SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Resource-aware pod (re)scheduling in Kubernetes

B.Sc. Jannik Straube

SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Resource-aware pod (re)scheduling in Kubernetes

Ressourcengewahre Umverteilung von Pods in Kubernetes

Author:	B.Sc. Jannik Straube
Supervisor:	Prof. Dr. Michael Gerndt
Advisor:	M.Sc. Mohak Chadha
Submission Date:	15.06.2023

I confirm that this master's thesis is my own work and I have documented all sources and material used.

J. Straube

Munich, 15.06.2023 B.Sc. Jannik Straube

Acknowledgments

I would like to sincerely appreciate two individuals who have played vital roles in completing my master's thesis. Their support, guidance, and encouragement have been invaluable throughout this journey.

First and foremost, I would like to extend my gratitude to my thesis supervisor, Mohak Chadha. His expertise, dedication, and interest in my research have been instrumental in shaping the direction of this thesis. I am genuinely grateful for his valuable insights, constructive feedback, and thought-provoking questions that have helped refine my work. Additionally, I am thankful for his motivation and encouragement, which have kept me motivated during the challenging moments.

Secondly, I am immensely grateful to my co-founder at vystem.io, Maximilian Mayr, for his invaluable contributions. During the initial stages of our journey, when we were confronted with the challenge of managing high traffic amidst the Covid pandemic, his calm and supportive presence provided much-needed reassurance. I deeply appreciate his unwavering support and understanding, especially during those moments of intense pressure when our RPS and latencies surged and I found myself overwhelmed with the task of taming our Kubernetes cluster.

To Mohak Chadha and Maximilian Mayr, thank you for your time, dedication, and unwavering support.

Finally, I would like to acknowledge the support and understanding of my friends, family, and all those who have provided emotional support throughout this demanding period. Your encouragement has been crucial in helping me overcome the obstacles and challenges that arose during the thesis journey.

Abstract

As the popularity of containerized applications grows, container orchestration platforms like Kubernetes are becoming more widely used. One of the critical challenges in managing these systems is the efficient allocation of resources to pods, which are the smallest deployable units in Kubernetes. Resource utilization issues can result in poor application performance or even application failure.

This thesis proposes Kinema, a resource-aware pod (re)scheduling system for Kubernetes. The goal of this system is to improve the efficiency of resource utilization during idle workload periods. To achieve this goal, Kinema takes a holistic cluster view by considering each pod's resource requirements and current utilization levels, as well as the current node configurations and the stability of the cluster.

By simulating a virtual cluster environment and optimizing the workload placement, also considering other cloud provider offerings, Kinema aims to find the most efficient cluster configuration. An internal rescheduling approach executes the necessary operations to reach this optimal cluster state.

The effectiveness of the proposed system is evaluated through load tests based on two sample applications and three different load patterns on a Kubernetes cluster running on the Google Cloud. Additionally, scalability experiments based on real-world cluster data are performed. When comparing with the state-of-the-art optimize-utilization autoscaling profile, Kinema achieves an average relative cost improvement of 39% in the context of 3 different load patterns based on two sample applications once an idle state is reached. The average CPU utilization shows a relative improvement of 113%.

In conclusion, the resource-aware pod (re)scheduling system, Kinema proposed in this thesis is a promising approach to optimizing cluster efficiency, especially during stable and idle load periods.

Contents

Acknowledgments **iii**

Abstract **iv**

1 Introduction **1**
- 1.1 Contribution . . . 3
- 1.2 Methodology . . . 4
  - 1.2.1 Research questions . . . 4
  - 1.2.2 Evaluation and experiments . . . 5

2 Background **6**
- 2.1 Kubernetes . . . 6
  - 2.1.1 Kubernetes architecture . . . 6
  - 2.1.2 Pod Autoscaling . . . 7
  - 2.1.3 Scheduling approaches in Kubernetes . . . 14
  - 2.1.4 Cluster Autoscaling . . . 14
- 2.2 Linear programming . . . 15
  - 2.2.1 Introduction to linear programming and its use in optimization problems . . . 15
  - 2.2.2 Mathematical foundations . . . 16
  - 2.2.3 Mixed Integer Program (MIP) . . . 16
  - 2.2.4 Challenges and limitations . . . 18

3 Related Work **20**
- 3.1 Minimizing resource wastage by improving initial pod scheduling . . . 20
- 3.2 Improve resource usage by adding real-time metrics . . . 21
- 3.3 Resource usage optimization by moving workloads between nodes . . . 22

4 Metrics-driven scheduling **23**
- 4.1 Load-metric aware scheduling . . . 23
- 4.2 Trimaran target load packaging . . . 24
- 4.3 Experimental comparison . . . 26
- 4.4 Summary . . . 28

5 Kinema - a holistic, metric aware Kubernetes rescheduler 31
5.1 Initial prototype design 31
5.1.1 Metrics deschedule approach 31
5.1.2 Issues with initial prototype design 32
5.2 Revised architecture 33
5.2.1 Architecture overview 33
5.2.2 Rescheduling sequence 35
5.3 Kinema components 37
5.3.1 Cluster Observer 38
5.3.2 Utilization Predictor 39
5.3.3 Placement optimizer 41
5.3.4 Decision processor 45
5.3.5 Rescheduler 45
5.4 Autoscaling 48
5.5 Limitations and potential optimizatizations 49

6 Experiments 50
6.1 Load Test experiment 50
6.1.1 Setup 50
6.1.2 Linear ramp up 53
6.1.3 Spike workload 58
6.1.4 Random workload 61
6.1.5 Conclusion 65
6.2 Kinema scalability 66
6.2.1 Dataset and setup 66
6.2.2 Results 67
6.2.3 Conclusion 68

7 Future work 70

Abbreviations 72

List of Figures 74

List of Tables 76

Bibliography 77

1 Introduction

Many mission-critical microservice applications in various business areas are nowadays deployed via Container-as-a-Service (CaaS) offerings due to the simplicity in scaling and portability [1]. Kubernetes has become one of the most popular open-source platforms to deploy and manage these workloads [2, 3, 4]. Containers are combined in Kubernetes as *pods*, which are scheduled onto underlying nodes. As each node only has a limited capacity in computing resources, a pod comes with computing requirements that are either defined during the deployment or set to a default value. Based on these resource requirements, the default kube-scheduler tries to find the best matching node for the workload. Actual resource usage metrics of the nodes are not considered in the default configuration. As setting resource requirements has to be done by hand, this bears a high probability of causing resource wastage [5, 3].

This resource wastage is visible in Figure 1.1. The data displayed is an export of production metric data from the startup vystem. The company offers an online platform that unifies livestreaming with videoconferencing and is used by various industries, such as the movie industry. Sample customers include Netflix and Amazon Prime, who utilize the tool to stream press conferences of upcoming movies in an event format. The software is fully hosted on Google Kubernetes Engine (GKE).

The first two plots in the Figure display the respective CPU and memory data. The stacked plot shows how many resources are actually used in comparison to the total available resources. For example, in the highlighted red area, an average of only 12% of the available CPU and 24% memory is being used.

One reason for this resource wastage is the task of engineering teams to set these resource requests manually and to ensure that all Service Level Objective (SLO) are met. This responsibility often leads to resource wastage as the pod's requirements are too high to ensure peak loads can be handled [6].

The Figure's line plot displays the accumulated resource requests of all currently running pods. If the line exceeds the total allocatable amount of resources, the cluster is currently overcommitted, as visible on the first day of the CPU plot. In the highlighted area, only 17.5% of the requested CPU millicores are being used, a resource wastage caused by the engineering team. Of the requested GiB memory, 85.5% is used, which is a good result.

A fallacy would be suggesting to drastically reduce the CPU requests of the ap-

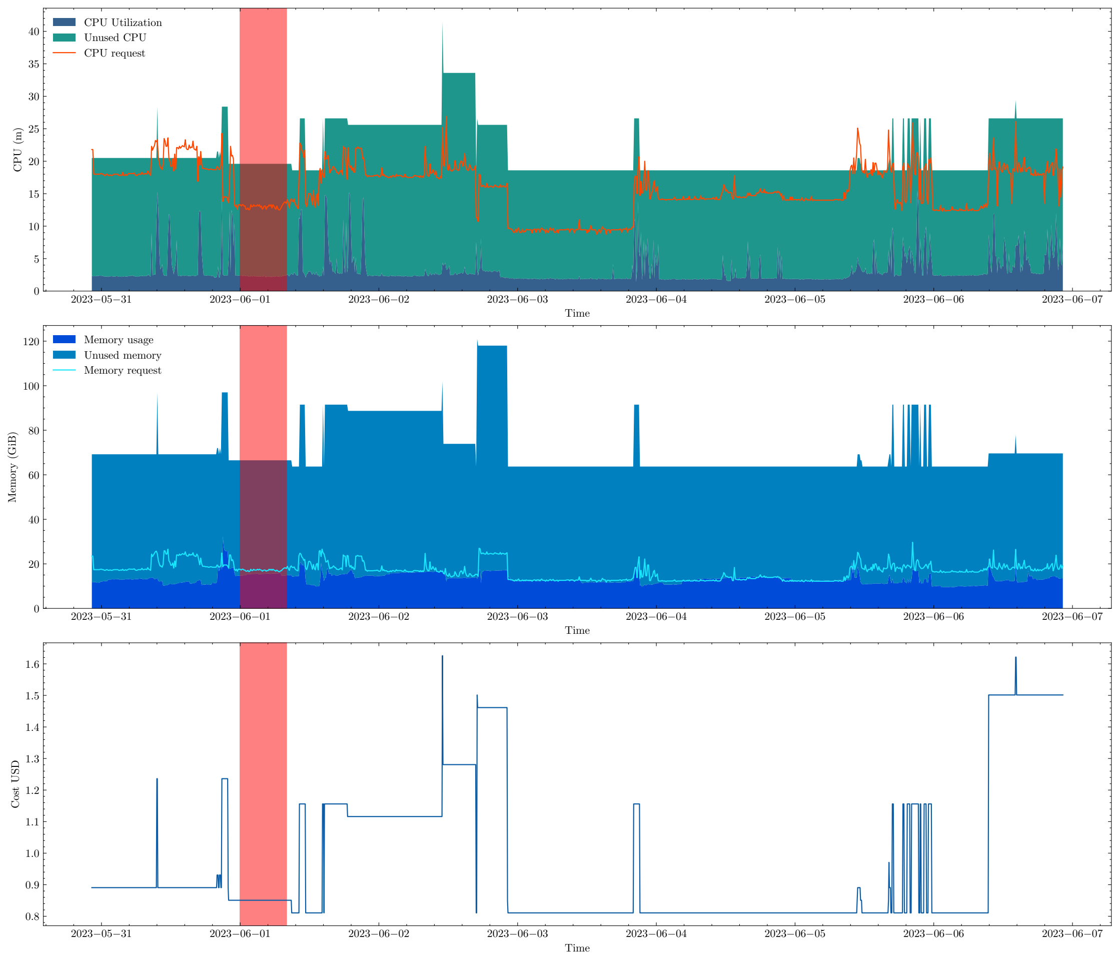

Figure 1.1: Visualization of the utilization of CPU and memory resources of the vystem cluster in one week

plications as traffic in the domain of the software could appear instantly, as visible by the spikes in the CPU plot. Making minor adjustments to the resource requests automatically when the probability of traffic spikes is low would be a possible solution.

Another improvement is selecting the node configurations to match the resource requests. Here, the highlighted area in the plot visualizes a discrepancy between the requested CPU millicores and the available CPU millicores of 34%. This is even higher in terms of memory, with a wastage of 74%. In the case of the data provided, the node configuration is chosen automatically by the Cluster Autoscaler (CA). The CA adds new nodes when a scale-up is required and removes nodes when their respective workloads can also run on other nodes. In the latter scale-down phase, only the currently running nodes are considered instead of considering all available node configurations. As present in the current data, this can lead to suboptimal results.

The highlighted area is an eight-hour time window starting at midnight with a node configuration of four e2-medium machines with two CPU cores and four GB of memory and two e2-standard-8 machines with eight CPU cores and 32 GB of memory. This configuration has a running cost of $0.85 per hour. An optimal node configuration has to offer at least 13.7 milicores of CPU and 18.2 GB of memory. Utilizing e.g. two e2-highcpu-8 machines and one e2-highcpu-4 machine would be able to accommodate all workloads and reduce the running costs per hour to $0.56, resulting in relative savings of 34%.

The savings could be even higher if the resource requests are adjusted by taking steps toward the actual utilization. This can be done by employing e.g. a Vertical Pod Autoscaler (VPA), which can suggest resource requests based on the actual resource usage of a pre-defined time window.

To the present day, a holistic system that puts this optimization into practice by reevaluating the node configuration and resource requests of workloads based on utilization metrics does not exist.

1.1 Contribution

The core contribution of this thesis is the system Kinema, a holistic rescheduling solution to optimize node configurations in Kubernetes clusters. Kinema addresses the challenges associated with efficient resource management in cluster environments, particularly during idle periods. In these periods, the default Kubernetes CA cannot reach the most efficient solution as only the currently available nodes are considered for rescheduling workloads.

Kinema considers factors such as whole cluster state, future predicted resource utilization, cloud vendor pricing, and offering to make intelligent decisions regarding node configuration and pod placement. It is, therefore, not a simple CA, but a holistic rescheduler that optimizes both, node and workload configurations.

When comparing with the state-of-the-art optimize-utilization autoscaling profile, Kinema achieves an average relative cost improvement of 39% in the context of three different load patterns based on two sample applications. The average CPU utilization shows an average relative improvement of 113%.

The practical implementation of Kinema can be seamlessly integrated into existing cloud cluster environments. The system runs within the kubesystem namespaces and gathers data directly from the Kubernetes API server. If an additional Prometheus instance is present, the existing utilization data can be used to employ Kinema's internal utilization predictor to make rescheduling choices. Alternatively, these recommendations can be fetched from the API Server also if a VPA is configured. The system is equipped with a cloud controller that leverages the public APIs of several cloud providers to manage and adjust the underlying node pools. Additionally, Kinema synchronizes an internal catalog of current instance prices to make the correct node configuration decisions.

1.2 Methodology

This section discusses the methodology employed in this thesis. First, the research questions guiding the investigation toward metric-aware rescheduling are presented before explaining the evaluation and experiments conducted to validate the system design.

1.2.1 Research questions

RQ1: Experimental evaluation of the current state-of-the-art in metrics-based scheduling

Research question one evaluates the state-of-the-art metrics-based scheduling using the Trimaran scheduler plugin. The default Kubernetes scheduler and a scheduler with the Trimaran plugin configured and activated are deployed on two different Kubernetes cluster configurations. One utilizes the default CA and another the optimize utilization autoscaling profile, which aims to reduce resource wastage. Multiple load-tests with the same load pattern are deployed in all configurations utilizing the sample application *Online Boutique*[1]. The resource utilization metrics are measured using Prometheus and are evaluated statistically to understand the significance of utilizing a load-aware

[1]https://github.com/GoogleCloudPlatform/microservices-demo

scheduling plugin to improve general CPU utilization. The findings from this research question aim to underline the need for a holistic metrics-aware rescheduling system.

RQ2: Iterative proposal a holistic, metric-aware rescheduling solution

The second research question focuses on developing a holistic, metric-aware rescheduling solution called Kinema and aims to prove the feasibility of such a system. The system includes various subsystems, such as a cluster observer that controls the state and scheduling of Kinema. This optimizer utilizes state-of-the-art optimization techniques to find a suitable rescheduling plan and, finally, an actual rescheduler that converts this plan into action. The output of this research question is a practical system that enables future research and optimization in the Kubernetes rescheduling domain.

1.2.2 Evaluation and experiments

The output of RQ2 shall be compared against the current state-of-the-art in cluster management. Specifically, Kinema will be directly compared with the default scheduler with a default CA as well as a CA with the optimize utilization profile. Two different sample applications are deployed with these configurations. However, only one sample application is tested with both autoscaling profiles, while the other only focuses on the more efficient optimize-utilization CA and Kinema. All result data are collected with Prometheus and evaluated statistically.

2 Background

2.1 Kubernetes

To successfully run containers in production, one has to solve various management challenges. These include scaling the applications horizontally and vertically, managing automatic roll-outs if a new update is released or change is made, and roll-backs if an issue with the newest update occurs. Kubernetes is an open-source platform designed to solve precisely these challenges and was initially developed by Google. Currently, it is actively maintained by the Cloud Native Computing Foundation with the vision to simplify the tasks developers face when orchestrating containers.

In the following, first, a basic overview of the architecture of Kubernetes is given before explaining how workloads are deployed and managed within Kubernetes from a developer perspective and, finally, how these workloads are mapped to the underlying compute infrastructure.

2.1.1 Kubernetes architecture

Figure 2.1 visualizes the basic architecture of Kubernetes that can be split into a control plane or master node and several worker nodes. The total number of worker nodes cannot exceed 5000 nodes with a total of 150000 pods[7]. A pod is the smallest workload unit within Kubernetes and can contain one or multiple containers. To illustrate, one can think of a standard Python Flask webserver that is containerized using a Dockerfile. To deploy this container on Kubernetes, an additional declarative configuration defines a pod. This pod gives the path to the Docker image, including optional resource requests that Kubernetes should reserve to run this workload. Using, for example, the kubectl command line, the developer can instantiate this workload by immitting a request to the Kubernetes API server that runs on the master node. This resource declaration is then stored on the etcd database that stores various state data for Kubernetes. Next, the scheduler tries to find a suitable worker node with enough space to run the container based on the resource requests. Once a suitable node has been found, the kubelet on every work node is informed to initiate the container. While the installation process is running, the developer can query the state of the newly deployed pod using the Kubernetes API server.

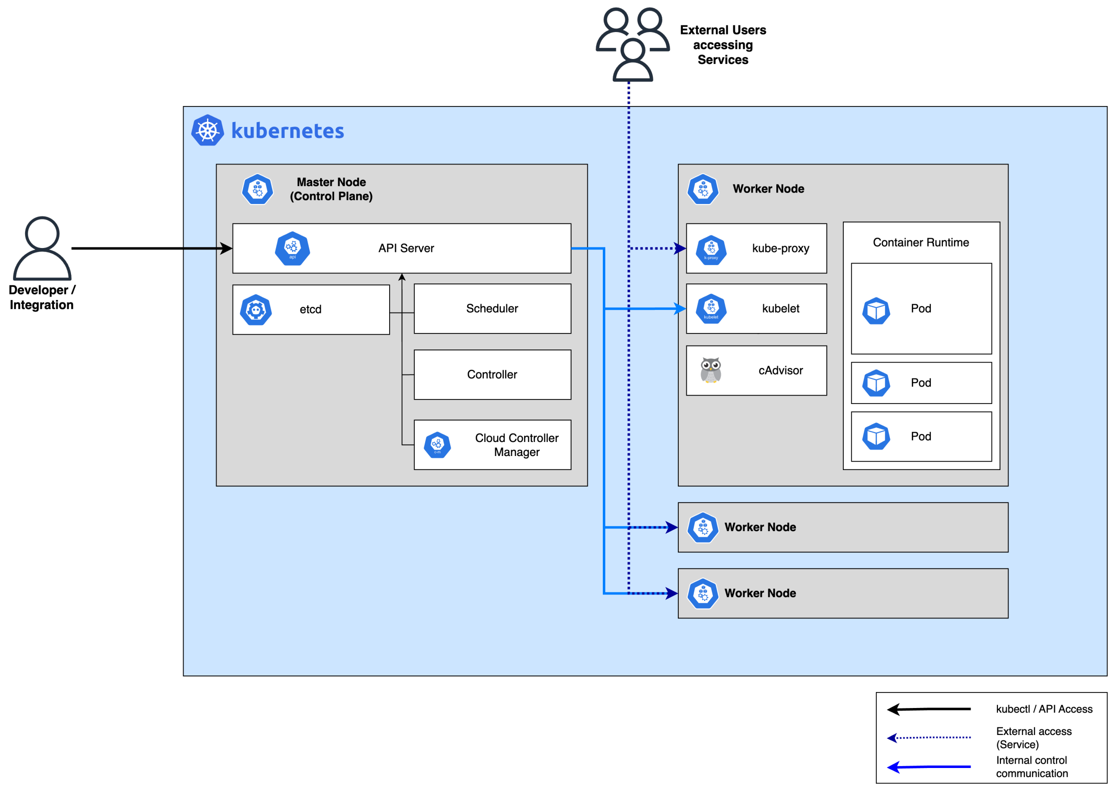

Figure 2.1: Basic Kubernetes Architecture

To enable external users to access this web server as well, Kubernetes utilizes services. These play a vital role in cluster and network communication. A service exposes a set of pod ports and allows other services and even external users on the web to access this pod. A service can have multiple types. Hence a service of type ClusterIp exposes an internal IP address that is only accessible within the cluster, whereas in cloud environments, the service type LoadBalancer exposes a stable, external IP and routes incoming traffic to the associated pods. After all, Kubernetes simplifies networking extensively using the concept of services. In the following sections, other core components of Kubernetes are explained in detail.

