TU WIEN Informatics

Enabling k8ssandra for Diagonal Elasticity Using the Polaris SLO Framework

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

Software & Information Engineering

eingereicht von

Nico Kratky
Matrikelnummer 11909858

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Assistant Prof. Dipl.-Ing. Dr.techn. Stefan Nastic, BSc
Mitwirkung: Projektass. Dipl.-Ing. Thomas Werner Pusztai

Wien, 1. Jänner 2024	________________ Nico Kratky	________________ Stefan Nastic

TU WIEN Informatics

Enabling k8ssandra for Diagonal Elasticity Using the Polaris SLO Framework

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Software & Information Engineering

by

Nico Kratky
Registration Number 11909858

to the Faculty of Informatics

at the TU Wien

Advisor: Assistant Prof. Dipl.-Ing. Dr.techn. Stefan Nastic, BSc
Assistance: Projektass. Dipl.-Ing. Thomas Werner Pusztai

Vienna, January 1, 2024

Nico Kratky	Stefan Nastic

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Nico Kratky

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 1. Jänner 2024

Nico Kratky

Acknowledgements

First of all, I would like to thank my supervisors Thomas Pusztai and Stefan Nastic for advising my work. Thank you for guiding me through this process.

Moreover, I would like to thank my parents and entire family for always supporting me, no matter what.

Kurzfassung

Cloud Computing hat in den letzten Jahren immens an Popularität gewonnen. Eigenschaften wie Elastizität und Pay-as-you-go-Preismodelle haben die Kunden dazu veranlasst, ihre Workload-Bereitstellungsmodelle zu überdenken. Um die Leistungserwartungen zu definieren, verwenden Cloud-Service-Anbieter Service Level Objectives (SLOs). Da die meisten dieser SLOs Low-Level-Metriken verwenden und eng an die elastischen Prozesse gekoppelt sind, wurde das Polaris SLO Framework entwickelt. Dieses Framework ermöglicht abstrakte, High-Level SLOs, die lose an Elastizitätsstrategien gekoppelt sind. In dieser Arbeit wird eine Implementierung einer diagonalen Elastizitätsstrategie vorgestellt, die für die Verwendung mit k8ssandra gedacht ist. Diagonale Elastizität ist definiert als eine Kombination aus vertikaler und horizontaler Elastizität. Es werden die verschiedenen Komponenten vorgestellt, die zur Erreichung dieses Ziels notwendig sind. Schließlich wird das Ergebnis evaluiert und mit der alleinigen Verwendung von vertikaler oder horizontaler Elastizität verglichen.

Abstract

Cloud computing has risen immensely in popularity over the recent years. Properties such as elasticity and pay-as-you-go pricing models have motivated customers to reconsider their workload deployment models. To define performance expectations, cloud service providers use Service Level Objectives (SLOs). As most of these SLOs use low-level metrics and are tightly coupled to the elastic processes the Polaris SLO framework was developed. This frameworks allows for abstract, high-level SLOs that are loosely coupled to elasticity strategies. This thesis presents an implementation of a diagonal elasticity strategy meant for the use with k8ssandra. Diagonal elasticity is defined as a combination of vertical and horizontal elasticity. The different components that are necessary to achieve this are introduced. Finally the result is evaluated and compared to using vertical or horizontal elasticity alone.

Contents

CHAPTER 1

Introduction

The cloud computing paradigm emerged in the recent decade and provides "ubiquitous, convenient, on-demand network access to a shared pool of configurable resources that can be rapidly provisioned and released with minimal management effort or service provider interaction" [1]. These properties together with a pay-per-use principle motivated many customers to adopt this technology. Cloud computing can be differentiated in three basic service models:

1. **Infrastructure as a Service (IaaS)**. This model enables customers to provision processing and storage infrastructure to run arbitrary software. While the consumer has control over both application and operating system, they are not responsible for controling and maintaining the underlying infrastructure. A well known example is Amazon Elastic Cloud Compute (EC2)[1].
2. **Platform as a Service (PaaS)**. The customer is able to deploy applications to provided hosting infrastructure. Control is given only to the deployed application and possibly single configuration settings. The underlying infrastructure is solely controlled by the provider. Notable products include Google App Engine[2].
3. **Software as a Service (SaaS)**. This model allows users to use a provided application that runs on cloud infrastructure. Users do not control the application nor the underlying infrastructure. A suitable example would include Astra DB by DataStax[3].

All three of these service models have one thing in common: they profit from elasticity. Be it the infrastructure that provisions more memory to accomodate more load or be

[1] https://aws.amazon.com/ec2/

[2] https://cloud.google.com/appengine

[3] https://www.datastax.com/products/datastax-astra

it the application that adapts its configuration. The fact that elasticity is a property that benefits a variety of workloads led to the idea to enable the k8ssandra database for different elasticity strategies.

1.1 Problem Statement

The main goal of this thesis is to implement an elasticity strategy that combines vertical scaling with horizontal scaling. This elasticity strategy should be implemented using the Polaris SLO framework. This to be implemented strategy will be from now on called "diagonal elasticity strategy".

As of now, common automatic scaling mechanisms include horizontal and vertical scaling. Kubernetes, for example, has solutions to both. The Horizontal Pod Autoscaler[4] updates the amount of deployed pods to match current demand. Likewise, the Vertical Pod Autoscaler[5] tries to set resource requests and limits based on current usage. Both of these are however limited to their single dimension. For some applications it can be definitely advantageous to be scaled both horizontally and vertically.

1.2 Structure of the Thesis

- Chapter 2 introduces concepts and terminology used throughout this thesis. It discusses the concepts of elasticity in cloud computing and introduces the framework that is used to implement elasticity strategies.
- Chapter 3 presents the implementation details of the project. It first shows the used metrics, then introduces the different service level objectives and finally discusses the elasticity strategies.
- Chapter 4 evaluates the implemented elasticity strategies. This is done by running stress tests in different scenarios.
- Chapter 5 concludes this thesis. It discusses limitations and provides an outlook into possible future work.

[4] https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/

[5] https://github.com/kubernetes/autoscaler/tree/master/vertical-pod-autoscaler

CHAPTER 2

Background

This chapter introduces some terminology and concepts that are used throughout this thesis. First the cloud computing concepts are defined and then the used framework is introduced.

2.1 Elasticity in Cloud Computing

Elasticity is one of the core concepts that solves a big problem of cloud computing: providing limited resources for potentially unlimited use. The solution is to scale workloads up and down as needed, to claim resources when bigger load is experienced and release resources when they are not needed, therefore making them available to other workloads.

The term elasticity in computing is conceptually similiar to the term in physics. Wikipedia, for example, defines elasticity as follows: “In physics and materials science, elasticity is the ability of a body to resist a distorting influence and to return to its original size and shape when that influence or force is removed. Solid objects will deform when adequate loads are applied to them; if the material is elastic, the object will return to its initial shape and size after removal.”[1]

The formula - which takes a more mathematical approach - of elasticity can be defined as

$$e(Y,X) = \frac{\mathrm{d}Y}{\mathrm{d}X}\frac{X}{Y},$$

where $e(Y,X)$ is the elasticity of Y with respect to X [2].

