Systems (FAS*W)*. IEEE, pp. 207–214. Available online: http://ieeexplore.ieee.org/document/8064125/

[12] Datadog. 8 surprising facts about real Docker adoption. 8 surprising facts about real Docker adoption. Available online: https://www.datadoghq.com/docker-adoption/ (Accessed 2021-03-02).

[13] S. Taherizadeh and M. Grobelnik, "Key influencing factors of the Kubernetes auto-scaler for computing-intensive microservice-native cloud-based applications," *Advances in Engineering Software*, vol. 140, p. 102734. Available online: https://linkinghub.elsevier.com/retrieve/pii/S0965997819304375

[14] J. Kephart and D. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, pp. 41–50. Available online: http://ieeexplore.ieee.org/document/1160055/

[15] The Kubernetes Authors. Kubernetes Documentation: Workloads. Available online: https://v1-20.docs.kubernetes.io/docs/concepts/workloads/ (Accessed 2021-04-13).

[16] The Kubernetes Authors. Kubernetes Documentation: Services. Available online: https://v1-20.docs.kubernetes.io/docs/concepts/services-networking/service/ (Accessed 2021-04-13).

[17] B. Ibryam and R. Huß, *Kubernetes Patterns*, 1st ed. O'Reilly Media. Available online: https://k8spatterns.io/

[18] The Kubernetes Authors. Kubernetes Documentation: Components. Available online: https://v1-20.docs.kubernetes.io/docs/concepts/overview/components/ (Accessed 2021-04-13).

[19] K. Rzadca, P. Findeisen, J. Swiderski, P. Zych, P. Broniek, J. Kusmierek, P. Nowak, B. Strack, P. Witusowski, S. Hand, and J. Wilkes, "Autopilot: Workload autoscaling at Google," in *Proceedings of the Fifteenth European Conference on Computer Systems*. ACM, pp. 1–16. Available online: https://dl.acm.org/doi/10.1145/3342195.3387524

[20] M. Kaboudan, "A dynamic-server queuing simulation," *Computers & Operations Research*, vol. 25, no. 6, pp. 431–439. Available online: https://linkinghub.elsevier.com/retrieve/pii/S0305054897000907

[21] G. Chen, W. He, J. Liu, S. Nath, L. Rigas, L. Xiao, and F. Zhao, "Energy-Aware Server Provisioning and Load Dispatching for Connection-Intensive Internet Services," ser. NSDI'08. USENIX Association, pp. 337–350. Available online: https://www.usenix.org/legacy/event/nsdi08/tech/full_papers/chen/chen.pdf

[22] Y.-h. Jia, L.-x. Tang, Z. G. Zhang, and X.-f. Chen, "MMPP/M/C queue with congestion-based staffing policy and applications in operations of steel industry," *Journal of Iron and Steel Research International*, vol. 26, no. 7, pp. 659–668. Available online: http://link.springer.com/10.1007/s42243-018-0151-y

[23] T. Lorido-Botran, J. Miguel-Alonso, and J. A. Lozano, "A Review of Auto-scaling Techniques for Elastic Applications in Cloud Environments," *Journal of Grid Computing*, vol. 12, no. 4, pp. 559–592. Available online: http://link.springer.com/10.1007/s10723-014-9314-7

[24] G. Galante, L. C. Erpen De Bona, A. R. Mury, B. Schulze, and R. da Rosa Righi, "An Analysis of Public Clouds Elasticity in the Execution of Scientific Applications: A Survey," *Journal of Grid Computing*, vol. 14, no. 2, pp. 193–216. Available online: http://link.springer.com/10.1007/s10723-016-9361-3

[25] A. R. Hummaida, N. W. Paton, and R. Sakellariou, "Adaptation in cloud resource configuration: A survey," *Journal of Cloud Computing*, vol. 5, no. 1, p. 7. Available online: http://journalofcloudcomputing.springeropen.com/articles/10.1186/s13677-016-0057-9

[26] Y. Al-Dhuraibi, F. Paraiso, N. Djarallah, and P. Merle, "Elasticity in Cloud Computing: State of the Art and Research Challenges," *IEEE Transactions on Services Computing*, vol. 11, no. 2, pp. 430–447. Available online: https://ieeexplore.ieee.org/document/7937885/

[27] E. Radhika and G. Sudha Sadasivam, "A review on prediction based autoscaling techniques for heterogeneous applications in cloud environment," *Materials Today: Proceedings*, p. S2214785320394657. Available online: https://linkinghub.elsevier.com/retrieve/pii/S2214785320394657

[28] The Kubernetes Authors. Kubernetes Documentation: Horizontal Pod Autoscaler. Available online: https://v1-20.docs.kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/ (Accessed 2021-04-13).