2.1.2 Pod Autoscaling

To ensure that the previously introduced sample application can handle fluctuations in traffic, threshold-based autoscaling can be employed. Kubernetes can autoscale in two directions: Adjusting the number of pods by Horizontal Pod Autoscaler (HPA) or scaling the resource of pods using VPA.

HPA

A HPA uses a control loop to monitor the metrics of deployments continuously. If a user-configured threshold is reached, scaling events are triggered, and the number of pods is automatically adjusted. Scaling policies, such as a minimum and maximum number of available pods, can also be enforced using a HPA.

Typical metrics that are utilized in horizontal scaling are CPU and memory metrics. A cAdvisor daemon set[1] continuously monitors the container utilization and reports these to the metrics server. Custom metrics, such as request counts or latency, can also be included. These are directly exposed by the application pods or by so-called sidecars, which run alongside the application containers and collect metrics. A metric endpoint is scraped by a monitoring database such as Prometheus. This database exposes the data via a metrics adapter directly to the metrics server, which is accessed by the HPA to make scaling decisions. Figure 2.2 visualizes the data architecture for both common metrics that are retrieved directly from the cAdvisor and custom metrics that retrieved from Prometheus.

An example of such a custom metric is the number of incoming requests. In the following, a sample application is deployed with a default HPA with a minimum instance count of 2 and a target CPU of 60%. The load balancer is also extended with a metrics threshold of 60 average Requests per second (RPS) per second across all instances. Hence, the autoscaler scales in the latter case, when the average CPU utilization is above 60% of the average RPS count, is above 60% for all instances. If both values exceed their threshold, the HPA considers the higher value. With respect to load patterns, a linear ramp-up from 100 RPS to 1500 RPS in 10 minutes and ramp down back to 100 is tested alongside a burst load pattern that runs for 20 minutes with 500 RPS but has a burst at minute 10 for 2 minutes of 1500 RPS.

The results in Figure 2.3 indicate that solely scaling based on the standard metrics of CPU and memory might not be enough. While the node count stayed consistent throughout the period of the burst in the CPU based HPA, the number of nodes increased in the request metrics driven HPA example. The node count increase is connected to an increased number in pods due to the scale-up, as the RPS threshold was reached. The results also show a scheduling delay encompassing scaling using an HPA. The RPS increase happened just after the burst period was over, and the Virtual user (VU) during the burst period did not benefit from the scale-up of the HPA. The RPS count is higher even though fewer VU are available after the burst because more pods can handle the requests, leading to lower latency. Latency is essential here, as the

[1]A type of deployment that runs on every node

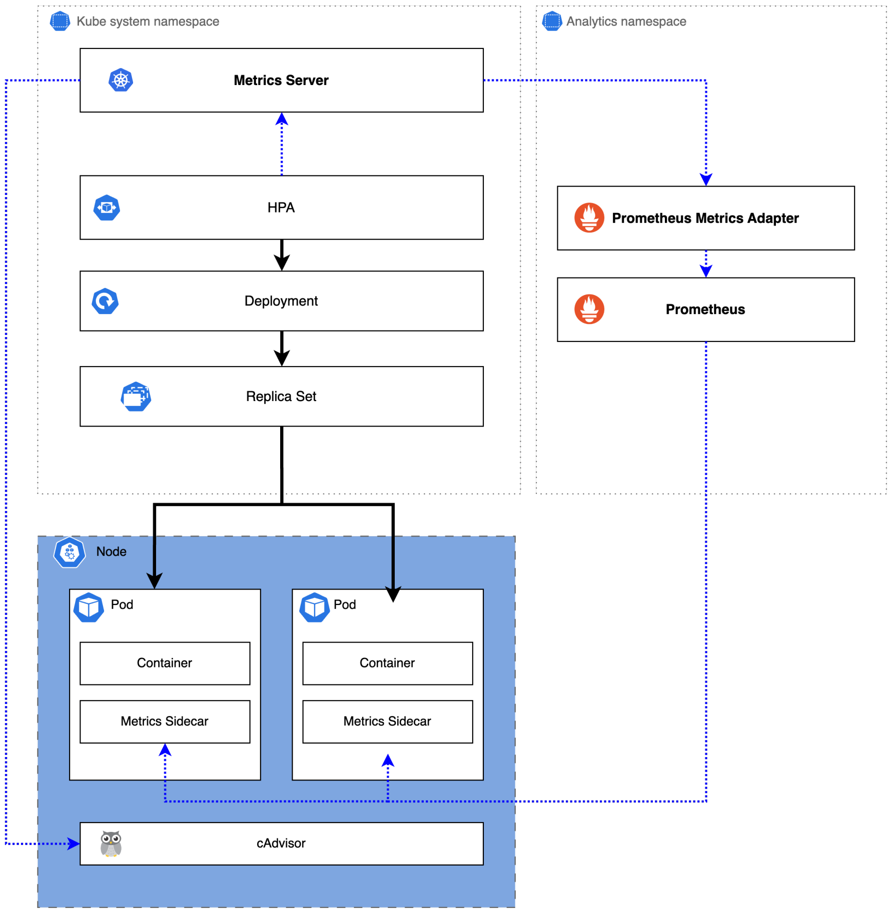

Figure 2.2: Metric HPA architecture

VU only re-executes the same Hypertext Transfer Protocol (HTTP) request as fast as possible.

After all, carefully choosing the proper HPA configuration, which includes metric and respective thresholds, is always application and SLO dependent and can lead to underutilization if done wrong.

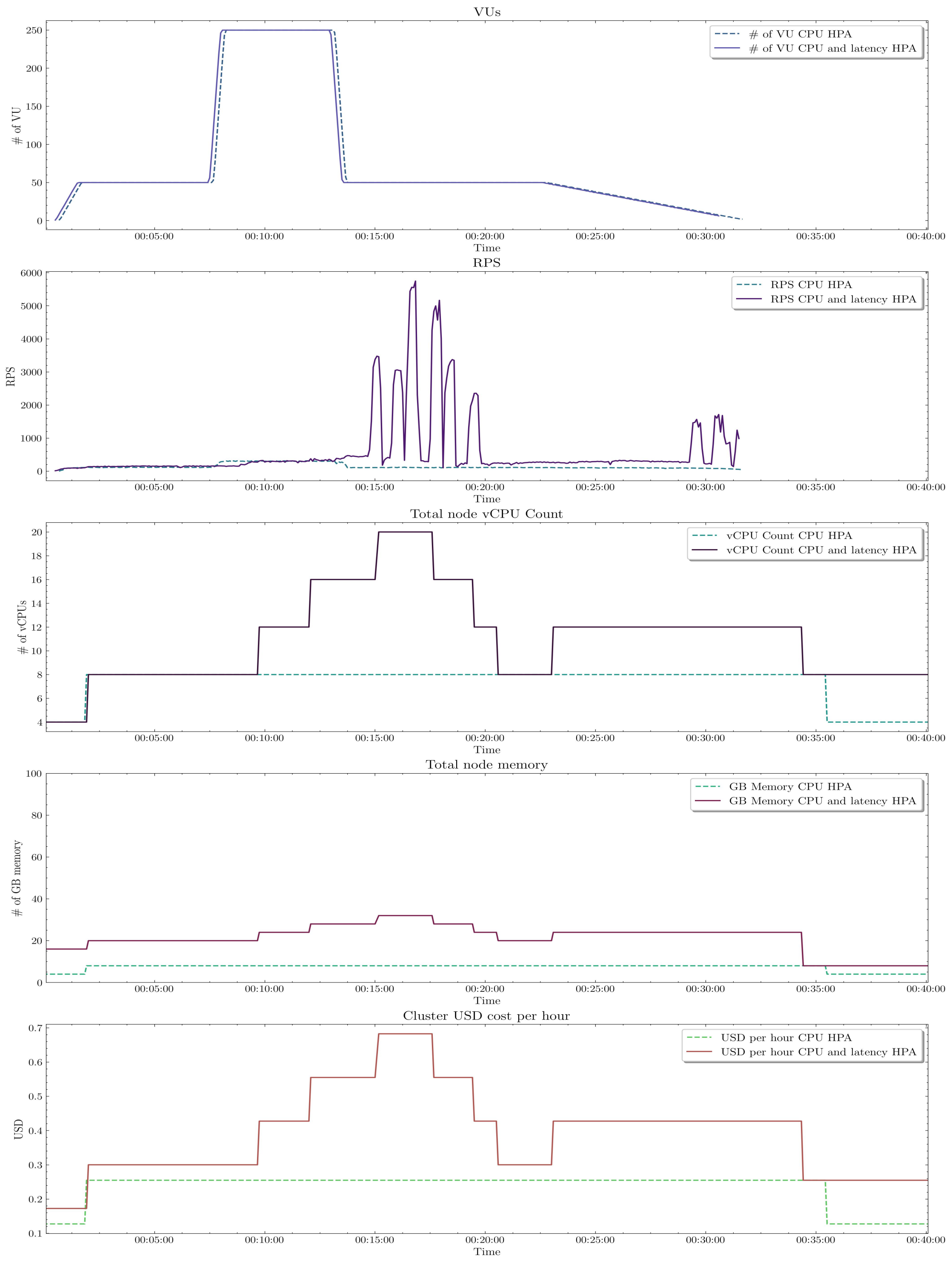

Figure 2.3: Comparison of scaling a sample application solely based on CPU metrics vs. combining CPU and latency metrics

VPA

An alternative scaling direction is vertical, hence scaling resources. A VPA is adjusting the resource requests for individual pods to improve overall resource utilization. This is achieved by the interplay of the three components, which can also be used separately:

The Recommender monitors resource usage and recommends usage-based resources. The recommender can use Prometheus metrics but also works with the default metrics API in the cluster. Based on historical data, resource request recommendations are made. For a workload, the recommender gives multiple recommendations. A lower and upper bound gives the minimum and maximum resources for the workload. A target recommendation provides the optimal level of resources that should be allocated to the pod.

The Updater and the admission plugin are other components in the VPA architecture. The Updater decides which pod should be removed to adjust the resource requests. The eviction of the pod is done using the eviction API. Hence pod disruption budgets are respected. The updater itself does not perform the resource request changes. The admission plugin is used to adjust the pod's resource requests during the admission process of the workload [8].

In the following, the VPA's recommender is used-stand alone to give target resource request recommendations. Based on these, the Kubernetes administrator makes the adjustments manually in the deployments.

To illustrate this, a sample application called Teastore that does not contain resource requests in the current research version is rightsized using the VPA as presented in Figure 2.4 and 2.5 [9]. This is done by running a load test with a spike ramp up from 35 RPS to 400 RPS and comparing the VPA recommendations during these load phases. This is possible as the recommender updates the recommendations values in a regular interval. The final resource requests for the application are presented in 2.1 and will be reused throughout this work. The target recommendations in experiment minute 20 (before the spike) and minute 30 during the spike are used.

Choosing the proper resource request depends on the expected amount of RPS. In this case, a solution would be to choose the smaller resource requests and work with a HPA to scale the application horizontally if the number of RPS increases to ensure there is no resource wastage during idle periods.

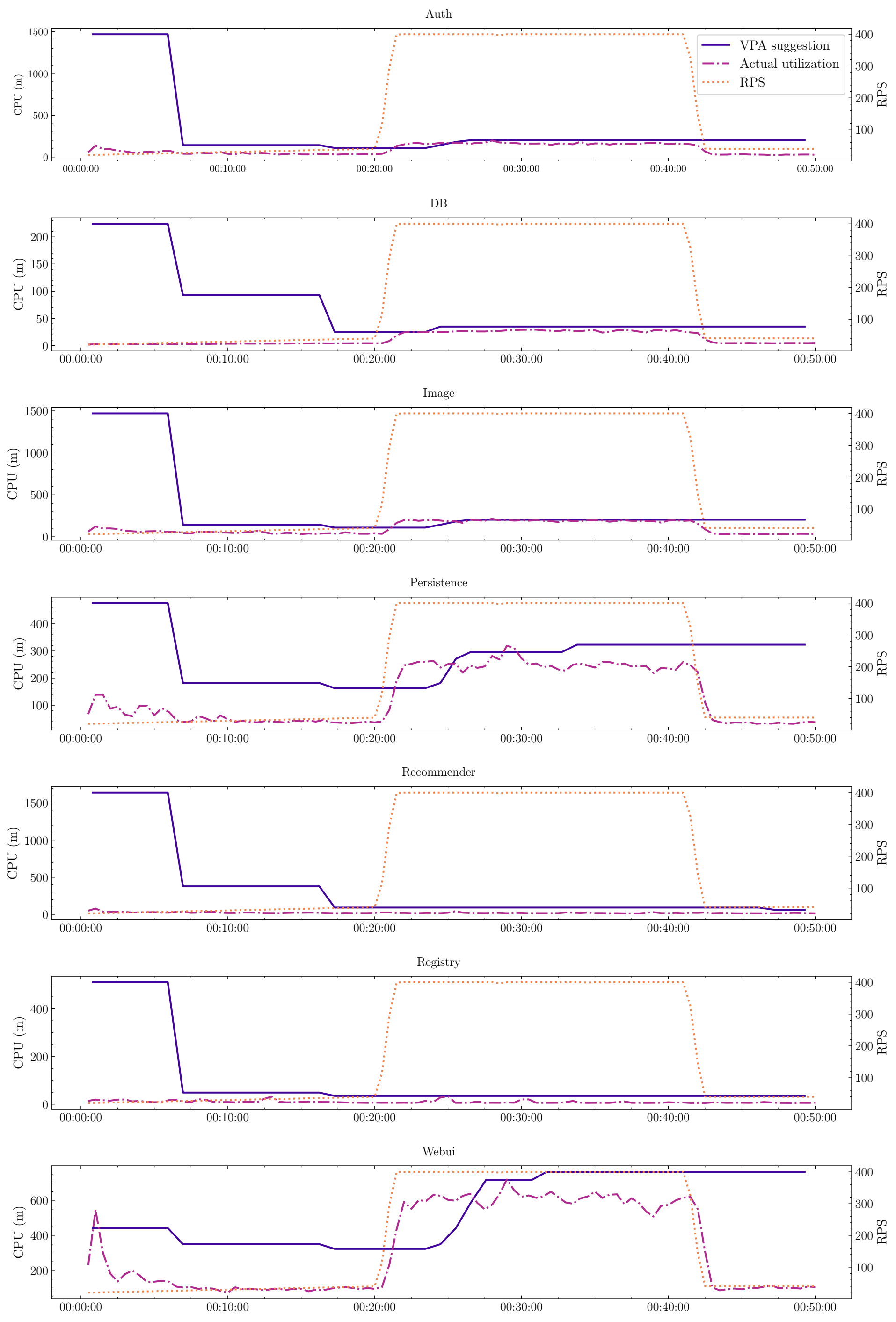

Figure 2.4: Teashop VPA CPU observations

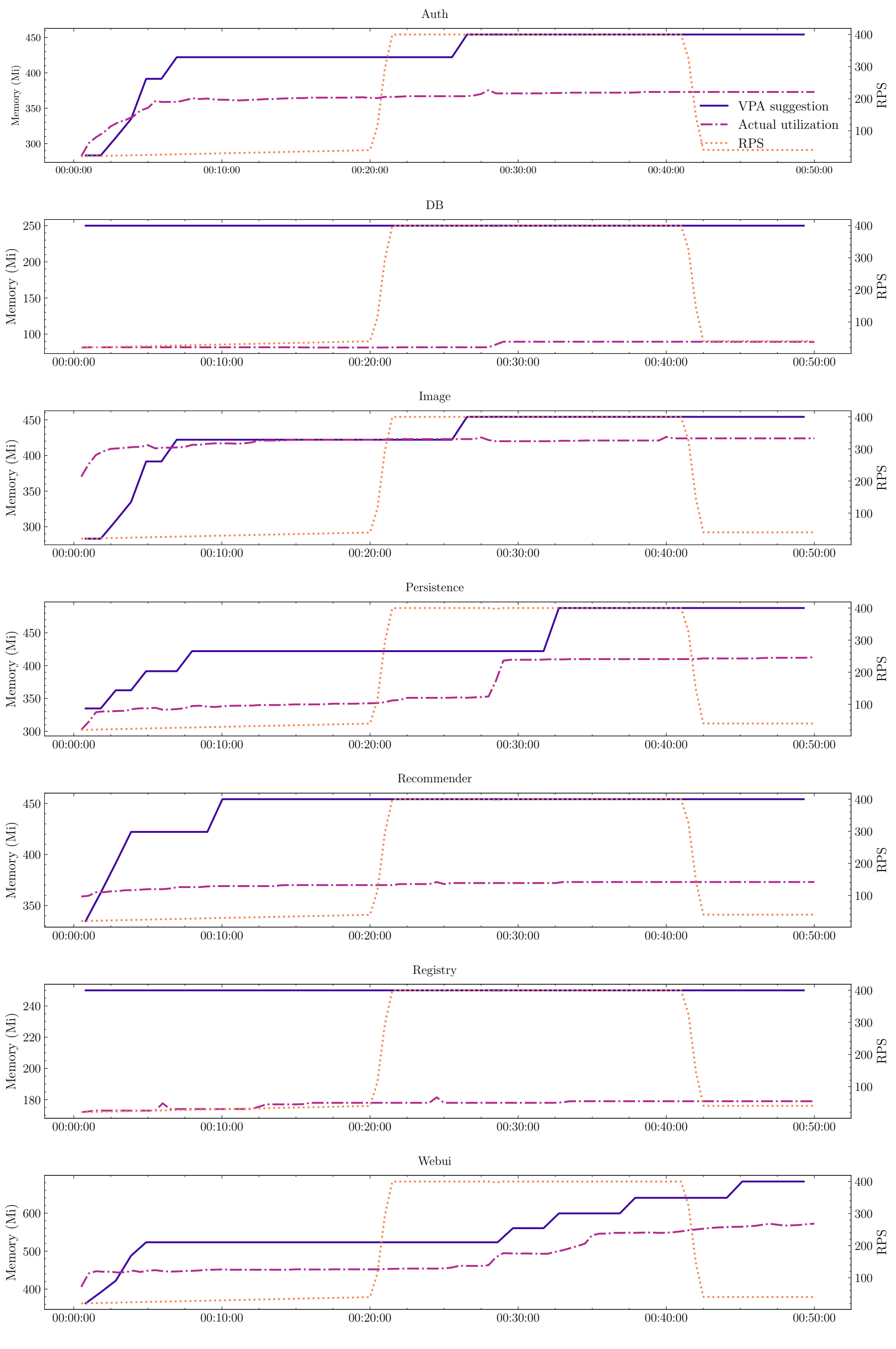

Figure 2.5: Teashop VPA memory observations

Microservice	Small request load (35 RPS)		Increased request load (400 RPS)	
	CPU (m)	**Memory (Mi)**	**CPU (m)**	**Memory (Mi)**
teastore-auth	109	422	203	454
teastore-db	25	250	35	250
teastore-image	109	422	203	454
teastore-persistence	163	422	296	487
teastore-recommender	93	454	93	454
teastore-registry	35	250	35	250
teastore-webui	323	523	763	599

Table 2.1: VPA recommendations for teastore application

2.1.3 Scheduling approaches in Kubernetes

As previously introduced with the sample application, a scheduler is required to manage pod placement onto nodes. The default scheduler evaluates the nodes based on various factors such as available resources or hardware level constraints such as access to a Graphics processing unit (GPU). The scheduler performs scheduling in a two-step process. First, all feasible nodes are filtered based on various constraints. One such filter is the ability to fit the resource requests onto the node. Once a set of nodes has been filtered, each node is assigned a score. One node could, for instance, receive a higher score if the placement on this node improves even pod distribution. The node with the highest score is chosen to install the pod.