To illustrate this, imagine an application that serves some content to its customers. These customers typically interact with the application during the day. This means that the

[1] https://en.wikipedia.org/wiki/Elasticity_(physics)

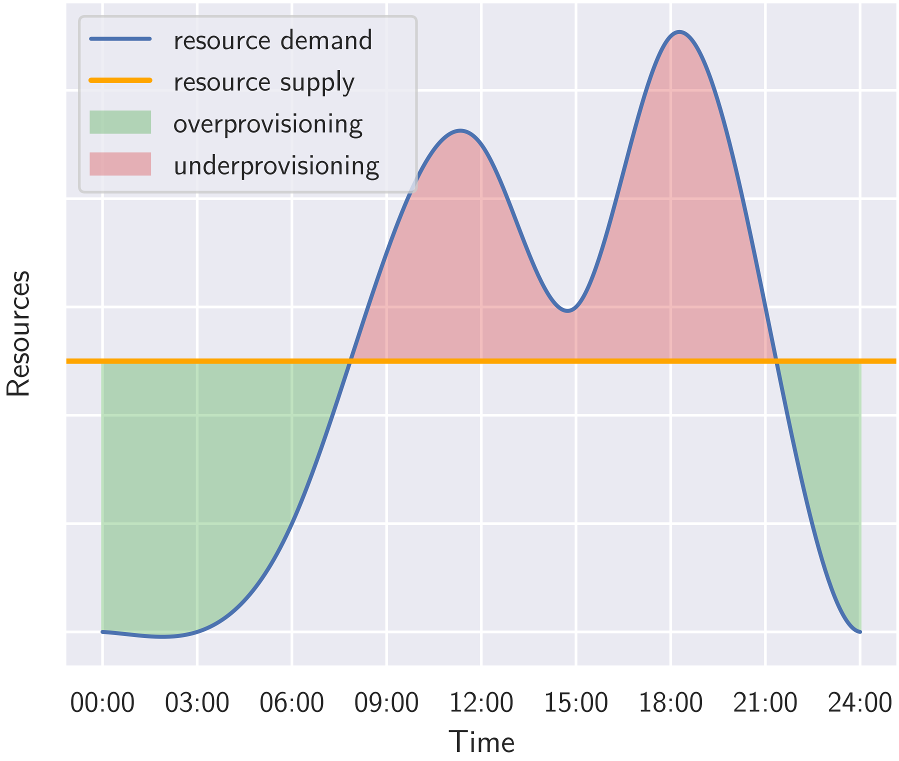

Figure 2.1: Resource demand and supply for a website during a typical day with no elastic processes.

application experiences significantly less load during the night. Once people wake up in the morning the load rises until it peaks in the afternoon. Then the load falls again when people go to sleep in the evening. Using this example it can be seen in figure 2.1 that during the night the resources of the application are overprovisioned and during the day the resoures are underprovisioned.

If the concept of elasticity is applied to this example, resources can be released during the night (so called *scale-in*) and more resources can be claimed as they are needed during the day (so called *scale-out*). This is illustrated in figure 2.2.

Elasticity has multiple properties which are interdependent: resource elasticity, cost elasticity and quality elasticity [2]. These properties are discussed in the following sections.

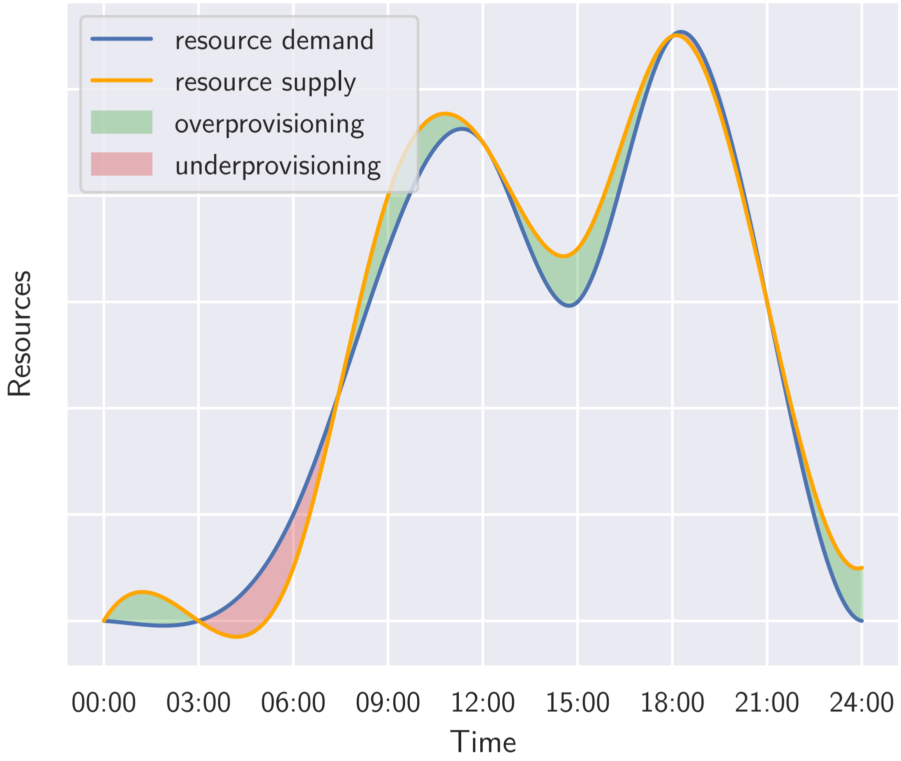

Figure 2.2: Resource demand and supply for a website during a typical day with elastic processes.

2.1.1 Resource Elasticity

The resource dimension of elasticity is mistakenly often used synonymously with elasticity. Meanwhile, resource elasticity is defined as the degree to which a system is able to adapt to workload changes by claiming and releasing resources autonomously, such that the resource supply matches the current demand as closely as possible [3]. Another way to think of this is “on the fly” adaptions to load variations [4].

What makes this definition easily mistaken, is that it solely considers the aquired resources and not the consequently incurred costs or changing quality.

2.1.2 Cost Elasticity

Cost elasticity uses cost as its main factor for elasticity decisions. One of the most popular models that build upon cost elasticity is *utility computing*, also known as the *pay-as-you-go* pricing model.

Amazon Web Services uses this elasticity dimension in their EC2 Spot Instances[2]. AWS provides its unused compute capacity at a large discount to its customers. But because these capacities are volatile, the prices are not fixed but are provided through a bidding process. The potential customer tells AWS their maximum price they are willing to pay. The customer can then run their instances as long as their bidding price is smaller than AWS's Spot Instance price.

2.1.3 Quality Elasticity

Similiar to the already discussed dimensions, quality elasticity is defined as letting software services adapt their mode of operastion to current operating conditions by providing results of varying output quality [5]. This means that when resource supply is low, the output quality also may be low. Likewise, if resource supply is sufficient, the output quality will also be high.

2.2 Service Level Agreements and Service Level Objectives

In order to deliver services up to a certain standard, agreements between the service provider, typically the cloud provider, and the service consumers are made - so called *Service Level Agreements (SLA)* [6]. Contained inside these SLAs are *Service Level Objectives (SLO)*, which are a "commitment to maintain a particular state of the service in a given period" [7].

SLOs are measurable values, e.g. an applications CPU usage or memory consumption, that have a specified operating target. In the case that this value is violated the supporting infrastructure of the application has to be either increased or decreased. This process of increasing or decreasing resources is called elasticity, which was further discussed in section 2.1.

2.3 Polaris SLO Framework

The Polaris SLO Framework[3] is a framework that provides a way to bring high-level SLOs to the cloud. It tries to solve the limitation that modern cloud cloud providers only offer rudimentary support for high-level SLOs and customers often need to manually map them to low-level metrics such as CPU usage or memory consumption [8].