[29] L. Toka, G. Dobreff, B. Fodor, and B. Sonkoly, "Adaptive AI-based auto-scaling for Kubernetes," in *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*. IEEE, pp. 599–608. Available online: https://ieeexplore.ieee.org/document/9139654/

[30] P. Versockas. Vertical Pod Autoscaling: The Definitive Guide. Available online: https://povilasv.me/vertical-pod-autoscaling-the-definitive-guide/ (Accessed 2021-04-13).

[31] G. Rattihalli, M. Govindaraju, H. Lu, and D. Tiwari, "Exploring Potential for Non-Disruptive Vertical Auto Scaling and Resource Estimation in Kubernetes," in *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*. IEEE, pp. 33–40. Available online: https://ieeexplore.ieee.org/document/8814504/

[32] KEDA Authors. The KEDA Documentation (Version 2.2). KEDA. Available online: https://keda.sh/docs/2.2/ (Accessed 2021-05-21).

[33] M. Tamiru, J. Tordsson, E. Elmroth, and G. Pierre, "An Experimental Evaluation of the Kubernetes Cluster Autoscaler in the Cloud," in *12th IEEE International Conference on Cloud Computing Technology and Science*, p. 10.

[34] Q. Wu, J. Yu, L. Lu, S. Qian, and G. Xue, "Dynamically Adjusting Scale of a Kubernetes Cluster under QoS Guarantee," in *2019 IEEE 25th International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE, pp. 193–200. Available online: https://ieeexplore.ieee.org/document/8975761/

[35] V. Medel, C. Tolon, U. Arronategui, R. Tolosana-Calasanz, J. A. Banares, and O. F. Rana, "Client-Side Scheduling Based on Application Characterization on Kubernetes," in *Economics of Grids, Clouds, Systems, and Services*, ser. Lecture Notes in Computer Science. Springer International Publishing, vol. 10537, pp. 162–176. Available online: http://link.springer.com/10.1007/978-3-319-68066-8_13

[36] U. Deshpande, "Caravel: Burst Tolerant Scheduling for Containerized Stateful Applications," in *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, pp. 1432–1442. Available online: https://ieeexplore.ieee.org/document/8885293/

[37] P. Townend, S. Clement, D. Burdett, R. Yang, J. Shaw, B. Slater, and J. Xu, "Improving Data Center Efficiency Through Holistic Scheduling In Kubernetes," in *2019 IEEE International Conference on Service-Oriented System Engineering (SOSE)*. IEEE, pp. 156–15 610. Available online: https://ieeexplore.ieee.org/document/8705815/

[38] E. Casalicchio, "A study on performance measures for auto-scaling CPU-intensive containerized applications," *Cluster Computing*, vol. 22, no. 3, pp. 995–1006. Available online: http://link.springer.com/10.1007/s10586-018-02890-1

[39] D. Balla, C. Simon, and M. Maliosz, "Adaptive scaling of Kubernetes pods," in *NOMS 2020 - 2020 IEEE/IFIP Network Operations and Management Symposium*. IEEE, pp. 1–5. Available online: https://ieeexplore.ieee.org/document/9110428/

[40] S. Horovitz and Y. Arian, "Efficient Cloud Auto-Scaling with SLA Objective Using Q-Learning," in *2018 IEEE 6th International Conference on Future Internet of Things and Cloud (FiCloud)*. IEEE, pp. 85–92. Available online: https://ieeexplore.ieee.org/document/8457997/

[41] F. Rossi, V. Cardellini, and F. L. Presti, "Hierarchical Scaling of Microservices in Kubernetes," in *2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*. IEEE, pp. 28–37. Available online: https://ieeexplore.ieee.org/document/9196461/