The default Kubernetes scheduler is highly customizable through scheduler plugins and extenders. The latter allows incorporating external components or services via HTTP requests into the scheduling pipeline to, e.g., enforce business logic. Besides external components, the Kubernetes scheduler supports plugins activated in the scheduling pipeline's filtering and scoring stages. These allow to extend or even modify the scheduling behavior [10].

If a pod cannot be fit onto existing nodes, it stays in the scheduling queue. A CA watches this scheduling queue and can boot new nodes to fit this pending workload if configured.

2.1.4 Cluster Autoscaling

To address the challenge of efficiently managing workload requirements fluctuations, Kubernetes uses a CA. With the latter activated, Kubernetes continuously monitors the pending workloads and cluster utilization and automatically adjusts the underlying node pools that provide the node compute instances. The CA automatically adds new nodes when demand increases. Conversely, underutilized nodes are automatically removed from the cluster when demand decreases. Depending on the cloud provider, different CA configurations are offered. GKE offers their custom autoscaling profiles "Balanced" and "Optimize utilization" [11].

- The *balance utilization* profile aims to balance the resource usage among nodes, balancing the workloads. This approach prevents resource hotspots and forces an equal workload distribution leading to enhanced cluster stability and performance.
- The *optimize utilization* meets workload demands while optimizing the compute node utilization. The strategy packs workloads more tightly, leading to a smaller cluster size. This essentially results in cost savings.

An alternative that Amazon Web Services (AWS) offers is, Karpenter, a groupless CA. It does not depend on node pools but can choose the right node configuration just-in-time for the pending workloads [12].

The decision for one or the other autoscaling profile has to be made based on application requirements. A workload with bursty traffic is a better fit with the optimized utilization profile, while workloads that require consistent resource availability may have a higher value in the balanced profile.

2.2 Linear programming

2.2.1 Introduction to linear programming and its use in optimization problems

Linear Program (LP) is an essential mathematical tool widely employed in optimization problems. It enables maximizing or minimizing a linear objective function, subject to a set of linear constraints. This powerful technique finds its applications in various industries, including manufacturing, transportation, finance, and logistics.

The primary goal of linear programming is to find the optimal solution that maximizes profits, minimizes costs, or achieves any other desired objective while adhering to the given constraints. It involves using mathematical models to represent real-world problems in a simplified manner. These models consist of decision variables, an objective function, and constraints.

Decision variables are the unknown quantities that the program aims to determine or optimize. They represent the quantities or actions one can control or adjust to achieve the desired outcome. These variables can be continuous, such as production levels, or discrete, such as the number of units to be produced.

The objective function is a linear mathematical expression that defines the goal. It may involve maximizing revenue, minimizing costs, or optimizing any measurable quantity. The objective function is typically formulated as a linear combination of the decision variables, with coefficients representing the importance or contribution of each variable to the objective.

To add restrictions to this program, constraints are used. These can be inequalities or equalities and define the boundaries for the operational field of the decision variables. Typical examples of constraints are resource availability or demand requirements [13, 14].

2.2.2 Mathematical foundations

Mathematically, a linear programming problem can be represented as follows:

$$\begin{aligned}
\text{Maximize (or Minimize):}\ & z = c_1x_1 + c_2x_2 + \ldots + c_nx_n \\
\text{Subject to:}\ & \\
& a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n \leq b_1 \\
& a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n \leq b_2 \\
& \ldots \\
& a_{m1}x_1 + a_{m2}x_2 + \ldots + a_{mn}x_n \leq b_m \\
& x_1, x_2, \ldots, x_n \geq 0
\end{aligned}$$

Here, $x_1, x_2, \ldots, x_n$ represent the decision variables, $c_1, c_2, \ldots, c_n$ are the coefficients of the objective function, a_{ij} denote the coefficients of the constraints, and $b_1, b_2, \ldots, b_m$ are the constants on the right-hand side of the constraints.

Geometrically, the feasible region represents the set of all possible solutions that satisfy the constraints. In a two-dimensional space, the feasible region is typically bounded by lines, forming a polygon. In higher dimensions, it becomes a polyhedron. Each vertex of the feasible region corresponds to a unique combination of variables that satisfies all constraints [15].

2.2.3 Mixed Integer Program (MIP)

MIP is a powerful extension of LP that introduces the capability to handle discrete decision variables alongside continuous variables, including binary variables that take on values of 1 or 0. This enhancement allows for more realistic modeling of real-world optimization problems, where decisions often must be made among a limited set of options or in a binary (yes/no) fashion. This section explores MIP in-depth and highlights its key differences from standard LP.

MIP extends LP by permitting certain decision variables to take on integer and continuous values. Including integer variables, including binary variables, significantly broadens the problem-solving scope, enabling the formulation of more accurate and practical models. In LP, decision variables are assumed to be continuous, allowing for fractional values. However, MIP introduces a level of granularity by enabling the selection of whole numbers or binary choices, which is particularly valuable when dealing with constraints related to quantities, counts, or discrete decision-making.

The main distinction between MIP and normal LP lies in including integer variables, including binary variables, and the resulting combinatorial aspects. In LP, the solution is obtained by finding the optimal values for the continuous decision variables, resulting in fractional or decimal solutions. On the other hand, MIP solutions require identifying the optimal values for both the continuous and integer decision variables, which may include binary decisions. This added complexity significantly increases the computational requirements compared to standard LP.

The presence of integer variables, including binary variables, in MIP problems poses computational challenges because it transforms the situation into a mixed-integer nonlinear program, which is generally NP-hard. Solving MIP problems involves searching through a discrete space of potential solutions, which becomes increasingly complex as the number of integer variables and their possible values increase. The additional combinatorial nature of MIP necessitates using specialized algorithms and techniques tailored to handle discrete decision spaces efficiently.

To solve MIP problems, a variety of optimization solvers employ branch-and-bound or branch-and-cut algorithms. These techniques systematically explore the solution space by partitioning it into smaller subspaces and applying bounds or cuts to eliminate infeasible or non-optimal regions. The solvers iteratively refine the search space until an optimal or near-optimal solution is found. However, due to the computational complexity, solving MIP problems can become challenging and time-consuming, especially when dealing with large-scale models or intricate constraints.

Despite the computational challenges, MIP, including binary variables, offers significant advantages in modeling and addressing real-world optimization problems. It enables the representation of discrete decisions, such as selecting the best facility locations, allocating resources, or scheduling tasks. By incorporating integer variables, including binary variables, MIP allows for a more accurate representation of decision-making processes and offers solutions that align with practical requirements.

In summary, MIP extends the capabilities of standard LP by introducing discrete (integer) decision variables alongside continuous variables, including binary variables. This added flexibility empowers practitioners to model real-world problems more accurately and efficiently. However, including integer and binary variables increases computational complexity, making solving more challenging. Nonetheless, with the aid of specialized algorithms and optimization solvers, MIP provides a robust framework for addressing a wide range of optimization problems that involve discrete decisions and combinatorial constraints [16].

2.2.4 Challenges and limitations

One of the primary challenges faced in linear programming relates to the assumptions made while formulating the problem. Linear programming assumes that the relationships between variables are linear and that a linear objective function and linear constraints can adequately represent the problem. The world, however, is highly nonlinear, rendering linear programming less effective or, in some cases, even infeasible [17].

Another significant limitation of linear programming lies in the requirement for numerical data. The accuracy and reliability of linear programming models heavily depend on the availability of precise numerical values for variables, coefficients, and constraints. However, obtaining such precise numerical data can be challenging in some scientific domains, leading to inaccuracies and uncertainties in the model. Moreover, linear programming models are often sensitive to small changes in input data, which can result in significant variations in the output, making the solutions less robust and reliable.

Additionally, linear programming suffers from the *curse of dimensionality*, which states that the computational complexity of solving the problem grows exponentially as the number of decision variables and constraints increases [18]. This problem hinders the scalability of large linear programs.

Another challenge stems from the assumption of certainty in linear programming models. Real-world problems often involve uncertainty, randomness, and variability. Linear programming models, however, do not account for such delays explicitly, leading to suboptimal solutions or decision-making that do not adequately reflect the inherent risk associated with the problem. Incorporating uncertainty into linear programming models requires more advanced techniques, such as stochastic programming or robust optimization, which can add complexity to the modeling process [19].

Lastly, the available computational power can limit the feasibility of solving large linear programs. Depending on the complexity of the constraints and number of variables, a large quantity of computational power (both CPU and memory) is required. This makes using linear programming to solve problems that require real-time decision-making impractical.

In conclusion, while linear programming has proven its effectiveness in optimization, it is important to address its challenges and limitations, including non-linearity, the requirement for numerical data, the curse of dimensionality, and the required computing power. Overcoming these issues allows more accurate decision-making in various scientific domains and industrial sectors.

3 Related Work

Reducing costs is a core and enduring task in most businesses once services run stable. Not surprisingly, cost reduction by lowering resource wastage has motivated research efforts in the cloud domain for many years. In the context of Kubernetes, resource usage optimization can be done in many areas, such as calculating the optimal number of pods required at a given time or optimizing the cluster autoscaler by providing different node types [20, 21]. The following section compares related work about optimizing node and cost efficiency by adjusting the Kubernetes scheduling approach. Table 3.1 summarizes the results compared to Kinema, a holistic rescheduling solution that is the outcome of this thesis. The comparison checks if the approach supports microservice applications, as some systems in the literature review can only achieve cluster utilization improvements by scheduling batch jobs. Additionally, a check is added if the system mentions general utilization improvements. Last, for a direct comparison with Kinema, a rescheduling column is added to see which systems are based on a rescheduling approach.

3.1 Minimizing resource wastage by improving initial pod scheduling

Multiple proposals reduce resource wastage by implementing a fully custom scheduling implementation. The architecture Stratus, for example, shows an aggressive cost-saving approach by maximizing node utilization with savings of up to 44% compared to the default scheduler. The savings are limited to jobs with predefined lengths that can be scheduled together [22]. A similar generic system is presented by Wu et al. that improves CPU utilization by 28.99% but includes long-running applications [6].

Ding et al. solve the cost reduction scheduling problem by formalizing it as an integer nonlinear model solved by evolutionary algorithms. The authors demonstrate that the example application is scheduled onto a lower number of nodes than when using the default scheduler. Nonetheless, the authors state that this scheduling approach is not yet ready for services with peak workloads [23].

Name	**Approach**	**Microservice support**	**Improves utilization**	**Rescheduling**
Initial pod scheduling				
Stratus [22]		-	+	-
Wu et al.[6]		+	+	-
Ding et al.[23]	MIP	(+)	+	-
Jian et al.[24]	RNN	+	+	-
Load awareness				
Oguachi et al.[25]		+	?	-
Li et al.[26]		+	?	-
Horn et al.[27]	MIP	+	+	-
Fard et al.[28]	MIP	+	+	-
Trimaran [29]		+	+	-
Pod movement				
Rovnyagin et al.[30]	RNN	+	+	+
Descheduler[31]	-	+	(+)	(+)
Kinema	MIP	+	+	+

Table 3.1: Approach comparison to optimize cluster utilization and reduce costs

3.2 Improve resource usage by adding real-time metrics

The default scheduler introduced in section 2.1.3 only considers resource requests when filtering feasible nodes and ignores the actual resource utilization. Motivated by the requirements of edge computing, Ogbuachi et al. propose a custom Kubernetes scheduler that takes real-time parameters about the utilization of each node into scheduling considerations. By doing so, they mitigate the problem of the kube scheduler scheduling pods onto dead nodes, as there is a delay until the default liveness probes discover a dead node. Each parameter is combined with a dynamic weight to adjust the parameter's influence on the node scoring [25].

Instead of writing a fully custom scheduler, Li et al. use the option to extend the default scheduler as presented in section 2.1.3. Not many publications have made use of this extensible plugin framework [3]. The plugin aims to adjust the scoring phase to balance the nodes' I/O and CPU utilization. This is achieved by collecting real-time data from all nodes extracted using Prometheus [26]. A similar approach can also be found in the recently published Trimaran scheduling plugin. The plugin allows ingesting real-time metrics from providers and balances CPU utilization on all nodes [29]. It is already available on a public cloud provider [32].

3.3 Resource usage optimization by moving workloads between nodes

Rovnyagin et al. suggest utilizing reinforcement learning to extend the default scheduler to move workloads between nodes. Experimental results and actual implementations are not given. Still, the usage of reinforcement learning to solve scheduling tasks has been successfully applied in other publications in the context of scheduling in green data centers [30, 33].

An alternative approach to moving workloads is a descheduler which evicts pods from nodes [31]. Reasons to move workloads could be manifold. An example of resource optimization could be the underutilization of a node. The descheduler comes with various strategies that are executed at regular intervals. The workload is not directly rescheduled if the program identifies a pod that should be moved to another node. Instead, the descheduler evicts the pod and lets the scheduler take care of recreating this pod. Currently, the descheduler follows the architecture of the default kube-scheduler and does not access real load metrics when considering if nodes are under-/overutilized. This creates room for further advancements [34].

4 Metrics-driven scheduling

In the following chapter, the previously introduced approach of improving resource utilization by adding real-time metrics to the default scheduler is assessed based on the Trimaran plugin. A particular focus lies in improving resource utilization during idle and scale-down periods.

4.1 Load-metric aware scheduling

Load-aware scheduling is a technique to improve resource utilization by considering the current resource allocation and utilization in the scheduling decisions. One example of load-aware scheduling is the Trimaran scheduler plugin with its target load packaging strategy developed by IBM [35]. By making the scheduler resource-aware, the plugin aims to improve resource usage efficiency in a cluster. This awareness should allow the scheduler to make better decisions when scheduling a workload onto a node in the cluster.

Kubernetes scheduling plugins can be categorized in *in-tree* and *out-of-tree* scheduling plugins, depending on their source code's location in or outside the core Kubernetes repository. Core plugins, such as *NodeResourceFit*, which filters out all nodes that do not have enough resources to handle the pod's resource requests, are in-tree and thus part of the Kubernetes source code. This has various limitations, such as aligning the plugin development with the release cycle of Kubernetes, which leads to slower releases of plugin features. Additionally, the Kubernetes source code has to be modified, which makes it hard to maintain these plugins.

To overcome these issues, out-of-tree scheduler plugins were introduced, with appropriate interfaces on the scheduler side. This allows adding external scheduler code via libraries during compile time. The most popular scheduler plugins are combined as a Golang library and available as Docker image and can hence be used out-of-the-box for scheduling[1]. Trimaran is part of the out-of-tree scheduling repository.

The scheduler configuration allows to control which scheduler plugin is active. This is important, as scheduler plugins might have different interests. In the specific case of Trimaran's target load packaging strategy, the plugin assigns the highest score to nodes

[1]https://github.com/kubernetes-sigs/scheduler-plugins

that are closest to the target utilization level. However, this conflicts with the objectives of other plugins, such as *PodTopologySpread*, which aims to distribute pods across nodes in the cluster evenly. Additionally, there is a conflict with the *NodeResourceLeastAllocated* plugin, which prioritizes nodes with the least resource allocation for pod placement. These plugins have distinct strategies that differ from Trimaran, so they need to be deactivated to ensure successful operation.

A scheduler with out-of-tree plugins can be installed in two ways. One option is to use the out-of-tree scheduler image as a secondary scheduler within Kubernetes. Pods can define the name of the scheduler that is responsible for scheduling in their configuration file. Hence it is possible to separate scheduling concerns with the default Kubernetes scheduler. Alternatively, the default scheduler can be entirely replaced by the out-of-tree scheduler, which is the recommended option for production use-cases [36].

4.2 Trimaran target load packaging

Algorithm 1 Target load packaging [29]

```
1:  $pod\_to\_schedule \leftarrow$ pod from scheduling queue
2:  $target\_utilization \leftarrow$ get target utilization from config
3:
4:  for $node\_to\_score$ in $nodes$ do
5:      $cpu\_utilization \leftarrow \text{GET\_UTILIZATION\_FOR\_NODE}(node)$
6:      $cpu\_request \leftarrow \text{GET\_CPU\_REQUEST}(pod\_to\_schedule)$
7:      $predicted\_utilization \leftarrow cpu\_request + cpu\_utilization$
8:
9:      if $predicted\_utilization \leq target\_utilization$ then
10:         score $\leftarrow (100 - target\_utilization)\frac{predicted\_utilization}{expected\_utilization} + expected\_utilization$
11:     else if $target\_utilization \leq predicted\_utilization \leq 100$ then
12:         score $\leftarrow \frac{expected\_utilization(100 - predicted\_utilization)}{(100 - expected\_utilization)}$
13:     else
14:         score $\leftarrow 0$
```

Algorithm 1 shows the details of Trimaran's target load packaging plugin. The plugin is part of the scoring stage of the scheduler, which calculates a score for each node with respect to the current pod that is being scheduled as in line 1. The following line 2 gets the target node CPU utilization from the scheduler plugin's configuration. Currently, target load packaging only supports CPU as a metric.

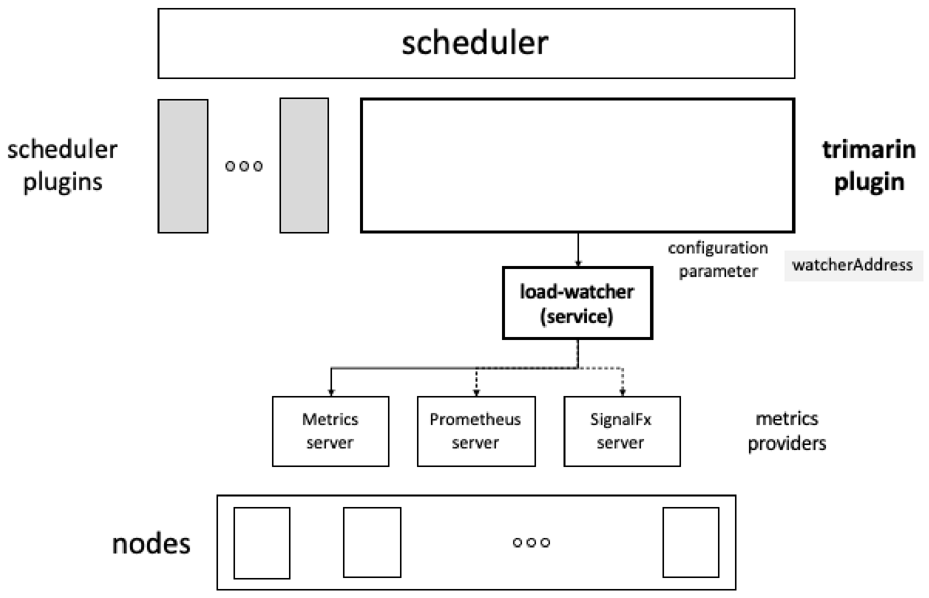

Figure 4.1: Trimaran load aware scheduling plugin architecture [29].

The scoring algorithm assigns a score for each node in the next stage. For this, it first requires the current CPU utilization metrics for the node as presented in line 5. Trimaran can use either an internal load-watcher library or a load-watcher service to fetch the latest utilization metrics for nodes. Load-watcher is an externally maintained service outside of Trimaran. Figure 4.1 shows the architecture when Trimaran uses the load-watcher as an external service to fetch the metrics data. Prometheus and SignalFx are supported as metrics providers besides the default Kubernetes metrics server. If metrics are unavailable, the utilization calculation function uses the resource definitions of each pod on the node as a fallback. The CPU utilization here is predicted based on the request limits of the pod. If these do not exist, the resource requests are used. Alternatively, if these also do not exist, Trimaran allows setting a default request value in the configuration.