The authors of this framework introduce the concept of *elasticity strategies*. An elasticity strategy is defined as any sequency of actions that adjust the amount of resources provisioned for a workload, their type or the workload configuration. The workload configuration adjustment is especially noteworthy, because workloads handled by Polaris can be affected in all three elasticity dimensions.

[2] https://aws.amazon.com/ec2/spot/

[3] https://polaris-slo-cloud.github.io/polaris-slo-framework/

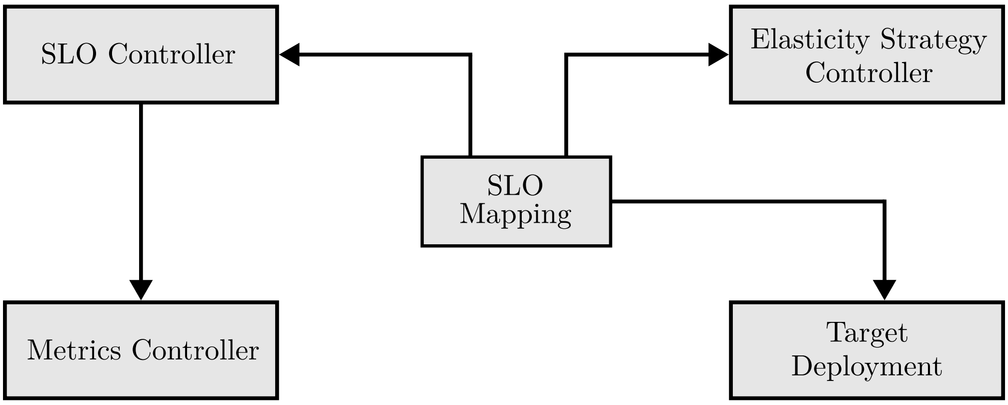

Figure 2.3: Architecture of the Polaris SLO framework. Metrics controllers, elasticity strategy controllers and targets are decoupled and mapped using a SLO mapping.

Another unique feature of Polaris is its object model, which allows for orchestrator independence. This is achieved by encapsulating all data that is transmitted to the orchestrator into a `ApiObject` type.

Decoupling SLOs from elasticity strategies is also a feature that Polaris provides. Tight coupling is a charactaristic that is even observed in industry standard scaling mechanisms such as Kubernetes' Horizontal Pod Autoscaler[4]. This autoscaler provides a CPU usage SLO which can only trigger horizontal elasticity, thus adding or removing CPU resources. To achieve this decoupling, Polaris utilizes an architecture that is depicted in figure 2.3. This allows the controllers to focus on a single task, for example calculating SLO compliance. These individual components are then mapped using a SLO mapping type.

2.4 k8ssandra

Cassandra is a popular wide-column store NoSQL database that was initially developed at Facebook and later integradet into the Apache Software Foundation[5]. Its main features include being easily horizontally scalable, being fully distributed and its schema-less data approach.

Being distributed means, that Cassandra is comprised of a set of nodes. Each nodes tasks and responsibilities are identical. Data is partitioned using a partition key and is replicated between nodes. How many times data is replicated is determined by the

[4] https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/

[5] https://cassandra.apache.org/_/cassandra-basics.html

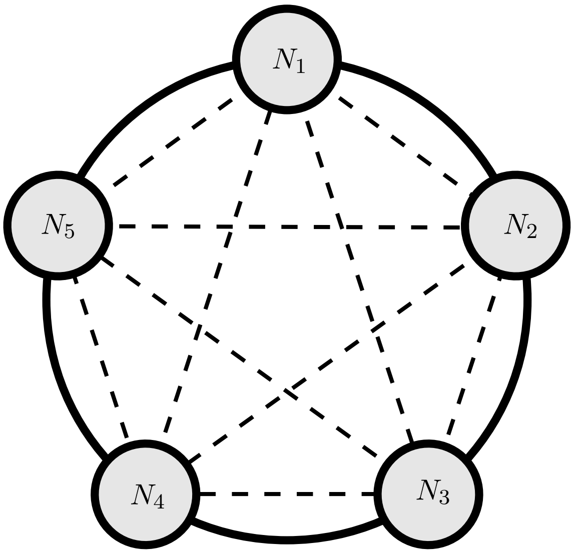

Figure 2.4: Architecture of a 5 node Cassandra Cluster. Dotted lines represent possible communication paths.

replication factor or RF. $RF = 3$ would therefore mean that each piece of data must exist on 3 nodes.

Distributed data also comes with a certain cost. These drawbacks are formulated in the CAP theorem [9]. CAP stands for consistency, availability and partition tolerance and the theorem states that databases which handle data in a distributed way can only provide two of these three guarantees. Cassandra, per default, is an AP database. This agreement, however, is configurable on a per-query basis. This means, that whatever consistency level is configured, it represents the minimum amount of nodes that must acknowledge an operation back to the query coordinator node to consider this operation successful.

Queries can be made to any node. Cassandra does not have a main node that takes queries, instead any node that a client connects to takes over the role of coordinator for this specific query. This coordinator node then is responsible for querying other nodes for data in other partitions. This also implies that Cassandra uses peer-to-peer communication between its nodes. This architecture is also depicted in figure 2.4.

Another powerful feature, which makes this database particular interesting for this thesis, is its capability to scale. If the partition key is chosen wisely and the database is therefore able to distribute data evenly between nodes, then doubling the amount of nodes also doubles the throughput [5] k8ssandra[6] (pronounced: “Kate” + “Sandra”) is an open-source cloud-native distribution of Cassandra that is specifically made to run on Kubernetes. It includes several tools for providing a data API, backup/restore processes and automated database repairs. It also includes Kubernetes custom resource definitions (CRDs) to be able to easily deploy Cassandra databases. It also allows easy integration in existing observability and monitoring stacks such as the `kube-prometheus-stack`[7].

[6] `https://k8ssandra.io`

[7] `https://artifacthub.io/packages/helm/prometheus-community/kube-prometheus-stack`

CHAPTER 3

Implementation

This chapter discusses the implementation of the metrics controller, SLO controllers and the elasticity strategy controllers. These are all components of the Polaris architecture that was described in section 2.3.

3.1 Metrics

In order to continously monitor the k8ssandra cluster, a custom metric is introduced. The Polaris SLO framework supports two kinds of metrics, raw metrics and composed metrics, with the new metric being of the latter type. A composed metric consists of a combination of raw metrics.

The new composed metrics includes three raw metrics: average CPU utilization, average memory utilization and average write utilization. All raw metrics are calculated using the Metrics Collector for Apache Cassandra[1] (MCAC), which is a component included with k8ssandra. The Metrics Collector for Apache Cassandra aggregates operating system level metrics alongside with Cassandra metrics. K8ssandra also provides preconfigured Grafana dashboards[2]. The following metrics were heavily influenced by the metrics that were used in these dashboards.

3.1.1 Average CPU Utilization

The average CPU utilization metric expresses the CPU utilization averaged over the target k8ssandra cluster. This metric is used for vertical elasticity. Listing 3.1 shows the respective PromQL query.

[1] https://docs.k8ssandra.io/components/metrics-collector/

[2] https://docs.k8ssandra.io/tasks/monitor/prometheus-grafana/grafana-dashboards.yaml

```
avg by (cluster) (
  1 - (
    sum by (cluster, dc, rack, instance) (
      rate(
        collectd_cpu_total{
          cluster="polaris-k8ssandra-cluster",
          type="idle"
        }[10m]
      )
    )
    /
    sum by (cluster, dc, rack, instance) (
      rate(
        collectd_cpu_total{cluster="polaris-k8ssandra-cluster"}[10m]
      )
    )
  )
)
```

Listing 3.1: PromQL query used for the average CPU utilisation metric

3.1.2 Average Memory Utilization

Similarly to the average CPU utilization metric, the average memory utilization metric measures the average memory consumption of the target k8ssandra cluster. It is also aimed to be used by vertical elasticity strategies. Listing 3.2 shows a trimmed down version of the PromQL query used by this metric.