[42] C.-C. Chang, S.-R. Yang, E.-H. Yeh, P. Lin, and J.-Y. Jeng, "A Kubernetes-Based Monitoring Platform for Dynamic Cloud Resource Provisioning," in *GLOBECOM 2017 - 2017 IEEE Global Communications Conference*. IEEE, pp. 1–6. Available online: http://ieeexplore.ieee.org/document/8254046/

[43] L. Toka, G. Dobreff, B. Fodor, and B. Sonkoly, "Machine Learning-based Scaling Management for Kubernetes Edge Clusters," *IEEE Transactions on Network and Service Management*, vol. 18, pp. 958–972.

[44] C. Delimitrou and C. Kozyrakis, "Quasar: Resource-Efficient and QoS-Aware Cluster Management," p. 17.

[45] A. Bauer, V. Lesch, L. Versluis, A. Ilyushkin, N. Herbst, and S. Kounev, "Chamulteon: Coordinated Auto-Scaling of Micro-Services," in *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, pp. 2015–2025. Available online: https://ieeexplore.ieee.org/document/8885153/

[46] N. Roy, A. Dubey, and A. Gokhale, "Efficient Autoscaling in the Cloud Using Predictive Models for Workload Forecasting," in *2011 IEEE 4th International Conference on Cloud Computing*. IEEE, pp. 500–507. Available online: http://ieeexplore.ieee.org/document/6008748/

[47] G. M. Hamidy, "Differential Fuzzing the WebAssembly." Available online: https://aaltodoc.aalto.fi/handle/123456789/46101

[48] Prometheus Authors. Prometheus Documentation. Available online: https://prometheus.io/docs/introduction/overview/ (Accessed 2021-04-13).

[49] N. Nguyen and T. Kim, "Toward Highly Scalable Load Balancing in Kubernetes Clusters," *IEEE Communications Magazine*, vol. 58, no. 7, pp. 78–83. Available online: https://ieeexplore.ieee.org/document/9161999/

[50] V. Mazalov and A. Gurtov, "Queueing System with On-Demand Number of Servers," *Mathematica Applicanda*, vol. 40, no. 2, pp. 1–12. Available online: http://wydawnictwa.ptm.org.pl/index.php/matematyka-stosowana/article/view/358

[51] T. Hoefler and R. Belli, "Scientific benchmarking of parallel computing systems: Twelve ways to tell the masses when reporting performance results," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, pp. 1–12. Available online: https://dl.acm.org/doi/10.1145/2807591.2807644

[52] C. Xu, K. Rajamani, and W. Felter, "NBWGuard: Realizing Network QoS for Kubernetes," in *Proceedings of the 19th International Middleware Conference Industry*. ACM, pp. 32–38. Available online: https://dl.acm.org/doi/10.1145/3284028.3284033

[53] S. K. Mohanty, G. Premsankar, and M. di Francesco, "An Evaluation of Open Source Serverless Computing Frameworks," in *2018 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE, pp. 115–120. Available online: https://ieeexplore.ieee.org/document/8591002/

[54] S. Eismann, J. Scheuner, E. van Eyk, M. Schwinger, J. Grohmann, N. Herbst, C. L. Abad, and A. Iosup, "A Review of Serverless Use Cases and their Characteristics," p. 45. Available online: https://research.spec.org/fileadmin/user_upload/documents/rg_cloud/endorsed_publications/SPEC_RG_2020_Serverless_Usecases.pdf

A Appendices

A.1 Prometheus Setup

Listing 9 – Installation of Prometheus Helm Chart with kube-state-metrics Exporter

```
helm repo add kube-state-metrics \
    https://kubernetes.github.io/kube-state-metrics
helm repo add prometheus-community \
    https://prometheus-community.github.io/helm-charts
helm repo update
helm install -n monitoring prometheus -f prometheus.values.yaml \
    --version 13.6.0 prometheus-community/prometheus
```