Before calculating a score, the utilization of the node has to be predicted if the new pod were to be scheduled onto this node. This is done by adding the resource requests to the current node utilization in line 7. The final score is calculated based on the predicted utilization as of line 9 and visualized in Figure 4.2. If the predicted utilization is below the target utilization, the score should increase the closer the predicted score is to the target utilization. Alternatively, the score should decrease linearly if the predicted utilization is above the target utilization but still below 100%. All predicted utilizations

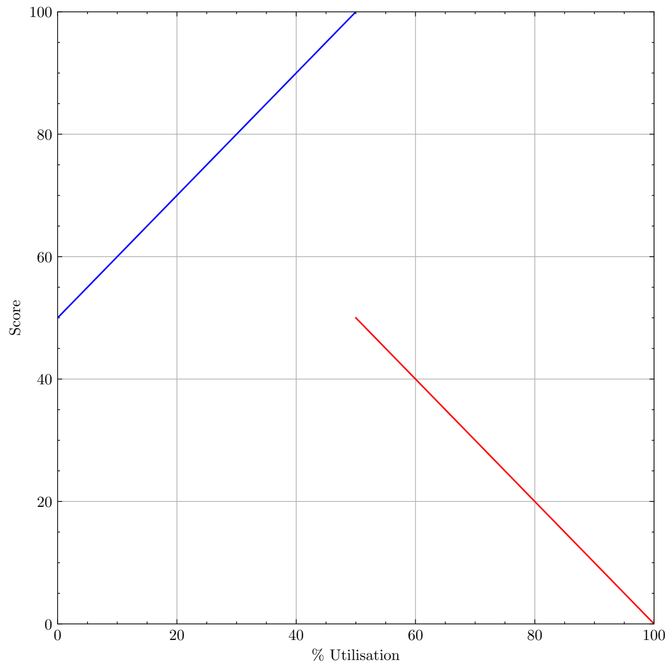

Figure 4.2: Trimaran scoring based on a target utilization of 50% [29]

above 100% should receive a score of 0.

Finally, the calculated scores are returned to the scheduler, calculating an average of all scoring plugins. The node with the highest average score is chosen for pod binding.

4.3 Experimental comparison

The following experiment compares the default scheduler with the scheduler with an activated trimaran target load packaging plugin. Additionally, both autoscaling profiles - balanced and optimize utilization - are used. The experimental cluster uses a HPA to scale the sample application with a target CPU of 60%. The metrics provider for Trimaran is Prometheus, with a target node utilization of 60%.

The workload pattern consists of two linear scale-ups. The first scale-up takes nine minutes and peaks at 300 VUs after four minutes. The second scale-up starts sequentially and takes 16 minutes, with a peak after 10 minutes at 1000 VUs.

The experiment results are summarized in Table 4.1 and Figure 4.3 A one-sided t-test is used to evaluate the results to determine whether a significant difference in CPU utilization between the Trimaran and default approaches exists for both autoscaling constellations. A one-sided (also called one-tailed) t-test is employed when a specific

	During load test					**Post load test**			
Name	avg. RPS	avg. CPU	avg. memory	avg. cost	min. cost	avg. CPU	avg. memory	avg. cost	min. cost
Trimaran comparison									
Default (balance)	371	49%	73%	0.35	0.13	8%	52%	0.39	0.13
Default (optimize)	370	44%	86%	0.31	0.13	14%	55%	0.17	0.13
Trimaran (balance)	442	40%	86%	0.29	0.13	8%	53%	0.23	0.13
Trimaran (optimize)	414	54%	74%	0.34	0.13	10%	56%	0.23	0.13

Table 4.1: Experimental comparison of the balance and optimize utilization autoscaling profiles with the default and load-aware scheduler

	Load Phase	
Name	**t-statistic**	**p-value / 2**
Default CA	0.87	0.10
Optimize utilization CA	0.312	0.189

Table 4.2: Statistical hypothesis test of a one-sided t-test to evaluate the significance of trimaran target load packaging on the CPU utilization levels

hypothesis about the direction of the mean difference is made.

Hypothesis: Let μ_t represent the mean CPU utilization of trimaran and μ_d the mean CPU utilization of the default scheduler. The null hypothesis (H0) states that there is no significant difference in the utilization levels when using Trimaran i.e., $\mu_t = \mu_d$. In contrast, the H1 hypothesis suggests that a higher CPU utilization can be reached by utilizing Trimaran as a scheduling plugin. A significance level with $\alpha = 0.05$ is used. Suppose the $p\_value/2$[2] returned by the t-test is below the significance level. In that case, the H0 hypothesis can be rejected with the result that utilizing Trimaran causes a higher CPU utilization with respect to the current load-test and experiment setup.

As summarized by Table 4.2, the *p_values* for both hypotheses are larger than the significance level of 0.05. Therefore, there is not enough evidence to reject the null hypothesis. This suggests that there is no significant difference in the measured CPU utilization levels between a scheduler that utilizes Trimaran and a scheduler that does

[2]As a one-sided t-test is performed, the p-value has to be divided by two

not utilize Trimaran. As all pods get scheduled onto one node in the post-load phase, evaluating the statistical significance of Trimaran does not bear any value. In fact, the scoring stage is not reached in the scheduling pipeline, as only one node is available.

4.4 Summary

All configurations in the experiment reach the same conclusion to run the workloads on one 4-core node during idle times. This state is reached faster when employing a CA with optimize utilization autoscaling profile. The final utilization levels of this node remain low, with an average utilization below 17% CPU and 60% memory. In an ideal scenario, even smaller node sizes could be chosen to save costs during this idle period. This is not possible with the current state-of-the-art. In the subsequent chapter, a pioneering system called Kinema is introduced, which aims to fill this gap. This solution focuses on improving resource utilization through intelligent workload and node rescheduling with a holistic cluster and cloud provider view.

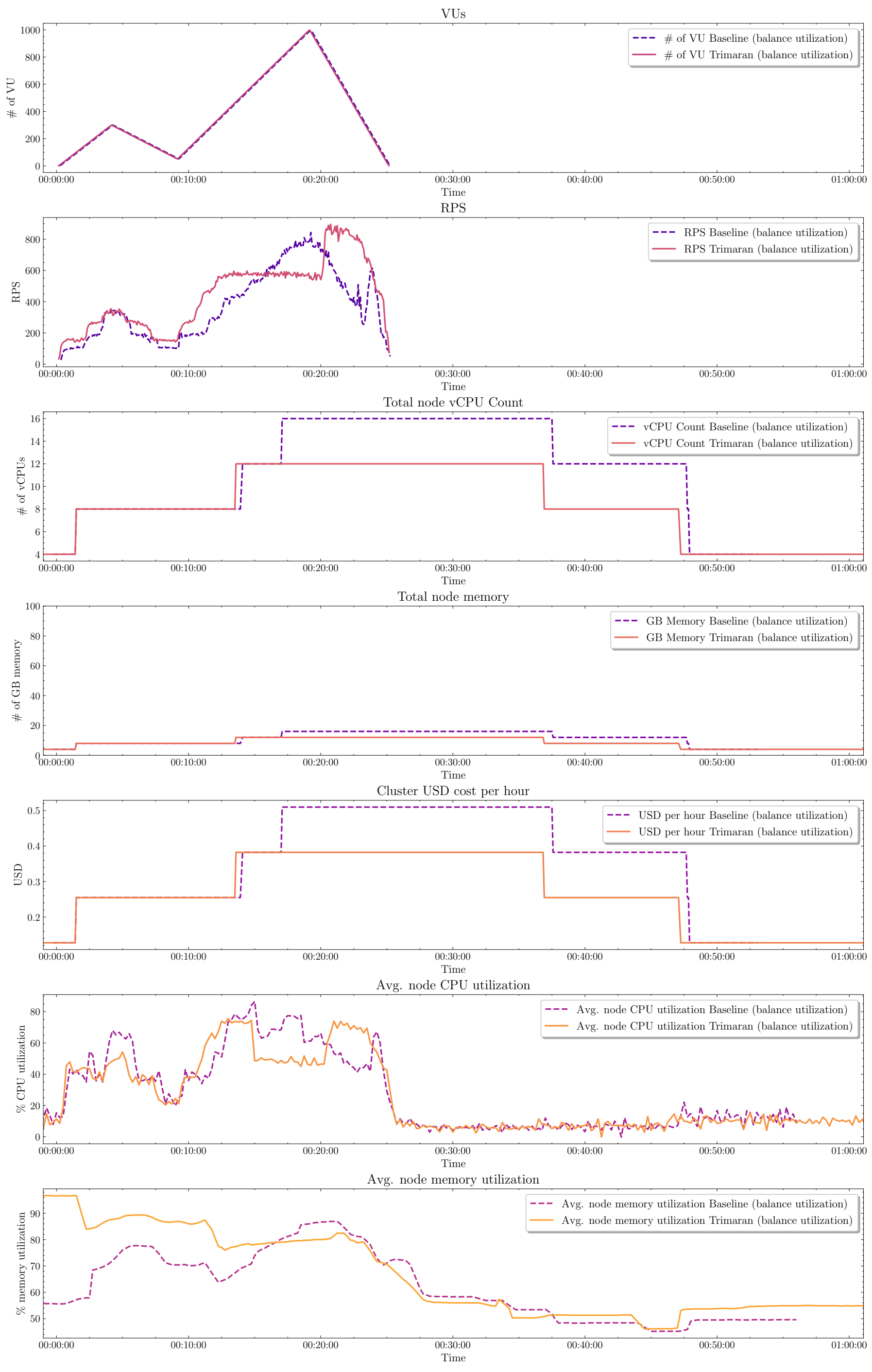

Figure 4.3: Comparison of the Trimaran target load packaging plugin vs. the default scheduler using a balance utilization CA profile

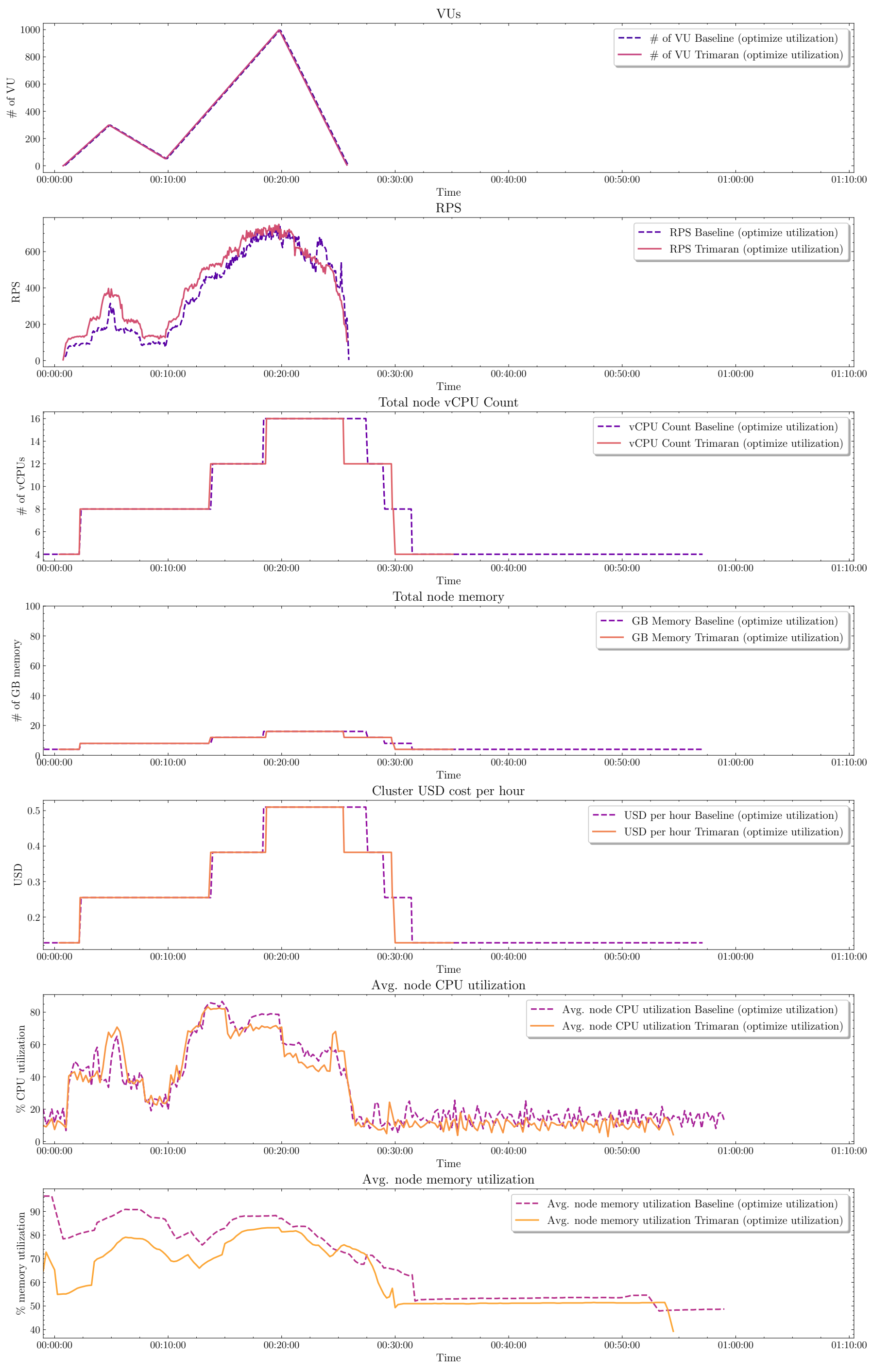

Figure 4.4: Comparison of the Trimaran target load packaging plugin vs. the default scheduler using a optimize utilization CA profile

5 Kinema - a holistic, metric aware Kubernetes rescheduler

5.1 Initial prototype design

The initial rescheduler design is based on the idea of the Kubernetes descheduler, which allows activating multiple strategies, such as removing pods automatically after a specific time as presented in section 3.3 [31]. To ensure resource utilization is high on all nodes, a new strategy *metric deschedule* is added to the descheduler, which automatically removes pods from nodes with low utilization.

5.1.1 Metrics deschedule approach

As shown in algorithm 2, the last node utilization window is fetched in line 1 from the same load-watcher microservice used in section 4.1 by Trimaran. In line 6, the node with the lowest CPU utilization is retrieved. A threshold defines when a node is considered to be underutilized. In the experiments, the threshold has been set to 30%. If the node with the lowest utilization is below this threshold, all pods are evicted as presented in line 11.

To ensure a certain number of pods is always available, Kubernetes offers so-called Pod Disruption Budgets (PDBs). These allow control of pods' disruption during operational events and help ensure the high availability and reliability of the application. The number of simultaneously unavailable pods can be specified in these disruption budgets [37]. The default eviction method used automatically respects these PDBs and ensures the metrics descheduler does not violate the high availability of applications.

The respective controller automatically recreates the evicted pods, and the default scheduler with an active target load packaging plugin starts to schedule the pods. In an optimal case, the evicted pods would be moved to other nodes to achieve high utilization. Their initial node should eventually be removed automatically by the CA due to low node utilization.

Algorithm 2 Evict Pods from Underutilized Nodes

```
 1: $metrics \leftarrow$ GetNodeMetrics
 2:
 3: lowest_utilization $\leftarrow +\infty$
 4: lowest_utilization_node $\leftarrow$ null
 5:
 6: for node $\in$ node_metrics do
 7:     if node.utilization $<$ lowest_utilization then
 8:         lowest_utilization $\leftarrow$ node.utilization
 9:         lowest_utilization_node $\leftarrow$ node
10:
11: if $lowest\_utilization \leq threshold$ then
12:     $pods \leftarrow$ GetPods($lowest\_utilization\_node$)
13:     for each $p \in pods$ do
14:         EvictPod($p$)
```

5.1.2 Issues with initial prototype design

The descheduler plugin can successfully find and evict pods from an underutilized node. Once the scheduler starts to schedule the evicted pods onto nodes again, most of the workload is visible on other moderately utilized nodes. At the same time, some pods begin to appear again on the node with the lowest utilization, which prevents the CA from detecting that this node is underutilized. One would expect the target load packaging approach not to schedule pods onto a low utilization node.

Inspecting the logs of the scheduler, it becomes apparent that the target load packaging plugin is never active, as the filtering stage that is performed first filters out all nodes but one - the node that the pods got descheduled from. The default plugin NodeResourceFit, which filters out all nodes that do not have enough capacity to fit the pod, only returns the single node with enough capacity.

Here, the discrepancy between node usage and allocation based on requests becomes apparent again, which initially motivated the thesis. One idea would be to disable the NodeResourceFit plugin. Once the pods are scheduled onto the node, the kubelet will not allow the pod to start as it is out of resources due to the size of the resource requests.

Simply taking the lowest utilized node into consideration and descheduling the running workloads is not enough. Taking a broader look at all of the workloads is necessary to enable the best cluster utilization and hence the best economic outcome.

5.2 Revised architecture

5.2.1 Architecture overview

The initial rescheduling experiments sparked the idea of a rescheduling system that can optimize node utilization based on a holistic cluster understanding. This means broadening the view from an underutilized single node to the entire cluster and the underlying cloud provider. Figure 5.1 depicts this architecture of the rescheduling system Kinema. At the core, Kinema has access to all workload placements in the cluster by leveraging the Kubernetes API. The system itself runs as a system-critical pod within the kube system namespace.

Apart from access to the current mapping of workloads onto nodes, the system also has access to current resource utilization that Prometheus actively scrapes. The resource utilization for each workload is provided by the daemon set cAdvisor, which monitors various parameters of the running containers. Additionally, current resource recommendations can be calculated using the available data from Prometheus and the VPA.

Kinema leverages an internal cloud controller to access the underlying cloud infrastructure seamlessly, granting comprehensive insights into available compute configurations and their corresponding pricing details. At present, Google Cloud is fully supported as a trusted cloud provider. However, owing to the cloud controllers' highly modular nature and architecture, expanding compatibility with other platforms is readily achievable.

By combining the knowledge of current workload mappings, resource utilization, and available compute resources offered by the cloud provider, the current best configuration workload and node configuration for the cluster can be calculated and finally put into action using a rescheduling approach. This approach relies on well-established Kubernetes techniques, specifically rolling updates. These updates are performed with configurable maximum unavailability of services, guaranteeing no disruptions. This methodology maintains service continuity throughout the update process, ensuring a seamless transition without any negative impact on user experience.

The following section provides a detailed insight into the optimization process of Kinema.