3.1.3 Average Write Utilization

This metric measures the average write load that one k8ssandra node experiences. It is used for horizontal scaling, which means adding nodes to the cluster. The metric consists of two separate queries which are shown in listings 3.3 and 3.4. The first query gets the total write load of the cluster and the second query calculates the current amount of active nodes. The before mentioned provided Grafana dashboards offer multiple ways of getting the node count, with the one listed here being among the simplest.

3.2 SLO Controllers

SLO controllers are used to configure and evaluate specific service level objectives. These evaluations are then used to configure the respective elasticity strategies.

As part of this thesis three SLOs and their corresponding controllers were implemented. Two of these are used for the vertical and horizontal elasticity strategies. The third one, called “k8ssandra-efficiency” is a combination of the other ones that is used for the diagonal elasticity strategy.

```
max(
   sum by (pod) (
      container_memory_working_set_bytes{cluster="",namespace="k8ssandra"}
    * on (namespace, pod) group_left (workload, workload_type)
      namespace_workload_pod:kube_pod_owner:relabel{
         namespace="k8ssandra",
         workload="dc1-default-sts",
         workload_type="statefulset"
      }
   )
 /
   sum by (pod) (
      kube_pod_container_resource_limits{
         job="kube-state-metrics",
         namespace="k8ssandra",
         resource="memory"
      }
    * on (namespace, pod) group_left (workload, workload_type)
      namespace_workload_pod:kube_pod_owner:relabel{
         namespace="k8ssandra",
         workload="dc1-default-sts",
         workload_type="statefulset"
      }
   )
)
```

Listing 3.2: PromQL query used for the average memory utilization metric

```
sum by (cluster, request_type) (
 rate(
   mcac_client_request_latency_total{
      cluster="polaris-k8ssandra-cluster",
      request_type="write"
   }[5m]
 )
)
```

Listing 3.3: PromQL query used to get the current write throughput

```
count(
 mcac_compaction_completed_tasks{cluster="polaris-k8ssandra-cluster"} >= 0
)
```

Listing 3.4: PromQL query used to get the amount of nodes in the k8ssandra cluster

3.2.1 Compliance Types

As both the vertical and diagonal elasticity strategies expect input types other than the generic `SloCompliance`, custom types have been created. This is necessary because the elasticity strategy controllers use this data to decide what dimension has to be scaled to what extend. For example, the diagonal elasticity strategy has three parameters that are adjustable: CPU, memory and node count. These values have to be passed from the SLO controller to the elasticity strategy controller.

`K8ssandraVerticalCompliance` is a type that is used, as the name suggests, for expressing vertical compliance. It contains two fields: `currCpuCompliancePercentage` and `currMemorySloCompliancePercentage`. Both these values indicate how much the target k8ssandra clusters current resource claims comply with the SLO.

`K8ssandraCompliance` is a type that includes both of the values from `K8ssandra-VerticalCompliance` and additionally a field `currHorizontalSloCompliance-Percentage`.

All of these values are given as percentages. Both of these types also have a field `tolerance`. By using all of these values it is possible to determine if scaling actions are required at any given time.

3.2.2 API Object

To enable the Polaris SLO framework to interact with the k8ssandra CRD subtype of `ApiObject` was used. `ApiObject` is used for any object that should be added, read, changed or deleted by Polaris using the orchestrator's API.

Because of this use of a subtype the framework is also able to automatically transform fields. Kubernetes for example uses two separate fields for resources, requests and limits. Polaris on the other hand simply uses "resources" as orchestrator details are abstracted. This conversion from requests and limits to resources is handled by Polaris through annotating the respective fields with `PolarisType`.

3.3 Elasticity Strategies

The elasticity strategy controllers perform the actions that are required to scale the workload. All elasticity strategy controllers must implement the interface `Elasticity-StrategyController` which requires the implementation of four methods: `check-IfActionNeeded`, `execute`, `onElasticityStrategyDeleted` and `onDestroy`, with the latter two being optional.

These elasticity strategy controllers are called with the appropriate `SloOutput` during the SLO control loop [10].

As part of this thesis, three elasticity strategies for k8ssandra have been implemented. One each for vertical and horizontal elasticity and one that combines these two into a diagonal elasticity strategy.

3.3.1 Vertical Elasticity Strategy

The vertical elasticity strategy controller is a subtype of `ElasticityStrategy-Controller`. It expects `K8ssandraVerticalSloCompliance`, as described in section 3.2.1, as input. The controller uses the CPU and memory compliance value to scale the current resources accordingly. If the current CPU and memory compliance is the given tolerance range, no scaling is performed by the elasticity strategy controller.

3.3.2 Horizontal Elasticity Strategy

The horizontal elasticity strategy controller is able to use the `SloComplianceElasticityStrategyControllerBase` as its supertype as it expects `SloCompliance` as input. This reduces the amount of boilerplate code and therefore also complies with the “Don’t repeat yourself” (DRY) principle. Again, the elasticity strategy controller performs a scaling action if the compliance is out of range of the set tolerance.

The here implemented version of horizontal scaling *only* performs scale-out. The reason for this is that for scaling-in databases, special considerations have to be made. This is especially true for storage. When, for example, reducing the node count in a Cassandra cluster from 3 to 2, the amount of stored data stays the same, therefore it is possible that the stored data per node increases. This, however, was considered out of scope of this thesis.

3.3.3 Diagonal Elasticity Strategy

The third and last elasticity strategy controller combines the controller described in sections 3.3.1 and 3.3.2.

Again, because this controller expects a different input that `SloCompliance`, `K8-ssandraSloCompliance` it is not possible to use any of the provided controller bases. Therefore a custom controller base that expects this input has been implemented. The diagonal elasticity controller then is a subtype of this newly created controller base.

Due to a normalization process that takes place after the actual scaling, it is possible that even if the elasticity strategy is executed no update to the target is made. This is because there are certain limits that are set statically that have to be adhered to. CPU and memory have physical limits as there is no infite amount of resources that can be claimed by the target. Similarly, a lower boundary is also in place because even if the current utilization is very low, a minimum amount of resources is necessary to guarantee normal operation.

CHAPTER 4

Evaluation

This chapter first introduces the setup that was used for evaluating the different elasticity strategies. Then the results of different tests are presented and discussed.

4.1 Test Setup

In order to test the different elasticity strategies a test environment has to be set up. It was decided to create three virtual machines (VM) that will form a Kubernetes cluster. Because of its ease of use microk8s was chosen as distribution[1]. All three virtual machines were assigned 10 vCPUs and 10GB of memory. One VM acts as the Kubernetes control plane while the other two join the cluster as worker nodes.

Everything that was deployed into the Kubernetes cluster was built using the infrastructure as code (IaC) tool HashiCorp Terraform[2]. This enables rapid changes and reproducibility. Deployed resources include the kube-prometheus-stack[3] for monitoring, the k8ssandra-operator[4] for managing k8ssandra clusters and a definition for a k8ssandra cluster. Additionally, the in section 3.1 mentioned Grafana dashboards are also deployed using Terraform.

Listing 4.1 illustrates a minimal definition of a 3 node k8ssandra cluster. Each node has resource limits of 800 millicpu and 6000MB of memory and 3GiB storage space.