Listing 10 – Configuration of Prometheus Helm Chart (`prometheus.values.yaml`)

```
alertmanager:
  enabled: false

kubeStateMetrics:
  enabled: true # deploys kube-state-metrics exporter

kube-state-metrics:
  image:
    repository: k8s.gcr.io/kube-state-metrics/kube-state-metrics
    tag: v1.9.8
  # enable only specific metric collectors:
  collectors:
    certificatesigningrequests: false
    configmaps: true
    cronjobs: true
    daemonsets: true
    deployments: true
    endpoints: true
    horizontalpodautoscalers: true
    ingresses: true
    jobs: true
    limitranges: true
    mutatingwebhookconfigurations: true
    namespaces: true
    networkpolicies: true
    nodes: true
    persistentvolumeclaims: false
    persistentvolumes: false
    poddisruptionbudgets: true
    pods: true
    replicasets: true
    replicationcontrollers: true
    resourcequotas: true
    secrets: true
    services: true
    statefulsets: true
    storageclasses: false
    validatingwebhookconfigurations: true
```

```
    verticalpodautoscalers: true
    volumeattachments: false

nodeExporter:
  enabled: false

pushgateway:
  enabled: false

configmapReload:
  prometheus:
    enabled: false

server:
  enabled: true
    repository: quay.io/prometheus/prometheus
    tag: v2.24.0
  global:
    scrape_interval: 15s
  retention: 15d
```

A.2 Grafana Setup

Listing 11 – Installation of Grafana Helm Chart

```
helm repo add grafana https://grafana.github.io/helm-charts
helm repo update
helm install -n monitoring grafana -f grafana.values.yaml \
    --version 6.6.4 grafana/grafana
```

Listing 12 – Configuration of Grafana Helm Chart (`grafana.values.yaml`)

```
image:
  repository: grafana/grafana
  tag: 7.4.5

persistence:
  enabled: true

grafana.ini:
  server:
    domain: localhost
    root_url: "%(protocol)s://%(domain)s/grafana"
    serve_from_sub_path: true

ingress:
  enabled: true
  hosts:
    - "localhost"
  path: "/grafana"

datasources:
 datasources.yaml:
   apiVersion: 1
   datasources:
```

```
    - name: Prometheus
      type: prometheus
      access: proxy
      url: http://prometheus-server:80/prometheus
      default: true
```

A.3 Prometheus Service Discovery with Kubernetes

Listing 13 – Kubernetes Service Definition with Prometheus Annotations

```
apiVersion: v1
kind: Service
metadata:
  name: api-server # name of this service definition
  annotations: # tells prometheus to scrape this service
    prometheus.io/scrape: "true"
    prometheus.io/port: "2112"
spec:
  ports:
    - name: "http"
      port: 8080
      protocol: TCP
      targetPort: http
    - name: "pm-exporter"
      port: 2112
      protocol: TCP
      targetPort: pm-exporter
  selector: # binds service to matching pods:
    service: api-server
```

A.4 Prometheus Adapter Setup

Listing 14 – Installation of Prometheus Adapter Helm Chart

```
helm repo add prometheus-community \
     https://prometheus-community.github.io/helm-charts
helm repo update
helm install -n monitoring prometheus-adapter --version 2.12.1 \
     -f prometheus-adapter.values.yaml \
     prometheus-community/prometheus-adapter
```

Listing 15 – Configuration Prometheus Adapter Helm Chart (`prometheus-adapter.values.yaml`)

```
image:
  repository: directxman12/k8s-prometheus-adapter-amd64
  tag: v0.8.3

prometheus:
  url: http://prometheus-server.monitoring
  path: /prometheus
  port: 80

rules:
```

```
  default: true
  external:
    - seriesQuery: '{__name__=~"^esm_execution_tasks_total$"}'
      resources:
        overrides:
          kubernetes_namespace: {resource: "namespace"}
      # metric series from custom ESM exporter
      metricsQuery: esm_tasks_total{status="queued"}
      name:
        matches: ""
        as: "esm_execution_tasks_queued_total"
    - seriesQuery: '{__name__=~"^rabbitmq_queue_messages$"}'
      resources:
        overrides:
          kubernetes_namespace: {resource: "namespace"}
      # metric series from RabbitMQ exporter
      metricsQuery:'rabbitmq_queue_messages{queue="restrictedQueue"}'
      name:
        as: "esm_executor_queue_messages"
    - seriesQuery: '{__name__=~"^esm_configuration_runs_total$"}'
      resources:
        overrides:
          kubernetes_namespace: {resource: "namespace"}
      # metric series from custom ESM exporter:
      metricsQuery:'esm_configuration_runs_total{status="processing"}'
      name:
        as: "esm_configuration_runs_processing"
```

A.5 HPA Scaling Policy

Scaling policy `sp-v1` had the `averageValue` set to 1 and `sp-v2` to 2, respectively.