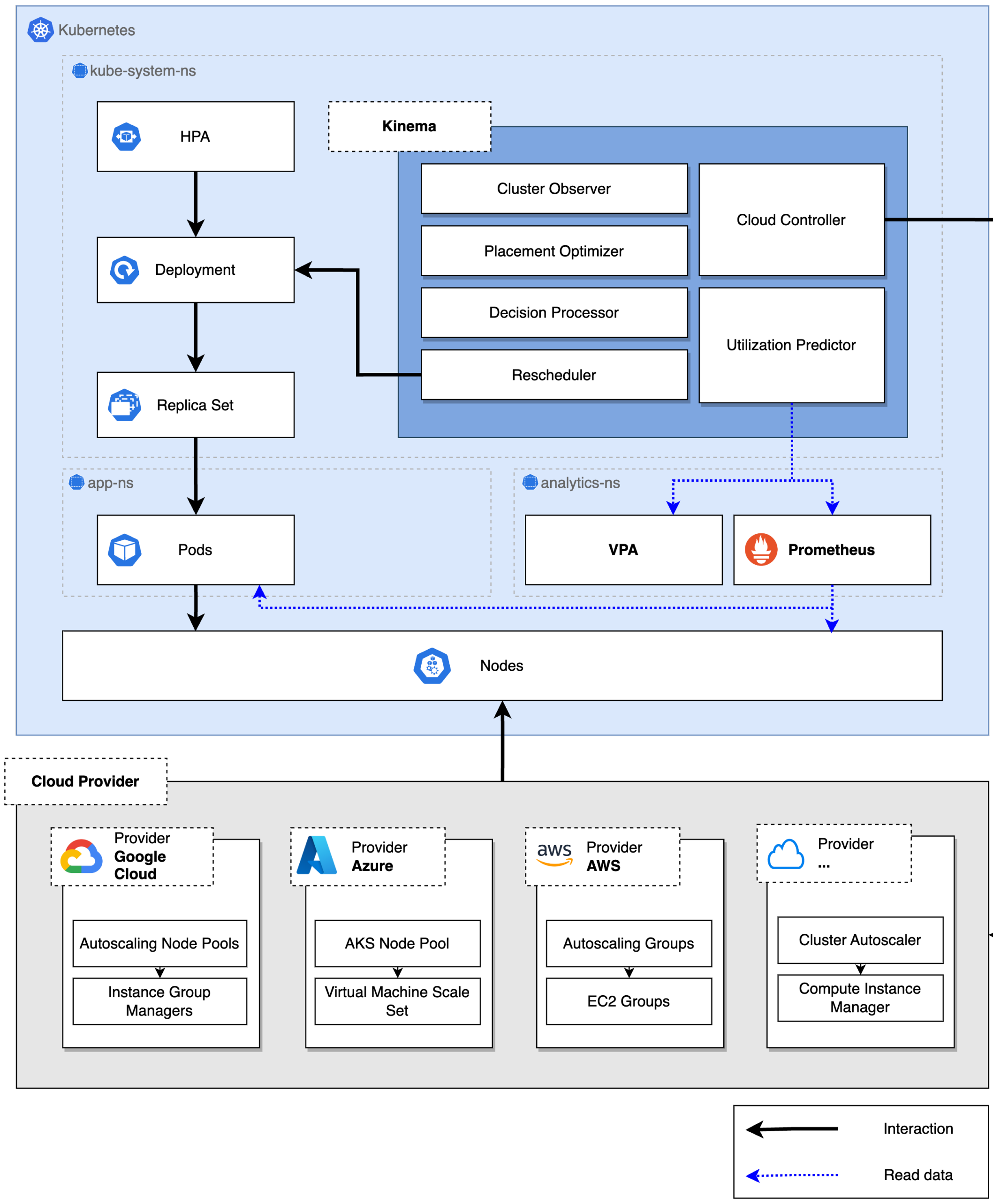

Figure 5.1: Kinema architecture overview

5.2.2 Rescheduling sequence

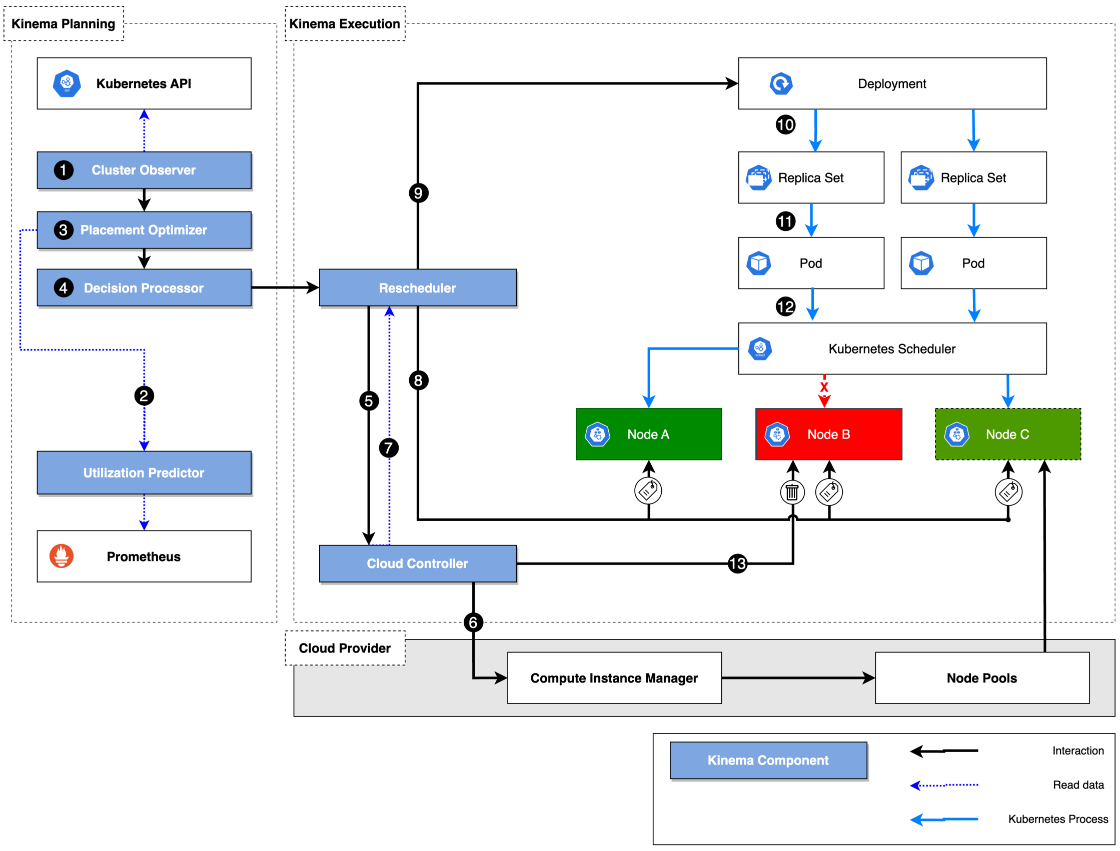

Figure 5.2: Rescheduling sequence

The Kinema process can be divided into the planning and the execution phase as presented in Figure 5.2. While this section primarily describes the rescheduling process, the next section will explain the major components in detail.

The cluster observer ❶ utilizes the Kubernetes API to store a local state of the cluster and is responsible for initiating and managing the current state of the Kinema rescheduling process. A new rescheduling iteration is initiated if no asynchronous operations, such as deletions of nodes or rolling updates, are active and the cluster is stable. The cluster observer's primary objective is to identify this stable time window in the cluster where no components are actively changing. It utilizes the Kubernetes API to query the current state of Replicasets (RSs) and HPAs to achieve this. If an RS does not have the desired number of resources in a ready state or if the HPA exceeds the predefined threshold, the optimizer will not initiate a new iteration. Instead, it monitors the cluster's stability, repeatedly querying the components until no active changes are detected.

Once a new iteration is initiated, the utilization predictor ❷ accesses the latest utilization metrics for memory and CPU for each pod. Based on these utilization metrics, a simple linear regression is performed to evaluate each deployment's current and future resource usage within the next 15 minutes. This window can be configured. The optimizer can use these resource utilization metrics to adjust the resource requests for each deployment by a preconfigured factor, currently set to 0.1 by default. The value was chosen, aiming to strike a balance between conserving initial resource requests and ensuring optimal performance. However, future experimentation and analysis may be necessary to fine-tune this value further based on specific use cases. Apart from using the internal utilization predictor, which works on a small time frame, Kinema can also use the recommendations provided by the VPA. These resource recommendations are passed to the optimizer alongside the cluster state.

The placement optimizer ❸ aims to find the most cost-efficient placement of pods on currently running and potential new virtual nodes from the available node pools. The objective of the optimizer can be configured, so setting an objective to keep the node CPU utilization below a certain threshold during daytime and optimizing for cost efficiency during nighttime is possible. Under the hood, the optimizer utilizes an MIP, as presented in section 2.2.3, to minimize the cost per hour for the cluster. To speed up the calculations of the MIP, the current cluster state, with its mapping of workloads to nodes, serves as starting point for the optimizer. Based on this starting point, it tries to find a better solution by adding and removing different node configurations. Once an optimization iteration is completed, it returns a list of optimized nodes and the calculated value for the objective. In the case of a cost optimization objective, the value returned would be the optimal cost per hour for running the node configuration.

This configuration is evaluated by the decision processor ❹ that eventually decides if the calculated plan is implemented. It henceforth bridges the Kinema planning with the Kinema execution process. The decision processor first calculates the number of changes that occurred in the cluster while the optimizer calculated a solution. Changes might include booted or removed nodes or new workloads appearing, e.g., by a scale-up of the HPA. If the sum of the individual changes[1] outweigh a predefined configuration - by default 10% - the optimized configuration is omitted, and the rescheduling process restarts with the cluster observer. If the changes within the cluster are below the target, the optimization outcome is compared against the current cluster state. In the case of cost optimization, this can be done by calculating and comparing the cost per hour for

[1]A change is e.g. the removal of one node or the appearance of a new pod.

the current and optimized cluster state. If the cost can be optimized by more than 10%, the plan is passed on to the rescheduler.

The first phase of the rescheduling process is responsible for booting new nodes. For this, the rescheduler ❺ passes the required machine configurations to the cloud controller, which issues a request to the respective vendor ❻. Currently, only a cloud controller for Google Cloud is implemented. But the software architecture allows adding of multiple other vendors. The cloud controller monitors the state of the boot operations. To speed up the process, the bootup operation for Google Cloud is directly performed on the compute manager instead of the Kubernetes interface, as the latter only allows to boot one node type at a time. Moreover, the boot operation has to be completed before a new operation can be started. By directly calling the compute manager, this process can be parallelized. Once the boot operations are completed, the rescheduler waits for the new nodes to appear in the Kubernetes cluster before continuing ❼.

In a typical fashion for Kubernetes, labels are used to categorize all nodes. Labels are key-value pairs that are stored with each node's metadata. The rescheduler generates two types of key-value pairs ❽ for optimal nodes, which appear in the optimizer plan, and one for all other nodes that will eventually be removed from the cluster. The keys stay the same throughout every iteration, whereas the value is generated with each initiation of a rescheduling process.

Once all compute resources have been patched, the rescheduler updates all deployments ❾ that currently run on nodes that are not optimal for the cluster state. Here, the node affinity is updated to prefer running on nodes with an optimization label. The deployment automatically generates a new RS version ❿, which starts a rolling update of pods ⓫. These newly created pods appear in the scheduling queue of the cluster scheduler ⓬. They are scheduled onto the running nodes that have an optimal label.

Once the roll-out process runs, the cloud controller regularly queries the state of all nodes with an expiry label ⓭. Once the node is empty, it is removed from the cluster. Once all unnecessary nodes are removed, the cluster observer can initiate a new rescheduling iteration.

5.3 Kinema components

The following section describes the core components of Kinema in detail.

5.3.1 Cluster Observer

Algorithm 3 Cluster observer iteration

```
1:  while true do
2:      $cluster\_state$ ← get_kubernetes_cluster_state()
3:      $run\_kinema\_optimization$ ← true
4:
5:      if len($cluster\_state.kinema\_expiry\_nodes$) > 0 then
6:          for $node \in cluster\_state.kinema\_expiry\_nodes$ do
7:              if len($node.workloads$) == 0 then
8:                  delete_node(node)
9:              else
10:                 $run\_Kinema\_optimization$ ← false
11:
12:     if len($cluster\_state.unschedulable\_pods$) > 0 then
13:         autoscale()
14:     else if cluster_stable(cluster_state) and run_Kinema_optimization then
15:         optimize_placement()                    ▷ Trigger placement optimizer
```

The cluster observer's lifecycle is presented in algorithm 3. Every iteration starts by fetching the current cluster state as in line 2 from the Kubernetes API. This cluster state includes all pods in the configured namespaces, all running nodes, and Kubernetes objects such as the HPA and RS. Additionally, nodes are sorted into groups (optimal nodes, nodes to be expired) based on previous Kinema iterations. Line 5 checks if nodes that should be expired still exist and then, for every node, if all workloads have successfully been moved to other nodes by rolling updates. If so, the node is deleted as in line 8. If nodes are still running workloads, the previous Kinema rescheduling iteration is incomplete, and a new iteration will not be started as shown in line 10

Next, in line 12, the cluster observer checks if the scheduling pipeline includes any unscheduled pods. If this is the case, an autoscaling iteration will be initiated, which is explained in detail in section 5.4.

If the previous Kinema iteration is complete, indicated by the *run_kinema_optimization* variable in line 3, the cluster has to be checked for stability as of line 14 before initiating a new iteration in line 15. In this case, a cluster is stable when the HPAs are not above their thresholds and the RSs desired nodes are equal to the number of nodes that are ready as presented in algorithm 4 If the cluster is not stable, there are still workloads moving and being booted. In this case, a Kinema iteration that adds additional movement due to rescheduling can cause conflicts.

Algorithm 4 Cluster stability check

```
1: function IS_CLUSTER_STABLE($cluster\_state$)
2:     for $hpa \in cluster\_state.hpas$ do
3:         if $hpa.target\_value < hpa.actual\_value$ then
4:             return false
5:
6:     for $rs \in cluster\_state.rs$ do
7:         if $rs.desired\_count \neq rs.current\_count$ then
8:             return false
9:     return true
```

5.3.2 Utilization Predictor

To improve the placement decisions of the placement optimizer, current and predicted future utilization trend values for memory and CPU are calculated before the optimization is run for every deployment. If recommendations exist, these are considered new resource requests during the placement optimization process.

Algorithm 5 describes the detailed utilization prediction approach. At the core, the utilization predictor queries the latest utilization metrics for all running pods of a workload from Prometheus. This data is fit to a linear regression model based on scikit-learn library. Linear regression is a statistical modeling approach that allows mapping a dependent variable to one or more independent variables. In this case, it is a trend predictor for the utilization metrics. By fitting the model, the predictor aims to find a best-fit line representing the relationship between the most recent utilization metrics and time. A configurable parameter allows changing the time taken into consideration in the Prometheus query to train the model. By default, 15 minutes are used. Once the model has been fit, the mean resource utilization for the next fifteen minutes is calculated using this model.

A label per deployment indicates the percentage that the utilization predictor can adjust the resource requests based on future utilization trends. By default, this percentage is set to 10%. The calculated resource recommendations are added to each workload and are then used by the placement optimizer as new resource requests.

An alternative to running the internal utilization prediction method within Kinema is the usage of the VPA recommendations. The VPA continuously monitors the respective workloads and utilizes its internal recommender to provide current resource utilization recommendations. This solution is only valid, though, if there is active usage for more than 24 hours on the application. These are accessible by the cluster observer and can also be used as memory and CPU recommendations within the optimizer.

Algorithm 5 Utilization Predictor

```
function PREDICT_UTILIZATION(metric_type, deployment)
    pods ← cluster_state.get_pods_for_deployment(deployment)
    adjustment_factor ← deployment.resource_adjustment_factor
    metrics ← query_prometheus_metrics(pods, learning_interval)

    $X$ ← metrics.timestamps                                  ▷ Extract features
    $y$ ← metrics.values                                      ▷ Extract target

    model ← LinearRegression()                                ▷ Fit linear regression model
    model.fit($X, y$)

    future_X ← np.array([[X[-1][0] + 15]])                    ▷ Predict next 15 minutes
    future_y ← model.predict(future_X)

    average_utilization ← mean(future_y) × $adjustment\_factor$
    return average_utilization

for deployment in cluster_state.deployments do
    cpu_utilization ← predict_utilization("cpu", deployment)
    memory_utilization ← predict_utilization("memory", deployment)
    deployment.recommendations ← (cpu_utilization,memory_utilization)
```

5.3.3 Placement optimizer

The placement optimizer is responsible for finding the best number and configuration of nodes to run the current workload. To do this, relying solely on the currently visible nodes to the Kubernetes API server is insufficient. Instead, the optimizer creates a virtual cluster representation. In this virtual environment, it is possible to schedule pods on currently available nodes and on so-called virtual nodes, which the rescheduler will boot up if they are part of the end result of the placement optimizer.

Virtual nodes

Algorithm 6 Virtual node calculation

```
function GET_WL_RESOURCE_REQUESTS(workloads)
    total_memory_request ← GET_REQUESTS(workloads, 'memory')
    total_cpu_request ← GET_REQUESTS(workloads, 'cpu')
    return (total_memory_request, total_cpu_request)

possible_node_configurations ← GET_NODE_CONFIGS_FROM_VENDOR
virtual_nodes ← empty dictionary

for all configuration in possible_node_configurations do
    allocatable_memory ← configuration.allocatable_memory
    allocatable_cpu ← configuration.allocatable_cpu

    resource_requests ← GET_WL_RESOURCE_REQUESTS(workloads)

    cpu_nodes ← ⌈resource_requests.total_cpu_request/allocatable_cpu⌉
    memory_nodes ← ⌈resource_requests.total_memory_request/allocatable_memory⌉
    required_node_count ← max(cpu_nodes, memory_nodes)

    virtual_nodes[configuration.name] ← required_node_count

return virtual_nodes
```

As presented in section 2.1.1, Kubernetes can work with node pools containing a specific machine type. This is especially interesting with large cloud providers, as many different machine configurations exist. Custom machine types that allow selecting the amount of memory and CPU resources make it even harder for the Kubernetes administrator to choose the best setup for their workload.

Algorithm 6 presents the approach to calculate and add virtual nodes to the placement optimizer. First, it is required to fetch the node configurations offered by the vendor via the cloud controller as shown in line 6. The goal is to enable the placement optimizer to be able to move all existing workloads to a single virtual node configuration. Hence, the algorithm must calculate the number of virtual nodes of a particular configuration required to fit all currently running workloads. In this case, working directly with the available compute resources promoted by a cloud vendor for a node configuration is impossible. Each vendor reserves a specific amount of resources, e.g., running the kubelet on the node. For instance, in the case of GKE, 6% of the first CPU core and 25% of the first 4GB of memory must be reserved.[38] Computing these available resources is done automatically by Kubernetes (K8); for virtual nodes, this has to be done before running the rescheduler as seen in lines 10 and 11.

Next, the sum of memory and CPU requests is divided by the respective allocatable resource to get the required number of nodes to fit all resources. The higher number is chosen, and the virtual node objects are added to the virtual representation of the cluster. In this case, the placement optimizer considers these virtual nodes as feasible compute resources to reschedule pods to.

MIP formalization

Using the information provided by the cluster observer and adding the virtual nodes, the optimizer can try to find the best solution using the following formalized MIP.

Equation 5.1 represents the objective of the MIP, which aims to minimize the sum of cost per hour for all nodes that are being used. An additional penalty is introduced for choosing large nodes over small nodes. This enables elasticity in the cluster in rescheduling. Hence the optimizer prefers to choose two 8-core CPU machines over a 16-core machine. Speed is one advantage here, as it is faster to remove a node if it is no longer needed than replacing the 16-core machine with an 8-core machine.

To ensure the optimizer follows general Kubernetes scheduling rules, multiple constraints are introduced. Constraint 5.2 ensures that a workload is only assigned to one particular node. To ensure that all workloads fit onto a node, constraints 5.4 and 5.3 are added. These constraints also include the resource requests of daemon sets. A daemon set ensures that specific workloads run on every node. An example of a daemon set is the Prometheus node exporter that allows Prometheus to scrape current node metrics. The Kubernetes limit of 110 pods per node is enforced through constraint 5.5.

The last constraint 5.6 is not Kubernetes related but ensures the functionality of the MIP. Here, the Big-M method is utilized, introducing a variable M that is significantly large. In this specific case, it has to be at least > 110 as no more than 110 workloads can run on a node as previously presented. If at least one workload is assigned to a node, the decision variable n_i becomes one, which means that the node is running and the cost of the node is considered in the objective function.