[1] `https://microk8s.io/`

[2] `https://www.terraform.io/`

[3] `https://artifacthub.io/packages/helm/prometheus-community/kube-prometheus-stack`

[4] `https://docs.k8ssandra.io/components/k8ssandra-operator/`

```
apiVersion: k8ssandra.io/v1alpha1
kind: K8ssandraCluster
metadata:
  name: polaris-test-cluster
  namespace: k8ssandra
spec:
  cassandra:
    resources:
      limits:
        cpu: 800m
        memory: 6000M
    datacenters:
      - metadata:
          name: dc1
        size: 3
        storageConfig:
          cassandraDataVolumeClaimSpec:
            resources:
              requests:
                storage: 3Gi
```

Listing 4.1: Minimal example of a K8ssandraCluster definition.

```
./cassandra-stress write n=1000000 -mode native cql3 \
    user=’USERNAME’ password=’PASSWORD’
```

4.2 Benchmarks

In the following sections, different test scenarios will be discussed. To let k8ssandra experience load, the built-in stress testing tool `cassandra-stress` was used[5].

4.2.1 Stress Testing

To set a baseline, three different k8ssandra cluster setups have been stress tested using `cassandra-stress`. The amount of write requests that the tool will make is set to be 1000000, the exact call is listed in section 4.2.1. These cluster setups merely differ in the cluster size, thus the amount of nodes. All clusters were provisioned with limits of 2 CPUs and 6GB of memory.

The results of these tests are depicted in figures 4.1, 4.2 and 4.3. The write throughput increases with the amount of nodes, but not linearly. This, however, was to be expected as `cassandra-stress` does not partition data in a way that favours linear scalability. The average write throughputs of these different clusters can be seen in table 4.1.

[5] https://cassandra.apache.org/doc/stable/cassandra/tools/cassandra_stress.html

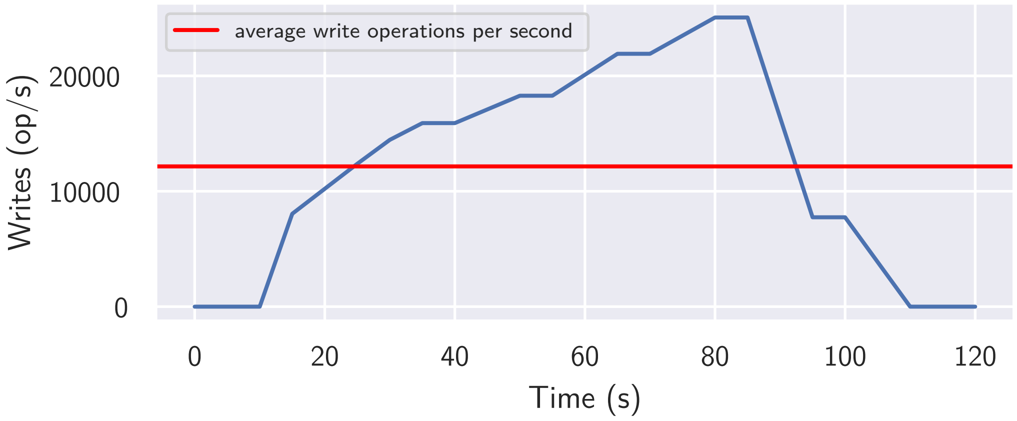

Figure 4.1: Stress test of 1 node with 1000000 writes

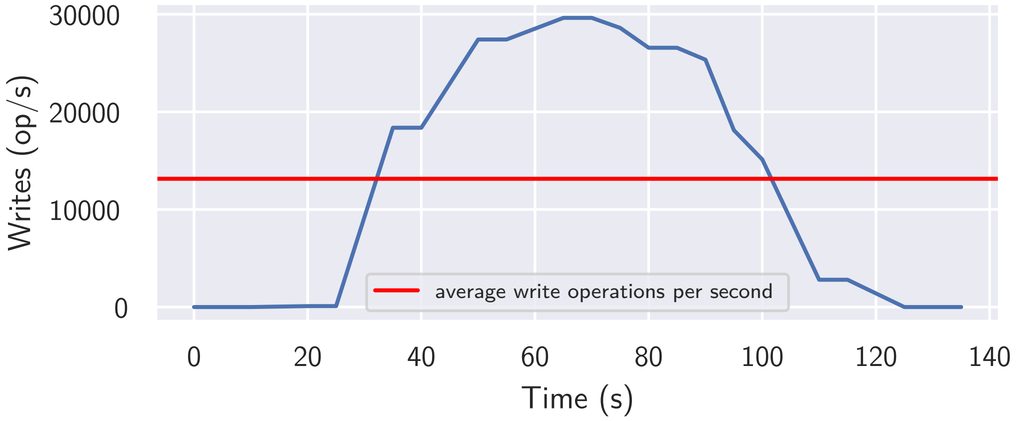

Figure 4.2: Stress test of 2 nodes with 1000000 writes

Cluster size	**operations/s**	**Time to complete**
1	12514	2m38s
2	13142	1m57s
3	14318	1m50s

Table 4.1: Average write throughput for different k8ssandra clusters. With increasing cluster size the throughput also increases

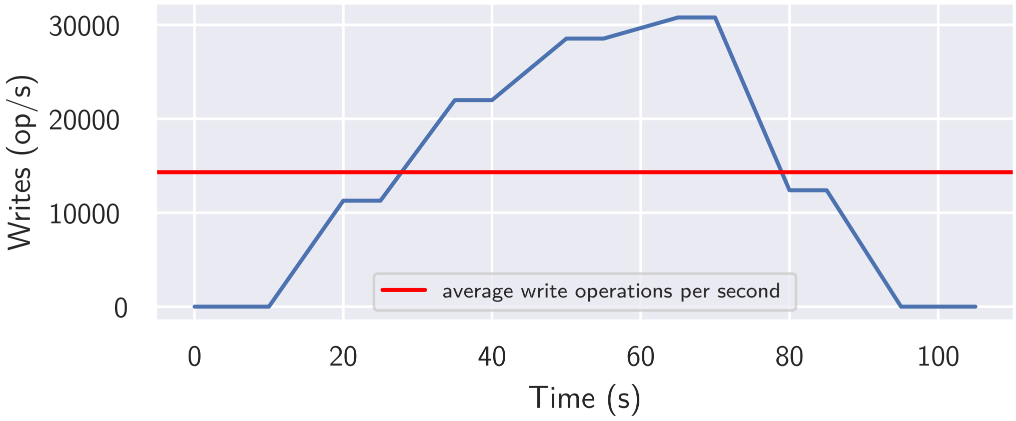

Figure 4.3: Stress test of 3 nodes with 1000000 writes

4.2.2 Vertical Elasticity Strategy

As mentioned in section 3.3.1 the vertical elasticity strategy adjusts the resource claims of k8ssandra according to its CPU and memory utilization.

As it can bee seen in figure 4.4 the elasticity strategy controller successfully changes the CPU and memory limits of the k8ssandra cluster once it is operational. Figure 4.5 shows the CPU and memory utilization that is used for triggering elasticity processes. Because the CPU utilization stays very low even after scaling takes place, it can be assumed that this metric was not a decisive factor. The memory utilization, however, changes notably. Before starting the elasticity strategy controller the actual memory utilization was off by $> 10\%$ from the target memory utilization. This triggers an elasticity event and the resources are adjusted proportionally.

Interestingly, during reconsiliation the exposed metrics of k8ssandra are not very meaningful. During this process utilization values of far more than 100% are exposed by the metrics controller. In order to keep the diagram clean, these nonsense-metrics have been filtered out. The reconsiliation process is marked red in figure 4.5.