Listing 16 – HPA scaling policy `sp-v1` for production-like environment

```
apiVersion: autoscaling/v2beta2
kind: HorizontalPodAutoscaler
metadata:
  name: executor
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: executor
  minReplicas: 1
  maxReplicas: 50
  metrics:
  - type: External
    external:
      metric:
        name: esm_executor_queue_messages
      target:
        type: AverageValue
        averageValue: 1
  - type: External
    external:
```

```
      metric:
        name: esm_configuration_runs_processing
      target:
        type: AverageValue
        averageValue: 1
  behavior:
    scaleDown:
      policies:
        - type: Percent
          value: 50
          periodSeconds: 60
      stabilizationWindowSeconds: 60
```

A.6 HPA Log Messages

Listing 17 – HPA log messages (abbreviated)

```
$ kubectl describe hpa/executor
Name: executor
Deployment pods: 1 current / 1 desired
Metrics:              ( current / target )
  "esm_tasks_queued_total":  0 / 1
Messages:
  Recommended size matches current size
  The HPA was able to successfully calculate a replica count from
  external metric esm_tasks_queued_total
  The desired replica count is less than the minimum replica count
  [...]
  New size: 5; external metric esm_tasks_queued_total above target
  New size: 10; external metric esm_tasks_queued_total above target
  New size: 5; All metrics below target
```

A.7 VPA Setup

Listing 18 – Installation of VPA Helm Chart

```
helm repo add fairwinds-stable https://charts.fairwinds.com/stable
helm repo update
helm install vpa fairwinds-stable/vpa -f vpa.values.yaml \
     --version 0.3.2 --namespace vpa --create-namespace
```

Listing 19 – Configuration of VPA Helm Chart (`vpa.values.yaml`)

```
recommender:
  enabled: true
  image:
    tag: "0.9.2"
  extraArgs:
    v: '4' # verbosity level
updater:
  enabled: false
admissionController:
  enabled: false
```

A.8 KEDA Setup

Listing 20 – Installation of KEDA Helm Chart

```
helm repo add kedacore https://kedacore.github.io/charts
helm repo update
helm install keda -f keda.values.yaml kedacore/keda \
    --version v2.2.2 --namespace keda --create-namespace
```

Listing 21 – Configuration of KEDA Helm Chart (`keda.values.yaml`)

```
image:
  keda:
    repository: ghcr.io/kedacore/keda
    tag: 2.2.0
  metricsApiServer:
    repository: ghcr.io/kedacore/keda-metrics-apiserver
    tag: 2.2.0
```

Listing 22 – Installation of KEDA ScaledObject

```
$ kubectl get hpa
NAME                REFERENCE          TARGETS     MINPODS   MAXPODS
keda-hpa-exec-so    deploy/executor    0/1 (avg)   1         50
$ kubectl get scaledobject
NAME       SCALETARGETNAME   MAX   TRIGGERS   READY   ACTIVE   AGE
exec-so    executor          50    rabbitmq   True    False    1m
$ kubectl get -n esm deployment -l service=executor
NAME       READY   UP-TO-DATE   AVAILABLE   AGE
executor   0/0     0            0           1h
```

A.9 Modular Kubernetes Autoscaler CRD

Listing 23 – Example CRD of modular Kubernetes autoscaler

```
apiVersion: wasmpa.io/v1alpha1
kind: ScaledObject
metadata:
  name: server-so
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: server
  metrics:
  - type: Resource
    metricName: cpu
    target: 0.6 # 60% CPU load
  - type: External
    metricName: http_request_duration_seconds
    target: 0.1 # 100ms response time
  algorithm:
    name: wasmpa-arima # use ARIMA algorithm for estimations
    params: [] # list of additional parameters
```