Objective function:

$$\text{Minimize} \quad \sum_{i=1}^{|n|} c_i \cdot n_i + 1 \cdot 10^{-6} \cdot (Mem_i + CPU_i) \tag{5.1}$$

subject to:

$$\sum_{i=1}^{|n|} a_{w,n} = 1 \quad \forall w \in W \tag{5.2}$$

$$\sum_{i=1}^{|w|} a_{i,n} \cdot mreq_i + \sum_{j=1}^{|d|} mreqdaemon_j \leq Mem_n \quad \forall n \in N \tag{5.3}$$

$$\sum_{i=1}^{|w|} a_{i,n} \cdot cpureq_i + \sum_{j=1}^{|d|} cpureqdaemon_j \leq CPU_n \quad \forall n \in N \tag{5.4}$$

$$\sum_{i=1}^{|w|} a_{i,n} \leq 110 \quad \forall n \in N \tag{5.5}$$

$$\sum_{i=1}^{|w|} a_{i,n} \leq M \cdot n_i \quad \forall n \in N \tag{5.6}$$

$$\begin{aligned} a_{w,i}, n_i &\in 0, 1 \\ c_i &\in \mathbb{R} \\ mreq_w, mreqdaemon_j &\in \mathbb{R} \\ cpureq_w, cpureqdaemon_j &\in \mathbb{R} \end{aligned}$$

where:

c_i :	Cost per hour in USD for the current node type
n_i :	Binary node usage variable indicating if the node is used
$a_{w,i}$:	Binary assignment variable indicating if the workload w is running on node i
$mreq_w$:	Memory request of workload w
$mreqdaemon_i$:	Memory request of daemon workload j
$cpureq_w$:	CPU request of workload w
$cpureqdaemon_j$:	CPU request of daemon workload j
Mem_i :	Allocatable memory of node i
CPU_i :	Allocatable CPU of node i

5.3.4 Decision processor

Algorithm 7 Decision processor

```
1:  function EVALUATE_OPTIMIZER_RESULT($objective\_value, initial\_cluster\_state$)
2:      $initial\_objective\_value \leftarrow calculate\_node\_cost(initial\_cluster\_state)$
3:      $improvement\_threshold \leftarrow 0.1 \times initial\_objective\_value$
4:
5:      if $objective\_value \geq initial\_objective\_value - improvement\_threshold$ then
6:          return false
7:
8:      $current\_cluster\_state \leftarrow get\_kubernetes\_cluster\_state()$
9:      $wl\_changes \leftarrow calculate\_wl\_changes(initial\_cluster\_state, current\_cluster\_state)$
10:     $n\_changes \leftarrow calculate\_node\_changes(initial\_cluster\_state, current\_cluster\_state)$

11:
12:     if $wl\_changes + n\_changes \geq 0.05$ then
13:         return false
14:
15:     return true
```

The decision processor decides if the calculated optimization is implemented or discarded. Reasons to discard a calculation are either that the optimization value is not high enough or that the cluster has changed too much since the start of the optimization and hence might not be valid anymore.

Currently, only the cluster cost per hour is considered an optimization value returned by the placement optimizer. In algorithm 7 line 5, this optimized objective value is compared against the current price per hour for the running cluster. To be implemented, The optimization must be greater than a predefined threshold, currently 10%.

To ensure this optimization is still valid, the total amount of changes that happened in the cluster, such as a new pod appearing or a node being removed, have to be less than 5%. The decision processor calculates these changes in line 9 and 10 by comparing the initial cluster state before the optimization started, with the current cluster state, thus after the optimization. Once this test has passed as well, the optimizer's plan is put into action by the rescheduler.

5.3.5 Rescheduler

The rescheduler's actions are divided into three phases. First, all required new nodes must be booted. Secondly, all nodes are labeled depending on their existence in the optimization plan. Last, workload rescheduling is started by modifying the deployments.

Boot phase

The cluster optimizer's optimal node configuration only consists of a dictionary with machine configuration and the required count. The rescheduler converts this into an actionable boot plan by comparing the current cluster state with all available nodes with the boot plan. Existing nodes that are part of the boot plan are reused. The cloud controller must boot all other nodes that do not exist.

One option to boot these new nodes is to work with the cloud provider's Kubernetes API to adjust the node pools. In the case of Google Cloud, only one node pool can be adjusted at a time until the boot operation is completed. As Kinema aims to benefit from the large variety of possible node configurations, a boot plan can often consist of multiple types. Hence it is beneficial to be able to boot these different node pool configurations in parallel. To realize this, Kinema directly accesses the underlying compute manager and issues boot requests. This has the same effect as issuing the updates via the cloud provider's Kubernetes API with the benefit of booting up multiple node types in parallel.

As this operation is asynchronous, the cloud provider returns an operation id, which the Kinema's cloud controller regularly queries to update the internal boot state. Once all node boot operations have been completed, the rescheduler waits until these booted nodes appear within Kubernetes. To complete the boot phase, all nodes receive a label: Nodes that are part of the optimizers boot plan are labeled with a Kinema-optimized label, whereas all other nodes receive an expiry label to indicate that these nodes will eventually be removed.

Rescheduling workloads

A simple rescheduling technique would include looping over all pods running on nodes with an expiry label. As each pod has a property pointing to the node name it currently runs on, one could change this to a node name with an optimized label. The pod would be evicted and directly scheduled onto the desired node. While this may seem straightforward, various dependencies and complexities make it highly impractical.

One reason for this is resource allocation on the nodes in Kubernetes. The node currently runs the workload and has reserved memory and CPU resources, whereas the new node might not have these resources. As already discovered in section 5.1.2, the rescheduled pod would not be permitted to boot on this new node. The current placement optimizer currently mitigates this problem as constraints 5.3 and 5.4 exist that prevent running additional workloads on the nodes when all resources have already been allocated. Hence the optimizer's output plan will not include rescheduling tasks that cause resource overcommitment on a node.

Ensuring SLO by providing enough compute resources is another challenge. While updating, the pod becomes unavailable for a particular time as it is evicted from the current node and restarted on the new, desired node. Kubernetes checks the readiness of the pod on the new node using several checks that can include calling an API. Until this readiness check is not passed, no traffic is directed to the pod. Depending on the number of pods that run on the node that is drained by updating the fields, the SLO of service could be violated as not enough resources are available to, e.g., serve traffic.

Given this complexity, Kubernetes already provides a mechanism to perform pod updates while maintaining stability through rolling updates within deployments, as presented in section 2.1.1. While updating the pod template, it's possible to adjust the likelihood of the pods running on specific nodes by utilizing node affinities. These affinities allow to influence scheduling decisions for pods. In this way, a pod can express a preference or a requirement to be placed on a node with specific attributes. This is commonly done if a workload has a specific requirement, e.g., to only run on nodes with SSD volumes attached.

In this case, the rolling update would recreate a pod but with an affinity towards running on nodes that Kinema has preselected to be an optimal future state. The affinities can be categorized into two types:

- RequiredDuringSchedulingIgnoredDuringExecution: This type of affinity defines a requirement that a node must fulfill to be eligible to run the pod
- PreferredDuringSchedulingIgnoredDuringExecution: Nodes that match these criteria are preferred to run the pod. If no node can fulfill this preference, the pod can still be scheduled onto a normal node.

Requiring all pods to run on workloads with a Kinema-optimized label is a feasible solution with the downside of influencing the scalability of the workload's deployments. A potential spike in traffic could cause the HPA to spin up new pods. The CA will, in turn, boot new nodes if there is not enough capacity available on the currently running nodes. The newly scheduled pods, however, cannot run on these new nodes as the required affinity is not met. Here it is beneficial that Kinema also labels all nodes that are not part of the optimal cluster configuration with a different label, an *expiry label*[2]. Hence instead of requiring pods to run on optimal nodes, it should be required **not** to run on nodes with an expiry label. Additionally, the pods should prefer to run on

[2]The name expiry is explicitly chosen as these nodes will eventually be removed from the cluster by Kinema once all workloads are running on other nodes.

nodes with a Kinema-optimized label, allowing them to also run on newly booted nodes without a Kinema label at all.

5.4 Autoscaling

Kinema and the CA are incompatible in the current version due to the nature of booting up new nodes and moving workloads while rescheduling. The default CA analyzes the scheduling pipeline and looks for pods that cannot be scheduled onto the current node configuration. For these pods, an ideal node size is calculated and booted. Workloads that Kinema reschedules appear in this scheduling pipeline. Due to the previously presented affinities, these workloads cannot be placed onto the running nodes selected for removal by Kinema. The CA starts to analyze this scheduling pipeline, not considering the nodes that Kinema booted, and creates an additional, unnecessary node. Due to this fact, Kinema and the default CA are currently conflicting and not used together in the experiments in chapter 6.

Hence, apart from rescheduling, Kinema must also autoscale the node pools using existing capabilities. To do so, the cluster observer tracks all workloads that are currently unschedulable due to missing compute resources. If this is the case, an autoscaling cycle is started instead of rescheduling. The unscheduled workloads, alongside virtual nodes, are passed to the placement optimizer to find an ideal node configuration to run. The optimizer's objective is adjusted, as seen in equation 5.7. It does not contain a penalty for choosing large nodes, as this does not yield any additional benefit in scaling up. The objective's simplicity is sufficient for the purpose of autoscaling because it focuses solely on cost efficiency, eliminating the need for sophisticated optimization techniques that may introduce unnecessary complexity or computational overhead.

$$\text{Minimize} \quad \sum_{i=1}^{|n|} c_i \cdot n_i \tag{5.7}$$

Once the optimizer has found a node configuration, the cloud controller boots up these nodes and the autoscaling iteration ends when the nodes are available in the cluster. In this case, the cluster observer can start a new iteration and check the scheduling cycle for currently unschedulable nodes. All pods already considered in the last autoscaling iteration are filtered out. The default rescheduling cycle continues if the cluster contains no pending pods.

5.5 Limitations and potential optimizatizations

The previously mentioned incompatibility with the default CA is one limitation of the current version of Kinema.

Moreover, as the current rescheduling process heavily depends on rolling updates, the usage of Kinema is restricted to clusters that manage workloads using RS. The rolling updates are an additional limitation. For a rescheduling operation, currently, all pods have to be updated. For instance, a cluster has a redundant node that should be removed. Ideally, Kinema can update the affinities on the pods on this node. This is impractical, as presented in section 5.3.5. Hence, Kinema has to update the deployments responsible for these workloads on the node. An update to the deployment automatically causes a new creation of a new RS, which causes a rolling update to all pods running on all nodes. Further future investigation and experimentation are necessary to find approaches that enable an equally stable rescheduling process that only adjusts workloads on nodes to be evicted. Here, so-called *in place updates* that come with Kubernetes version 1.27 are an interesting starting point that allows adjusting pod attributes, such as resource requirements, without restarting. Currently, the highest available version on common cloud vendors is 1.24-1.26 [39, 40, 41]. Combining in-place updates with optimizing the currently simple approach to forecasting future resource usage can be a compelling addition to ensuring more efficient cluster usage by improving node utilization.

Another limitation is the type of application that is feasible for rescheduling in Kinema. Currently, only stateless applications can be rescheduled. Stateful applications have data or a state tied to a specific node. Rescheduling these applications can lead to data inconsistencies and loss of access to the storage. This type of application requires careful planning and data synchronization to ensure a smooth rescheduling process and is therefore not considered in Kinema. The best principles for microservice applications require these to be stateless anyways, though [42].

6 Experiments

6.1 Load Test experiment

The following experiment aims to analyze the behavior of Kinema with different workload patterns and two different sample applications. Before presenting the experiment result, the experimental architecture and sample applications are described.

6.1.1 Setup

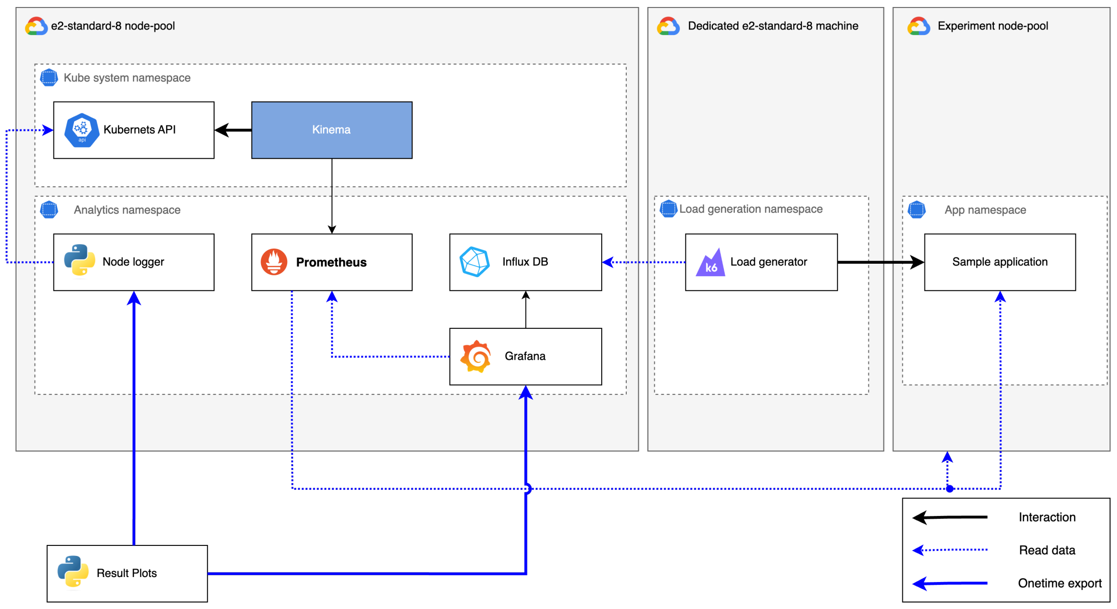

Figure 6.1: Kinema experiment setup

Figure 6.1 shows the experimental architecture that is used to evaluate Kinema. The cluster runs on GKE version 1.24.11. To ensure that the individual components do not influence one another, the deployments are separated in different namespaces and run on different node pools. The kube-system and analytics namespace run on Google Cloud e2-standard-8 virtual machines with 8vCPUs and 32 GiB of memory. A

Machine type	vCPUs	Memory (GiB)	Cost per Hour
e2-standard-2	2	8	$0.09
e2-standard-4	4	16	$0.17
e2-standard-8	8	32	$0.35
e2-standard-16	16	64	$0.69
e2-standard-32	32	128	$1.38
e2-highcpu-2	2	2	$0.06
e2-highcpu-4	4	4	$0.12
e2-highcpu-8	8	8	$0.25
e2-highcpu-16	16	16	$0.51
e2-highcpu-32	32	32	$1.01

Table 6.1: Selected Node pools available to the Kubernetes cluster in europe-west3 (Frankfurt)

dedicated machine of the same type is provisioned for the load testing tool, Grafana k6[1], which generates the traffic patterns that run against the sample application.

A CA scales the experimental node-pools to provide compute power to the experiment applications when Kinema is not used. Kinema does not need a CA as it can independently access and scale the node pools. As mentioned in section 5.4, kinema and the CA are not yet compatible. Multiple node pools with different configurations can be accessed, as shown in Table 6.1. To ensure that only the deployed sample applications run on these nodes, so-called taints are used. Taints can be used to control which pods run on which nodes. In practice, this is a key-value pair with an associated effect such as NoSchedule. If a specific pod can run on this node, toleration has to be added to the pod's deployment file, specifying the key-value pair and the effect.

One must also ensure the sample application's pods do not run on the e2-standard-8 machines that are reserved for the kube-system and analytics namespace. This can be done through a node selector in the deployment file. A node selector can only select one node pool, but this is not a sufficient option as multiple node pools exist. Node affinity is the solution which can also be specified in the deployment file. This allows to match a list of values to a node's label. This is ideal, as one can match the node pool label with all node pool names as listed in 6.1.

Once a load test is started, k6 logs the experiment results to an influx DB running in the analytics namespace. Prometheus collects utilization values of the pods and underlying nodes. The data is collected and combined within Grafana for final export. A separate Python docker container also collects information about the current node

[1]https://k6.io/

configurations and the status of the HPA. All data from Grafana and the Python container is exported and visualized on a local machine using Matplotlib.

Beauty-shop sample application

Figure 6.2 depicts the architecture of the first sample application, a boutique online store comprising 11 microservices written in different programming languages that use google Remote Procedure Call (gRPC) for inter-service communication. The experiments will target the front end's user-facing HTTP API.

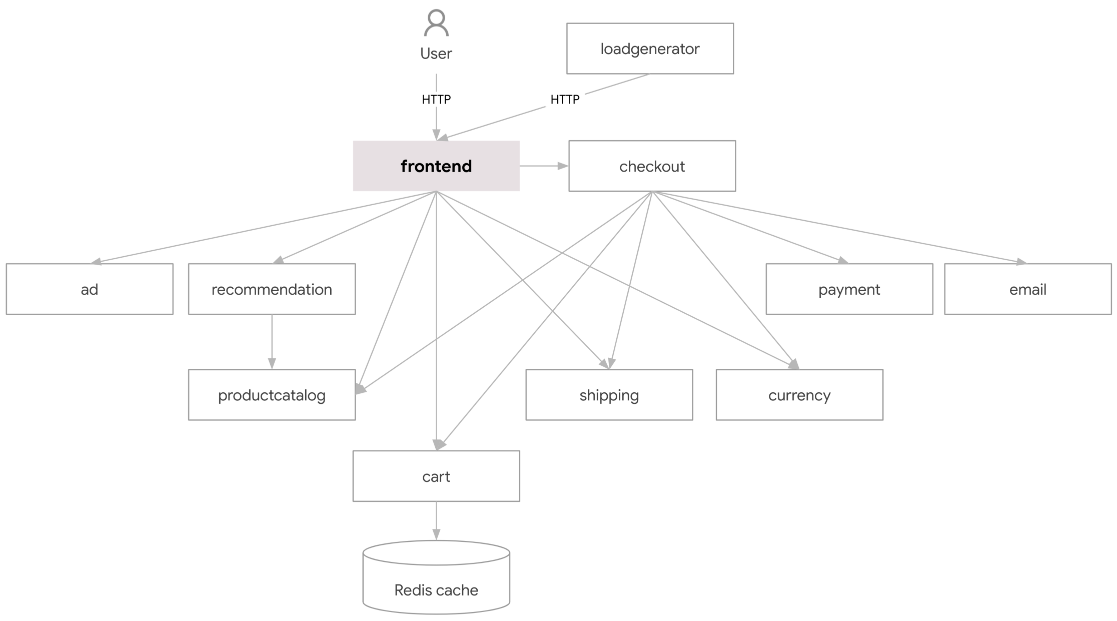

Figure 6.2: Beautyshop sample application architecture

Tea-store sample application

The second sample microservice architecture, as shown in Figure 6.3, is a tea-online store consisting of five microservices and a service registry written in Java [9]. The inter-service communication is established via Representational State Transfer (REST) protocol. The resource requests established in section 2.1.2 are being used for the experiments, as the public repository contains no resource suggestions.

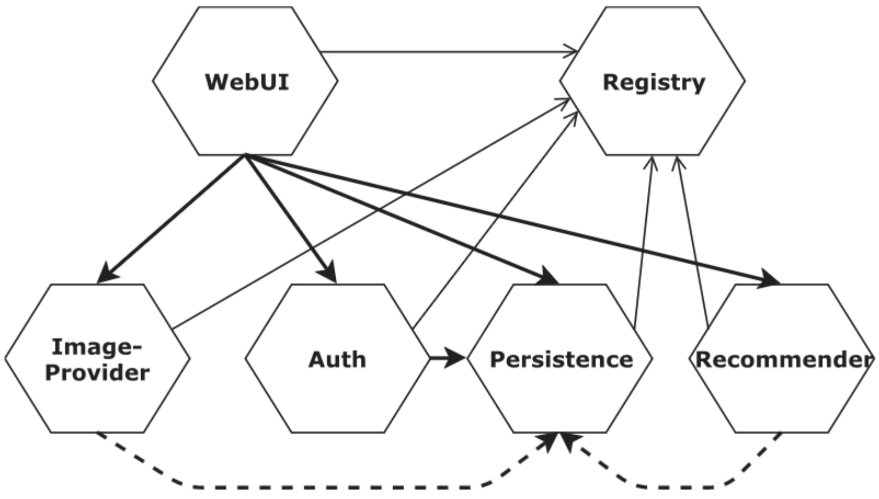

Figure 6.3: Teastore sample architecture[9]

Load Test configuration

The load tests are written in a javascript file that the k6 load runner inside a docker container executes. So-called VUs are utilized to simulate traffic, where a single VU is a simple while loop executing the same API requests each time. Hence, the RPS that each test can reach depends on the throughput of the deployed application.

In the following, three different experiments are presented that differ in the load pattern employed. All three load patterns are executed for both sample applications. Both applications directly compare Kinema with the optimize utilization CA. The load patterns for the beauty shop application also compare to the default CA.