This elasticity strategy mirrors real-life scenarios. The advantage lies in being able to scale down when demand and therefore CPU and memory utilization is low, thus potentially reducing cost. This obviously only applies when not using dedicated resources.

4.2.3 Horizontal Elasticity Strategy

The horizontal elasticity strategy controller scales the target k8ssandra cluster horizontally, thus adding nodes as demand increases. Demand is measured as write throughput by the metrics controller as described in section 3.1.3.

As in the example stress tests discussed in section 4.2.1, `cassandra-stress` was used

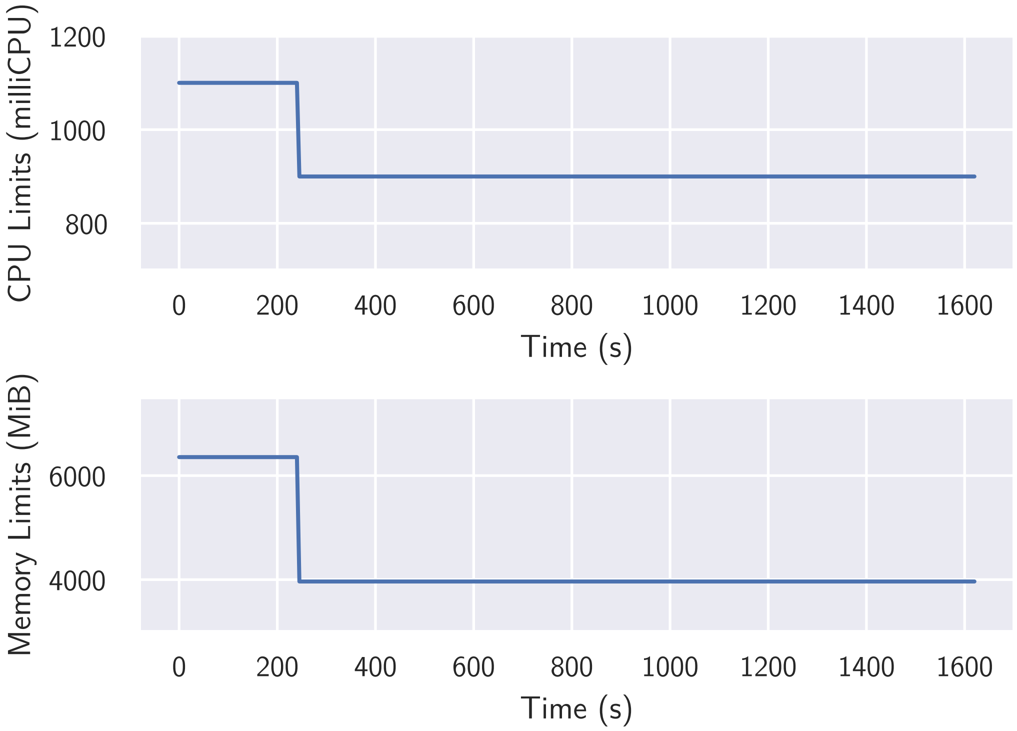

Figure 4.4: Adjustment of CPU and memory limits by the vertical elasticity strategy controller

to generate load on the target k8ssandra cluster. During this load generation process, the horizontal elasticity controller was running. The target write load per node was defined in the SLO mapping as 5000. Depicted in figure 4.6 is the average write load per node metric and the corresponding node count during the testing process. It can be seen that the node count does not increase immediatly when the scaling action takes place. That is because when the k8ssandra CRD is updated by the elasticity strategy controller, first the `k8ssandra-operator` has to recognize the made changes and adjust the configuration accordingly. When the second k8ssandra node is successfully scheduled it still needs time to start and finally register in the cluster. The final action is the Cassandra reconciliation process.

At approximately 290s a sudden drop in the metric can be observed. This is the point when the scaling action becomes effective and the k8ssandra node is ready. Then, after another few moments the metric drops under the set boundary of 5000. Tests of this kind are difficult to run over an extended period of time because of a limitation of `cassandra-stress`. When the load generator is started, it collects all available nodes in the cluster through Cassandra's communication protocol `Gossip`. `Gossip` is the

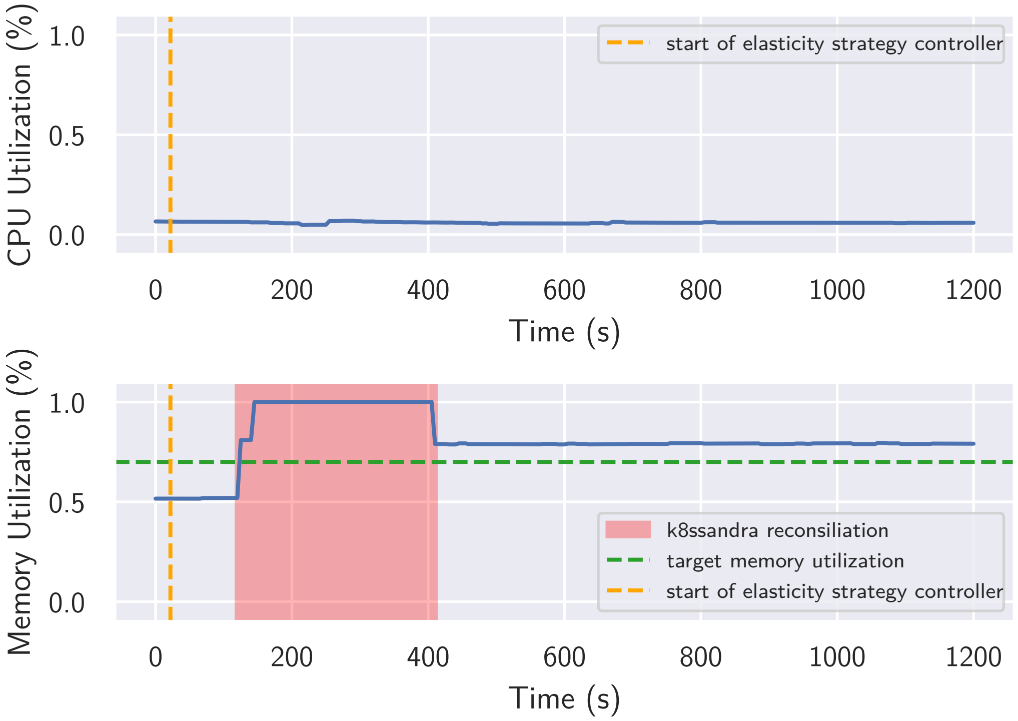

Figure 4.5: Utilization of CPU and memory during an vertical scaling action

protocol that Cassandra uses internally for its nodes to communicate with each other[6]. While `cassandra-stress` is running, new nodes are not recognized and requests are therefore not sent to added nodes. Possible solutions to this will be discussed in chapter 5.

4.2.4 Diagonal Elasticity Strategy

As explained earlier, the diagonal elasticity strategy combines the capabilites of the vertical and horizontal elasticity strategy into one single elasticity strategy.

Figure 4.7 summarizes all metrics into a single illustration. The starting configuration was set to be a single k8ssandra node with resources of 2 CPUs and 6GB of memory. After starting the elasticity strategy controller it can be seen in figure 4.8 that the controller immediatly reduces both CPU and memory resources. The reason for that can be seen in figure 4.7, subfigure `c` and `d`. Right at the start, both CPU and memory utilization was not within the tolerance range of the target utilization. Therefore both CPU and memory limits were reduced. After the inital adjustment, the CPU utilization was still far away from the targeted amount. That is because the CPU resources hit the statically

[6] `https://docs.datastax.com/en/cassandra-oss/3.x/cassandra/architecture/archGossipAbout.html`

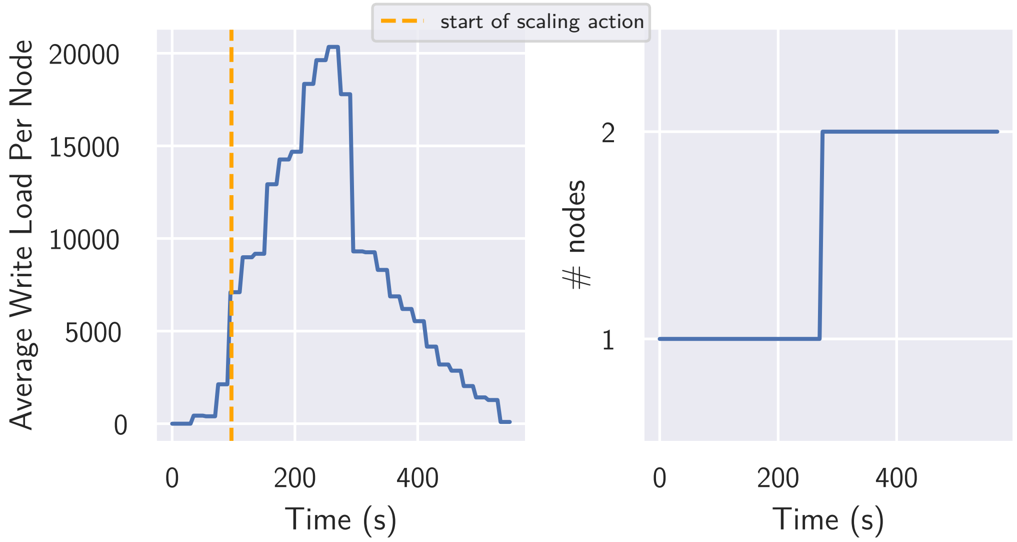

Figure 4.6: Average write load per node and amount of nodes during a horizontal scaling action

set lower bounds. The memory utilization however climbed above the targeted amount, therefore it was reduced again during the second scaling action.

Similarly to section 4.2.2, during Cassandra reconsiliation metrics are not very useful. This is again highlighted in red in figure 4.7.

During the second scaling action it can be seen that vertical and horizontal scaling indeed can happen simultaneously. In subfigure `b` of figure 4.7 the node count increased to 2, whereas in figure 4.8 the memory limits increased. Note, that the `k8ssandra-operator` adjusts those values one at a time. This means that first the second k8ssandra node is started and then both Pods will get its resources updated accordingly.

Horizontal scaling action are taken when the write load per node reaches a certain threshold, 5000 in this example. In figure 4.7, subfigure `a` it can be seen that after the second, third and fourth scaling event, the k8ssandra cluster size is increased, thus an additional node is started. During the fifth and last scaling event no additional node is started, because the statically set maximum amount of nodes is reached. It is also visible, that the total write throughput increases with increasing node count. This can be further illustrated by multiplying the estimated peak write load with the current node count. $18000 * 1 = 18000,\ 12000 * 2 = 24000,\ 9000 * 3 = 27000 \rightarrow 18000 < 24000 < 27000.$

Because of the in section 4.2.3 addressed drawback of `cassandra-stress`, which does not detect changes to the cluster architecture, stress tests were cancelled after new nodes were added, and restarted when Cassandra had finished its reconsiliation process.

The advantage of this elasticity strategy is its ability to scale vertically and horizontally

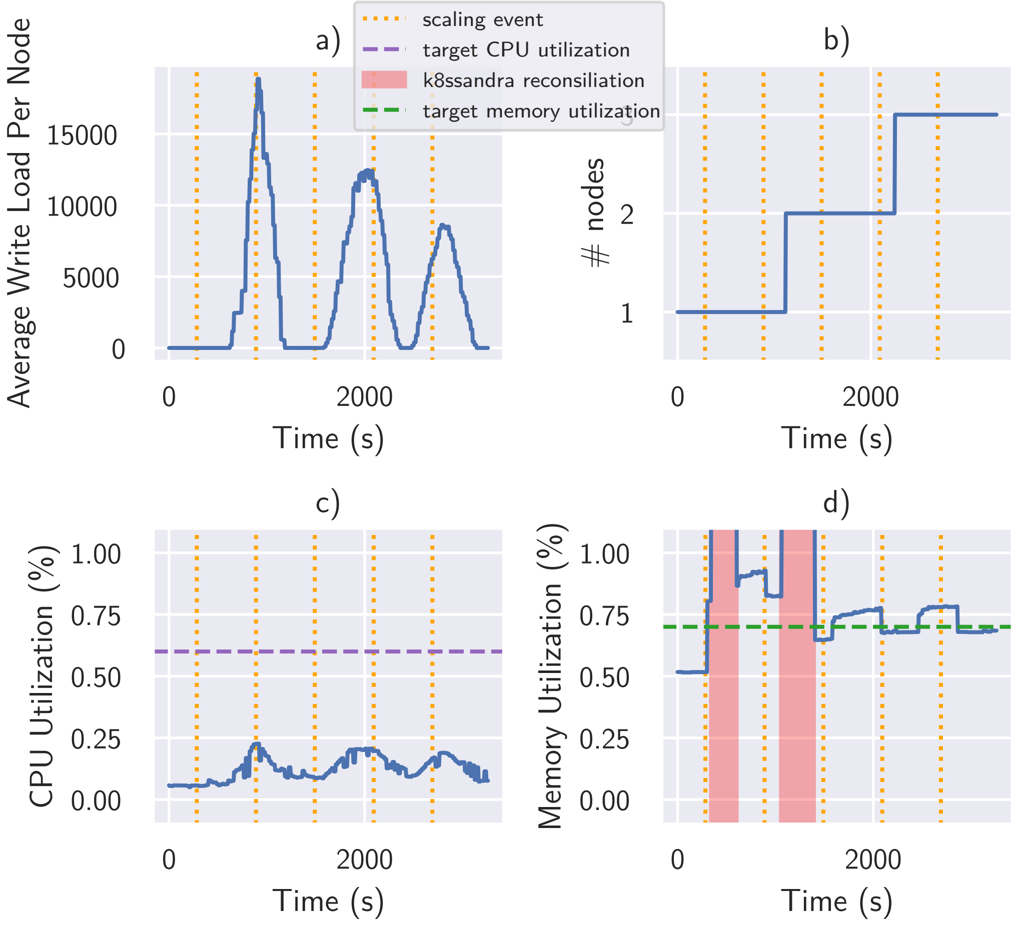

Figure 4.7: Adjustment of CPU and memory resources as well as cluster size during diagonal elasticity

independently. This means that during times of low demand resources can be saved or used by other applications. A lower amount of resources also implies lower costs. During high demand times resources can be claimed again to provide a sufficient service level. If k8ssandra reports a high amount of writes the elasticity strategy can the also decide to scale-out horizontally by adding more nodes. As it was shown in section 4.2.1 this increases the total throughput. As mentioned before, horizontal scale-in is not implemented in this project. This will be further addressed in section 5.1.