6.1.2 Linear ramp up

A linear VU ramp-up load pattern experiment is performed in the following. Within ten minutes, the number of VUs scales from 0 to 1500 and back down to zero again in ten minutes.

The results for the beauty shop experiments are visible in Figure 6.4 and Table 6.2. During the load experiment, the optimize utilization autoscaler achieved the highest number of RPS while achieving a similar average cost per hour for the infrastructure as Kinema and the default CA. The latter reached the highest minimum prices for the cluster, as the initial minimum number of nodes differed from Kinema and the optimized utilization CA. Before starting the experiments, all autoscaling systems

	During load test					**Post load test**			
Name	avg. RPS	avg. CPU	avg. memory	avg. cost	min. cost	avg. CPU	avg. memory	avg. cost	min. cost
Teastore - linear ramp up									
Baseline	1165	38%	74%	0.69	0.38	8%	62%	0.63	0.43
Kinema	2832.00	31%	84%	0.93	0.32	19%	58%	0.73	0.32
Beautyshop - linear ramp up									
Default	341	45%	84%	0.36	0.25	8%	49%	0.43	0.13
Baseline (opt. util)	614	61%	73%	0.38	0.13	13%	60%	0.16	0.13
Kinema	455.6	59%	79%	0.38	0.13	17%	69%	0.35	**0.09**

Table 6.2: Linear ramp-up results for sample applications

booted the necessary node count based on their internal logic.

Once the load test was over and the post-workload phase commenced with most pods idle, Kinema achieved the highest CPU and memory utilization while also reaching the lowest cluster cost per hour. While both the default and optimize-utilization CA stayed with a final node configuration, Kinema continued to optimize until the placement optimizer could no longer find a better node configuration. In this phase, the average CPU utilization has improved by 31% while the memory utilization has a relative improvement of 15%. The average cost is temporarily higher, as rescheduling operations require booting new nodes for a short period of time until a final cost improvement of 31% is reached.

The cost optimization is also visible in the Teastore sample application. Here, the minimum price of the baseline optimizer could be improved by 25.6% during the post-load test phase. In this phase, the average CPU utilization is 11% higher in the deployment with Kinema, which means an improvement of 138%, whereas the memory utilization had a relative decrease of 7%. Figure 6.5 presents the findings in detail.

When deployed with Kinema, the RPS linear ramp up and down is consistent except for minor latency at around 4000 RPS. This is not the case for the baseline application. A drop in RPS is visible in Figure 6.5 at ❶. This is caused by removing a node at ❷. The CA removed a node because the workload could be scheduled onto another node. Here, the rescheduling process significantly differs from Kinema. While the latter aims to ensure that there is no downtime by utilizing rolling updates, the CA works with pod eviction. The pods on the node are deleted and automatically recreated by the controller. This causes the application to be unavailable for a short period, leading to the reduced number of RPS. The same observation can be made at ❸ where a substantial drop in RPS is visible caused by a similar node removal operation in ❹. Shortly after, multiple nodes are readded. The RPS count does not recover to a comparable level of Kinema, though.

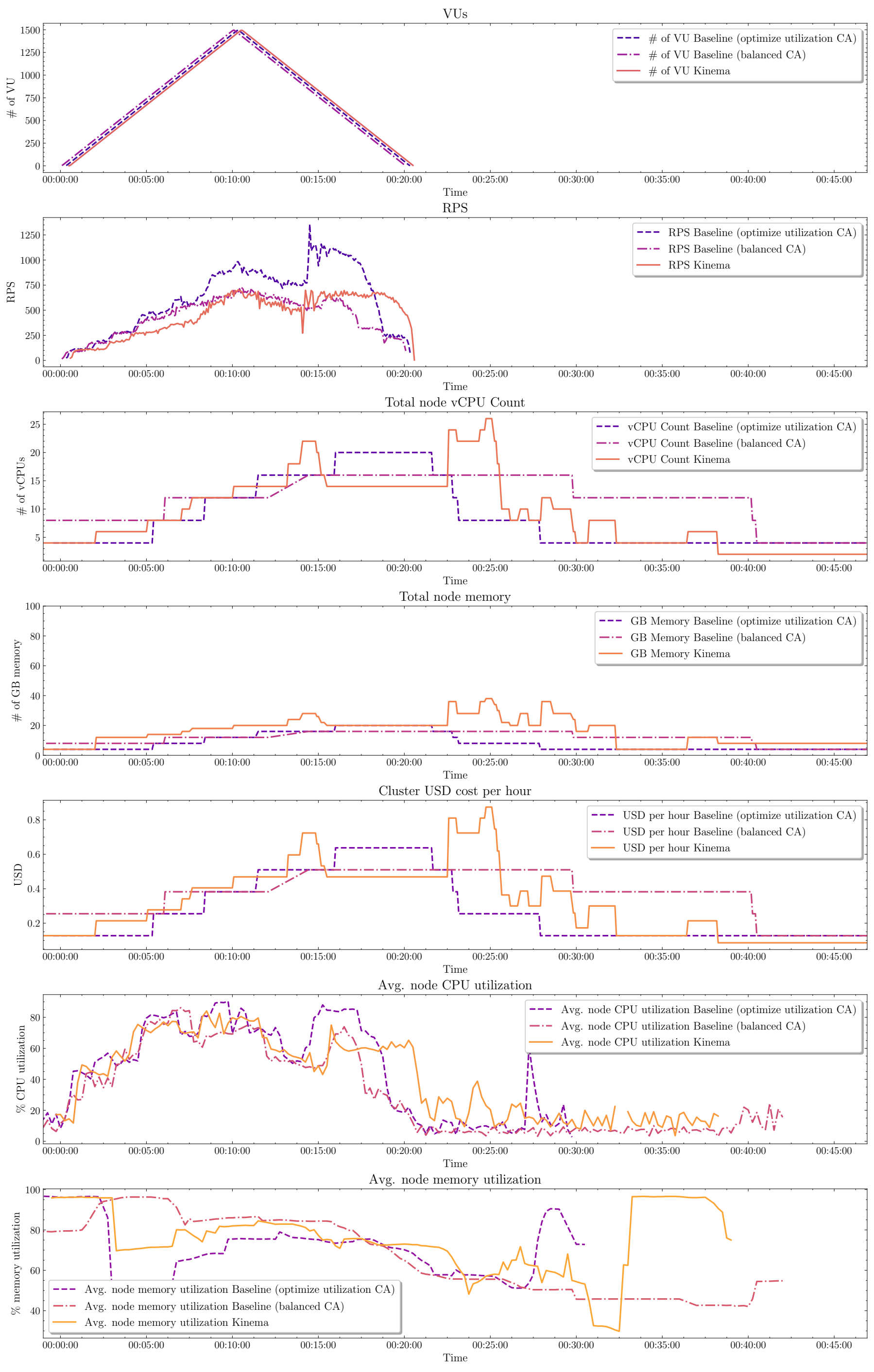

Figure 6.4: Beautyshop Linear ramp up

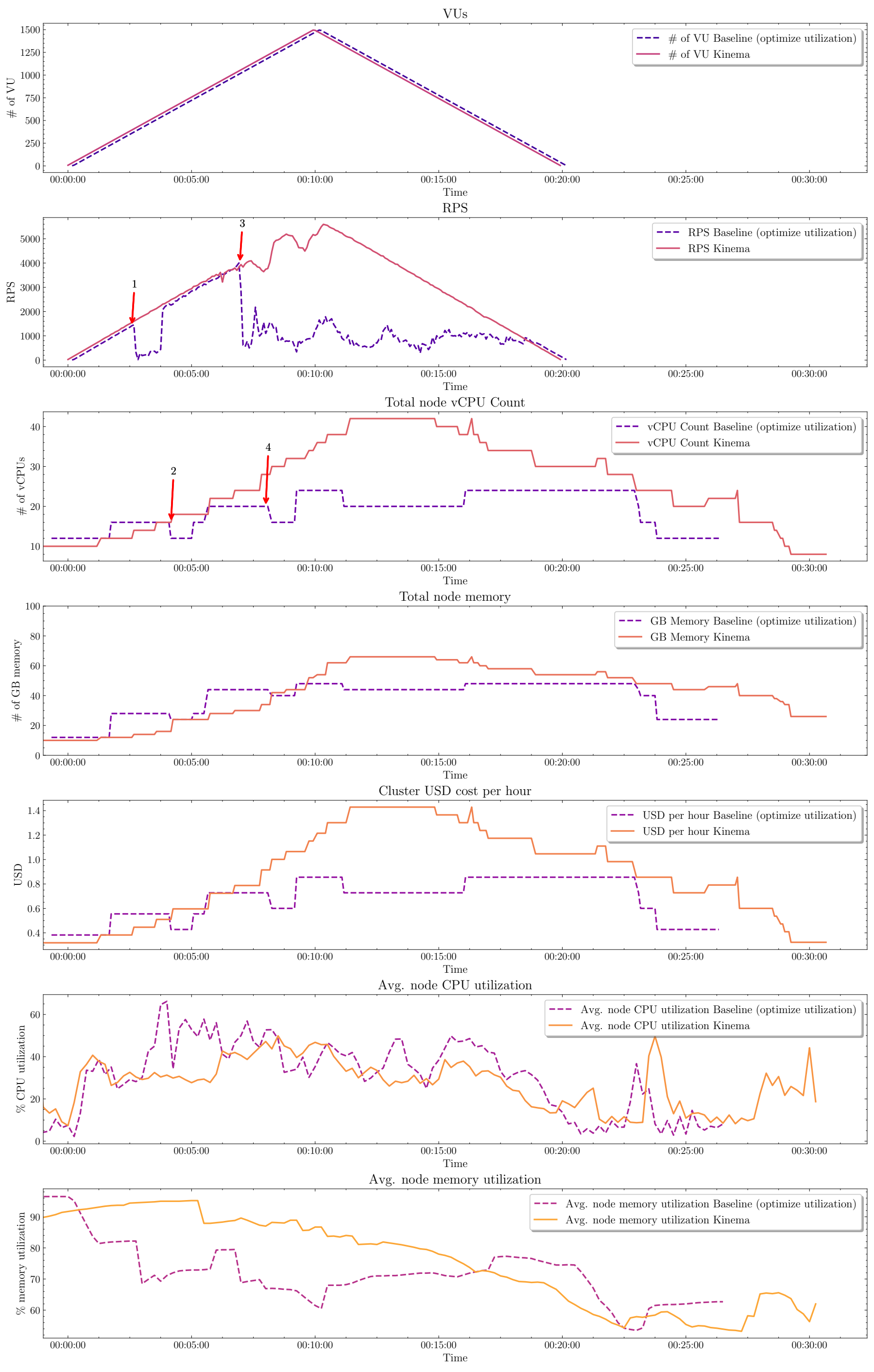

Figure 6.5: Teastore linear ramp-up experiment

	During load test					Post load test			
Name	avg. RPS	avg. CPU	avg. memory	avg. cost	min. cost	avg. CPU	avg. memory	avg. cost	min. cost
Teastore - spike									
Baseline	623.7	21%	86%	0.44	0.38	6.4%	73%	0.38	0.38
Kinema	596.1	26%	78%	0.86	0.38	14%	51%	0.27	0.17
Beautyshop - spike									
Default	130	27%	50%	0.29	0.13	10%	38%	0.29	0.13
Baseline (opt. util)	185.1	39%	85%	0.24	0.13	11%	56%	0.17	0.13
Kinema	210	47%	79%	0.28	0.13	14%	77%	0.24	**0.09**

Table 6.3: Spike load pattern results for tea store and beauty shop experiments

6.1.3 Spike workload

In the second load pattern, the number of VUs ramps to 100 and stays consistent at this value for six minutes. After that, the number of VUs increases to 500. This number stays consistent for five minutes until it takes 30 seconds to ramp back to 50 VUs. This number of VUs is held for nine additional minutes.

Regarding the beauty shop application, Kinema performed similarly to the optimize utilization autoscaler during the workload experiment, as depicted in Figure 6.6. The default baseline autoscaler showed lower utilization values and achieved a lower RPS score while having higher costs. In the post-load phase, Kinema achieved the highest CPU and memory utilization values while optimizing for the lowest minimum cost as presented in Table 6.3. The CPU utilization shows a relative improvement of 27% and the memory utilization of 38% when compared with the optimize utilization CA in the post-load phase. The costs were improved by 31%.

These findings are confirmed by the experiments using the Teastore application in Figure 6.7. Here, Kinema reaches similar RPS scores. A reschedule operation causes a temporary latency increase during the spike phase, which is caused by the cold-start latency of the application. Moreover, while both the baseline and Kinema reach average RPS values above 600, the costs for Kinema are higher during the load phase. Here, the autoscaler added more resources than the optimized utilization CA. However, Kinema outperformed the CA by reaching a lower minimum cost with a relative improvement of 56% and higher average CPU values with a relative improvement of 119%.

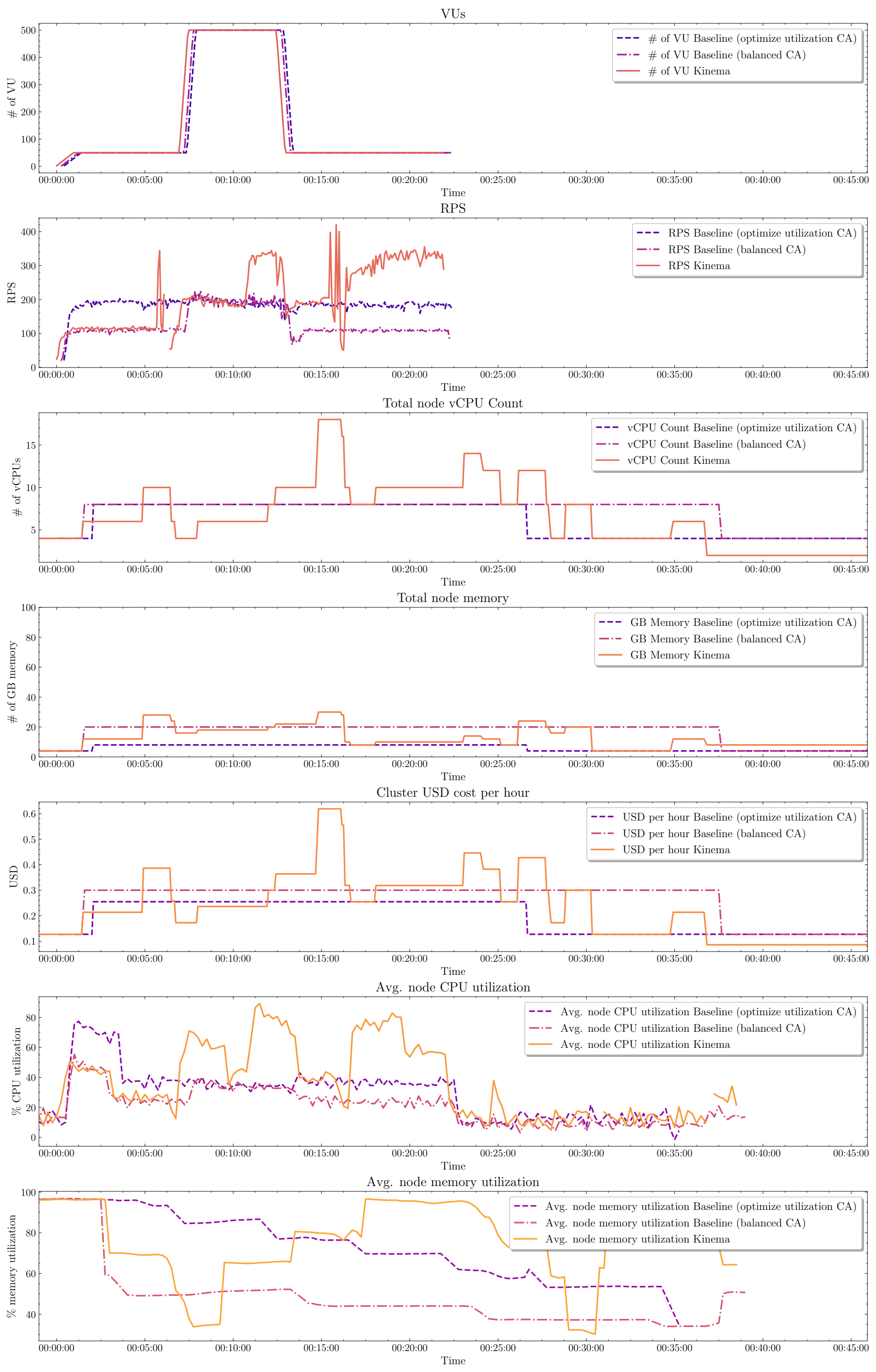

Figure 6.6: Beautyshop spike load pattern

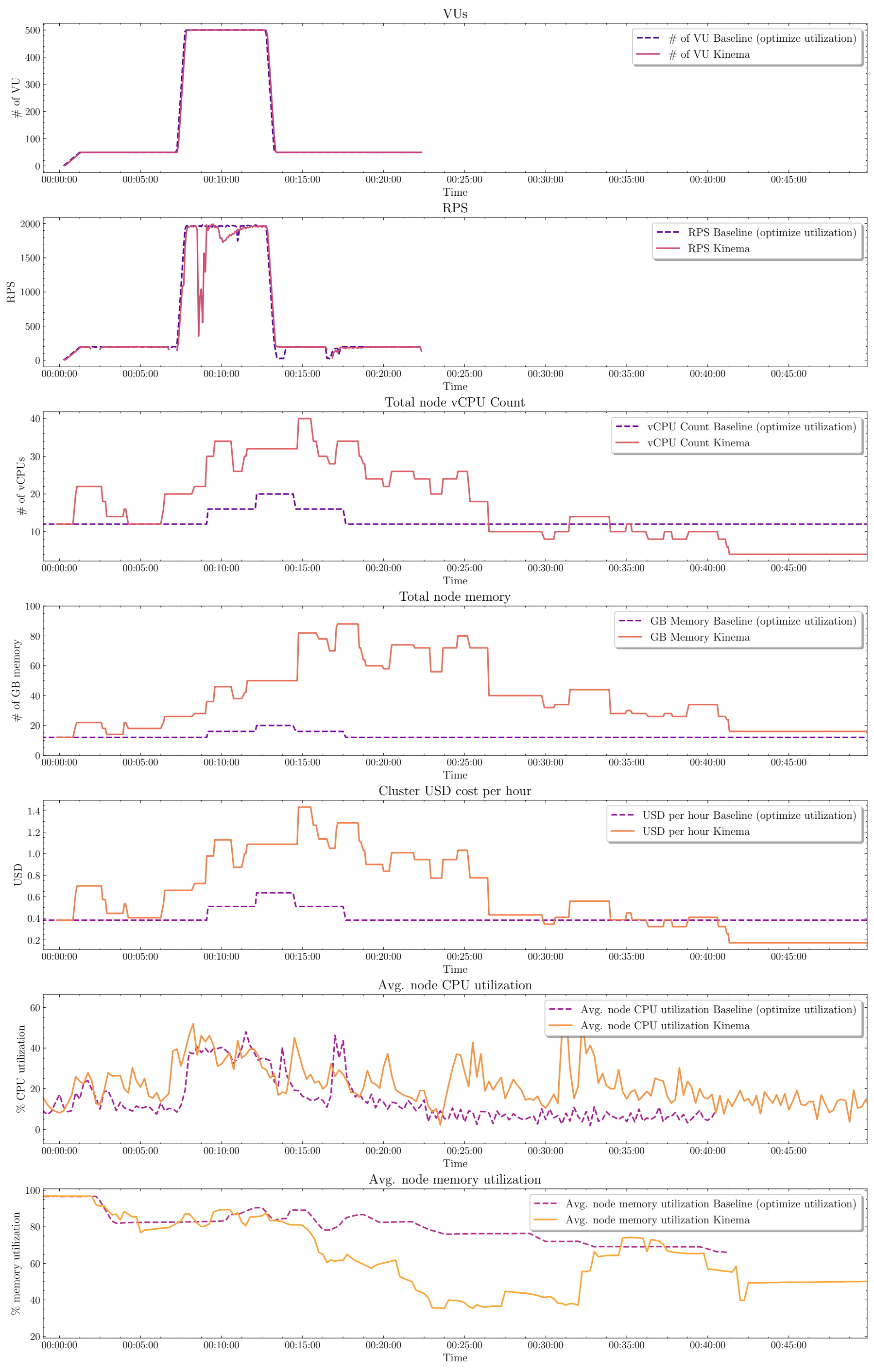

Figure 6.7: Teastore spike load pattern

	During load test					Post load test			
Name	avg. RPS	avg. CPU	avg. memory	avg. cost	min. cost	avg. CPU	avg. memory	avg. cost	min. cost
Teastore - random									
Baseline	1543	35%	80%	0.59	0.38	7%	67%	0.54	0.43
Kinema	1524	30%	67%	0.85	0.38	23.5%	49%	0.55	0.17
Beautyshop - random									
Default	323.7	51%	76.3%	0.33	0.13	7%	57%	0.53	0.25
Baseline (opt. util)	595	58%	88%	0.39	0.13	11%	58%	0.27	0.13
Kinema	530	67%	70%	0.50	0.13	25%	57%	0.36	0.09

Table 6.4: Experiment results for random load pattern

6.1.4 Random workload

The last load pattern is a random workload pattern. In this experiment, the number of VUs increases from zero to 300 linearly within the first four minutes. After that, the number decreases again to 50. A linear ramp up to 1000 VUs follows suit within ten minutes. Once this number is reached, a linear scale back to zero takes six minutes.