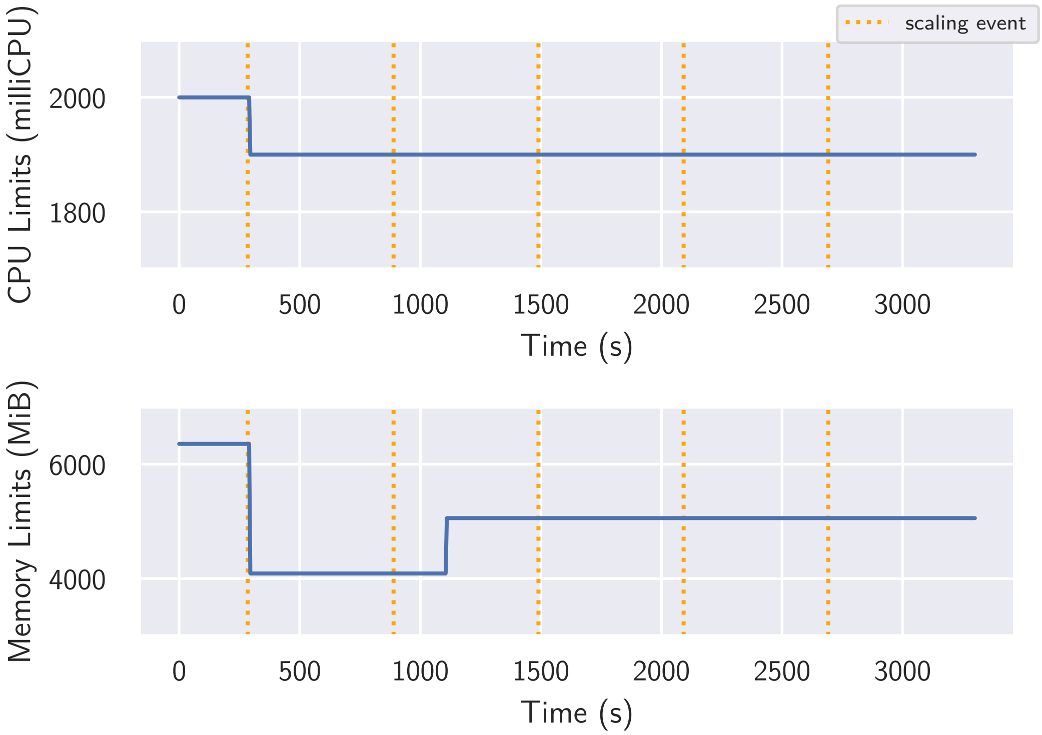

Figure 4.8: CPU and memory limits while the diagonal elasticity strategy controller is running

CHAPTER 5

Conclusion

This chapter concludes the thesis by summarizing the results, discussing the limitations and outlining possible future work.

By enabling k8ssandra to scale vertically, horizontally and both combined, thus scaling diagonally, different tasks can be achieved. Vertical scaling reduces to allocated resources when not in use. This in turn reduces cost when using cloud computing infrastructure through its pay-as-you-go pricing model. On the other hand, freeing resources when not in use allows other applications to claim them, making scheduling applications much easier when working with a limited amount of resources.

Horizontal scaling allows k8ssandra to essentially scale its throughput linearly. This was not only shown by the Apache Software Foundation, the developers of Cassandra, but also by industry leading companies such as Netflix[1].

Combining those two dimensions into a single elasticity strategy using the Polaris SLO framework, the benefits from both dimensions can be combined. Cost reduction through releasing and claiming resources dynamically and scaling throughput by adding nodes when demand is sufficient.

Nevertheless, limiting factors exist. First and foremost, Cassandra is not designed to be a dynamic application. While it is possible to remove and add nodes to a running k8ssandra cluster, substantial load is generated because Cassandra needs to reconcile the cluster. During this time, the newly added nodes are not operational and other already existing nodes experience significant load that impairs operability.

[1] https://netflixtechblog.com/benchmarking-cassandra-scalability-on-aws-over-a-million-writes-per-second-39f45f066c9e

5.1 Future Work

During implementation various issues arose that were deemed out of scope to solve. These imply the following suggestions:

- **Horizontal scale-in**. As described in section 3.3.2, the within this thesis implemented version of horizontal scaling only perform scale-out due to the fact that further considerations related to storage have to be made. Some Kubernetes storage drivers support dynamic volume expansion[2], therefore this poses an opportunity for further development.
- **In-place resource resize**. Earlier this year Kubernetes released a feature that allows resource updates to pods without them needing to restart[3]. This would be beneficial as restarting k8ssandra nodes takes a long time.
- **Improve stress testing**. Using `cassandra-stress` as load generation tool has the advantage of being a native Cassandra tool. The downside of this tool is that it is relativly inflexible. As mention in section 4.2.3 the cluster architecture is only discovered once during startup. Therefore changes to the architecture are not immediately reflected in the stress test.
- **Scale to zero**. To provide even more cost effectiveness during times where there is no demand, a scale-to-zero approach could be taken. k8ssandra supports stopping the cluster as a whole. This could be subject to further research.

[2] https://kubernetes.io/blog/2022/05/05/volume-expansion-ga/

[3] https://kubernetes.io/blog/2023/05/12/in-place-pod-resize-alpha/

Bibliography

[1] P. Mell and T. Grance, "The NIST Definition of Cloud Computing," Tech. Rep. NIST Special Publication (SP) 800-145, National Institute of Standards and Technology, Sept. 2011.

[2] S. Dustdar, Y. Guo, B. Satzger, and H.-L. Truong, "Principles of Elastic Processes," *Internet Computing, IEEE*, vol. 15, pp. 66–71, Nov. 2011.

[3] N. R. Herbst, S. Kounev, and R. Reussner, "Elasticity in Cloud Computing: What It Is, and What It Is Not," in *10th International Conference on Autonomic Computing (ICAC 13)*, pp. 23–27, June 2013.

[4] 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, pp. 430–447, Mar. 2018.

[5] L. Larsson, W. Tärneberg, C. Klein, and E. Elmroth, "Quality-Elasticity: Improved Resource Utilization, Throughput, and Response Times Via Adjusting Output Quality to Current Operating Conditions," in *2019 IEEE International Conference on Autonomic Computing (ICAC)*, pp. 52–62, June 2019.

[6] V. C. Emeakaroha, I. Brandic, M. Maurer, and S. Dustdar, "Low level Metrics to High level SLAs - LoM2HiS framework: Bridging the gap between monitored metrics and SLA parameters in cloud environments," in *2010 International Conference on High Performance Computing & Simulation*, pp. 48–54, June 2010.

[7] A. Keller and H. Ludwig, "The WSLA Framework: Specifying and Monitoring Service Level Agreements for Web Services," *Journal of Network and Systems Management*, vol. 11, pp. 57–81, Mar. 2003.

[8] T. Pusztai, A. Morichetta, V. C. Pujol, S. Dustdar, S. Nastic, X. Ding, D. Vij, and Y. Xiong, "SLO Script: A Novel Language for Implementing Complex Cloud-Native Elasticity-Driven SLOs," in *2021 IEEE International Conference on Web Services (ICWS)*, pp. 21–31, Sept. 2021.

[9] A. Fox and E. Brewer, "Harvest, yield, and scalable tolerant systems," in *Proceedings of the Seventh Workshop on Hot Topics in Operating Systems*, pp. 174–178, Mar. 1999.

[10] T. Pusztai, A. Morichetta, V. C. Pujol, S. Dustdar, S. Nastic, X. Ding, D. Vij, and Y. Xiong, “A Novel Middleware for Efficiently Implementing Complex Cloud-Native SLOs,” in *2021 IEEE 14th International Conference on Cloud Computing (CLOUD)*, pp. 410–420, Sept. 2021.