Regarding the beauty shop application, the highest RPS score was reached by the optimize utilization autoscaler while having lower costs than Kinema, which also reached less RPS. Towards the second spike in the random workload, Kinema autoscaled more resources than any other autoscaler, as presented in Figure 6.8. This over-scaling of Kinema is also visible for the Teastore application with the random load pattern and shall be part of future improvement. The costs for running Kinema are higher while achieving the same average RPS values.

The core capabilities of Kinema lie in the optimization when the cluster is in an idle state, hence while being in the post-load phase. In this phase, Kinema reaches a higher average CPU utilization in both experiments. The relative improvement for the beautyshop when compared with the optimize-utilization CA is at 127% CPU utilization improvement while the final costs are relatively reduced by 31%. This improvement is visible in the case of the Teastore application as well, with a CPU utilization improvement of 236% while reaching a relative cost reduction of 60%. The final node configuration that causes this cost optimization remains stable until new requests arrive, causing the HPA to add new pods. A real-world scenario of this could be an application running idle overnight, with higher cost savings by utilizing Kinema, and becoming active again in the morning. The feasibility of still being able to serve traffic after a high degree of cost and resource optimization is demonstrated in Figure

6.9, where the same load pattern is repeated on the Kinema load test to show that the cluster boots again. In this case, a similar RPS performance was reached compared to when running the load test for the first time. However, the previously identified over-scaling pattern of Kinema is visible again, causing a higher cluster cost than in the first iteration.

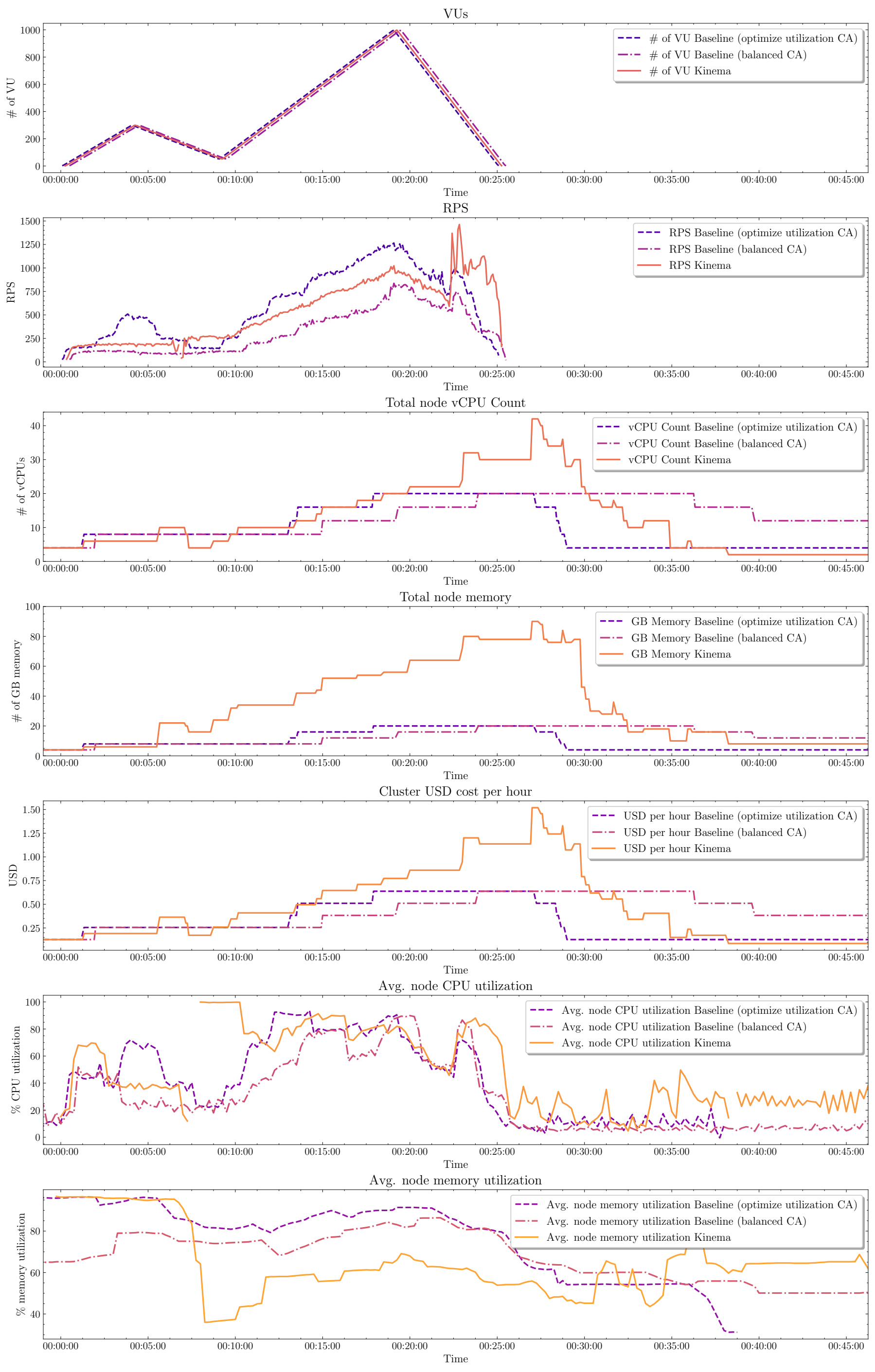

Figure 6.8: Beautyshop random load pattern

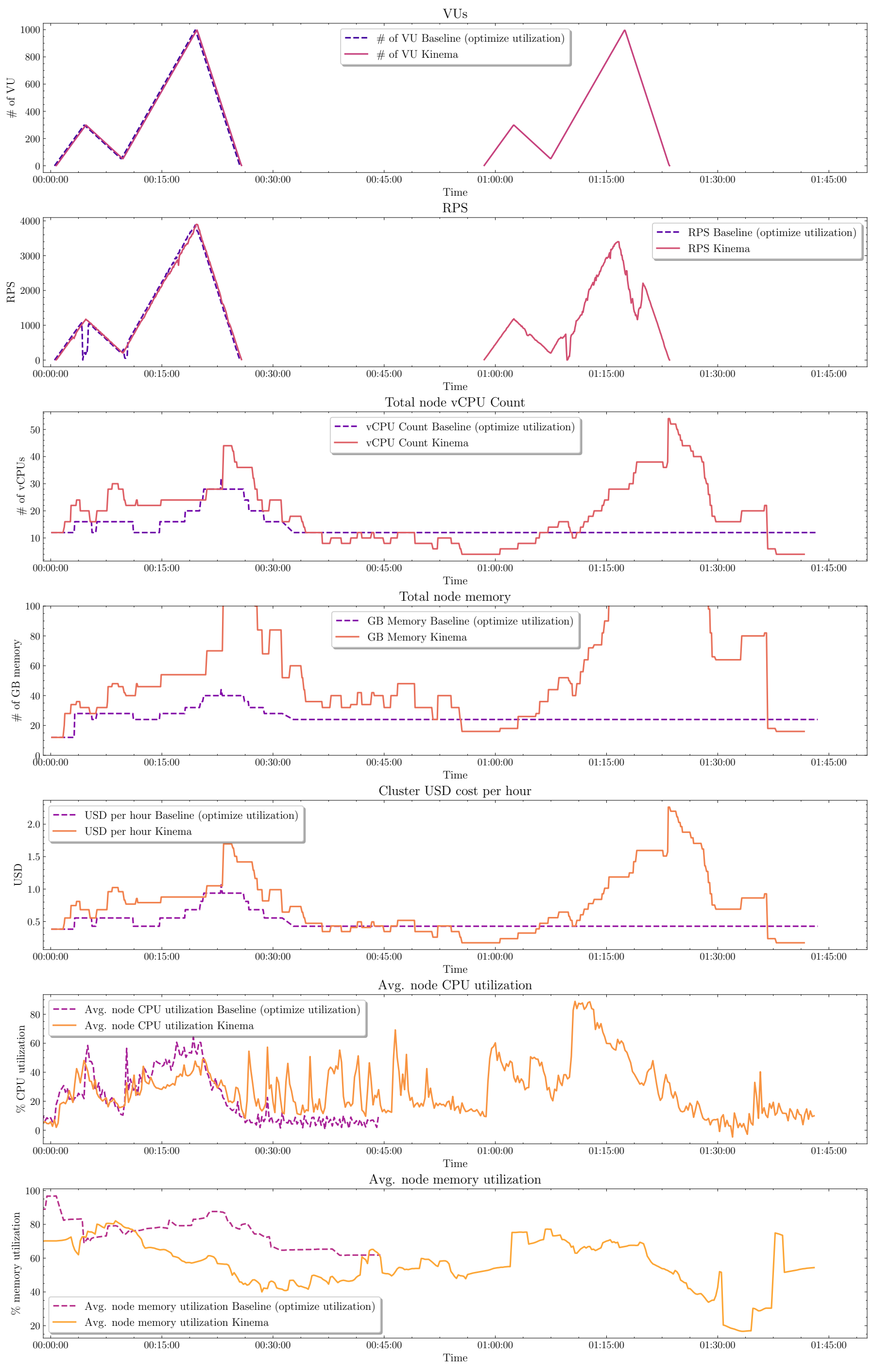

Figure 6.9: Teastore random load pattern

6.1.5 Conclusion

In conclusion, the experiments demonstrate that Kinema performs similarly to the default autoscaler when utilizing an optimized autoscaling profile. The default autoscaler consistently underperformed in terms of average RPS across all the conducted experiments.

It is difficult to confirm the widespread usage of the optimize-utilization autoscaling profile in production, as there are no sources other than Google's documentation available. According to the documentation, the feature has been available since August 2021, following a beta phase. However, it is likely that this profile is not extensively used in production due to certain drawbacks mentioned in section 6.1.2. These drawbacks include the potential for service disruptions and a lack of understanding regarding the scaling behavior, which can be compared to a black box autoscaler [43].

Kinema, on the other hand, is designed with observability in mind to address these concerns and ensure users do not experience a black-box feeling when deploying the system to production. The cluster observer logs the reasons behind scaling iterations, including both the decisions to start or not start an iteration. Similarly, the decision processor logs the decisions made when calculating new placement optimizations. This logging capability empowers users to fine-tune the parameters of Kinema based on their specific needs.

Regarding service disruptions, Kinema follows the standard process of rolling updates to apply reschedule operations. In the event of disruptions or latency spikes, users can optimize the tolerability of their applications towards rolling updates and adjust parameters such as the maximum number of pods that can be unavailable simultaneously.

In summary, while the "optimize-utilization" autoscaling profile lacks extensive information beyond Google's documentation, Kinema addresses concerns of understanding and fine-tuning by providing observability and leveraging rolling updates for mitigating service disruptions.

One of the drawbacks of autoscaling using kinema is the tendency to overscale, resulting in the wastage of resources. However, this issue can be mitigated by integrating Kinema with the CA, leading to more optimized resource allocation and reduced resource wastage in the future.

Moreover, during idle periods, Kinema demonstrated its core capability to optimize costs by up to 60% while maintaining reliable system functionality. This cost optimization further emphasizes the potential benefits of incorporating Kinema into autoscaling practices.

6.2 Kinema scalability

The following experiment tests the scalability and overhead of the Kinema system based on a varying number of available nodes and workloads.

6.2.1 Dataset and setup

The dataset in the experiment is derived from Alibaba cloud traces from 2017, specifically the ClusterData201708 dataset. It provides information about a production cluster over 12 hours [44]. This dataset contains data from approximately 1.3k machines that run online services and batch jobs.

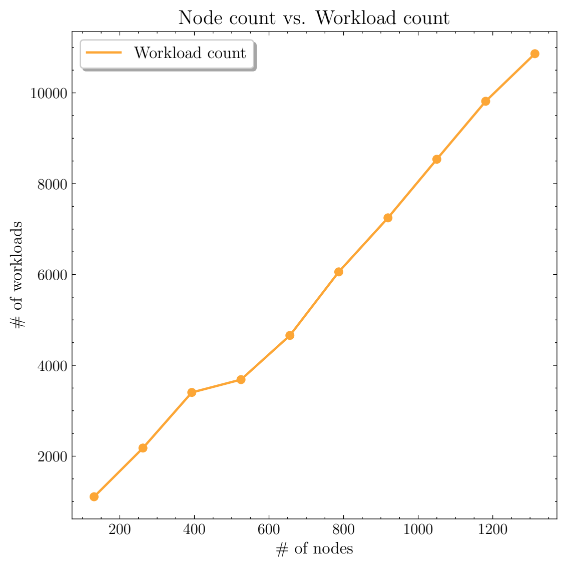

Figure 6.10: Comparison between the number of nodes and the total number of workloads distributed across these nodes based on the Alibaba 2017 cluster dataset.

To conduct the experiment, a virtual Kubernetes cluster is created to simulate the resource allocation system. The workloads and node information obtained from the dataset are utilized to configure the cluster environment accurately. The experiment is performed in an iterative fashion where the total number of nodes is increased by 10% in each iteration. The number of workloads does not scale accordingly, as the workload to node mapping is used from the dataset as depicted in Figure 6.10.

The primary objective of the experiment is to measure the time the Kinema optimizer takes to generate the first feasible solution. Feasibility is determined based on the system's ability to allocate workloads to nodes while adhering to scheduling constraints such as resource limits on nodes.

Before conducting the experiment, the dataset is analyzed to determine the appropriate machine configurations. All machines in the dataset have 64 CPU cores. The memory consumption data also reveals different normalized memory values, indicating varying memory requirements.

Two machine configurations that are available on GKE are chosen as starting points as these match the resource requirements of the Alibaba cloud machines in the dataset. The first configuration, "n2-highcpu-64," features 64 CPU cores and 64GiB of memory. The second configuration, named "n2-highcpu-91," aims to accommodate larger machines and includes approximately 91GiB of memory[2].

6.2.2 Results

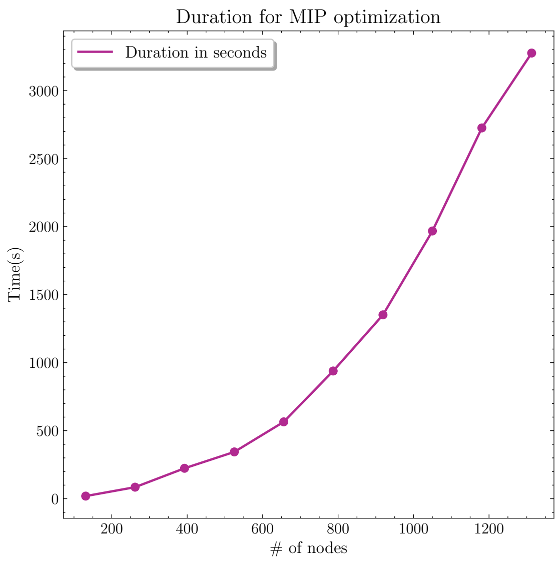

Figure 6.11: Comparison between the number of nodes and the execution time of the MIP to find a feasible solution

From Figure 6.11, it is evident that there is a positive correlation between the number of nodes and the calculation time per node. This indicates that as the number of nodes utilized for the calculations increases, the workload per node becomes more demanding, resulting in longer computation times. The relationship between the

[2]Currently, no machine with 91GiB of memory exists. Custom machine configurations are possible on GKE, though.

percentage of total node count and the average calculation time suggests that the Kinema optimizer's scalability may not be linear. Instead, it demonstrates that the rate at which the calculation time increases per node becomes more pronounced as a more significant proportion of the total nodes are employed. The total time can be reduced if Kinema only considers a certain set of nodes or a certain set of namespaces instead of optimizing the full cluster.

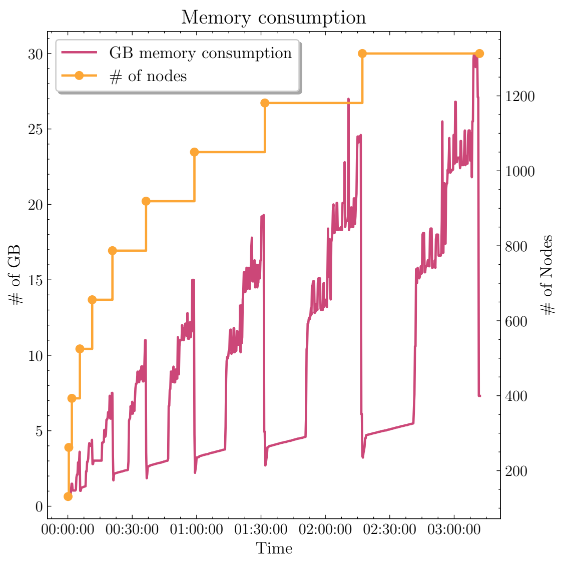

Figure 6.12: Measurement of the memory consumption of the MIP for each iteration with a changing subset of nodes.

Figure 6.12 demonstrates the memory consumption of the MIP with varying numbers of nodes. Memory consumption also positively correlates with the number of nodes and workloads used. Figure 6.11 visualizes the number of workloads distributed across the nodes. As each workload and each node is associated with various constraints and decision variables, it is clear that the memory consumption scales appropriately.

6.2.3 Conclusion

The Kinema optimizer can find feasible solutions for large clusters with more than 1.000 nodes and 10.000 workloads. The feasibility in practice of this solution highly depends on the cluster's rate of change as the optimizer works with a static view on the cluster. If the cluster changes while the optimizers are calculating a solution, the current implementation abandons this solution if the total number of workloads and nodes changes by a preconfigured percentage as explained in section 5.3.4. As demonstrated in the results, the larger the cluster is, the longer the calculation for an optimized solution takes. Hence, Kinema may have difficulty finding a feasible solution when the cluster changes a lot. To overcome this issue, future work can include a distributed optimization approach with multiple instances of Kinema, where each instance is responsible for optimizing a separate namespace or individual node pools.

7 Future work

Several areas warrant future investigation and improvements to enhance Kinema's performance and efficiency.

First, the over-scaling issue of Kinema shall be addressed to ensure running Kinema in production is possible. The system currently has the tendency to add too many small nodes too quickly, causing resource wastage. A potential solution could be to adjust the objective function of the placement optimizer to prefer booting larger nodes that are eventually rescheduled by Kinema if unnecessary. Various experiments will be conducted to test if a comparable performance to the CA can be achieved. If this is not the case, future work should focus on improving the compatibility with the CA. An idea here would be that Kinema only cordons[1] the nodes it wants to deschedule and start a rolling update with affinities right away. The CA will be forced to boot new nodes.

Next, utilization forecasting shall be improved. The resource request adjustments that Kinema can make allow the internal placement optimizer to find even better node configurations. This is also the reason why Kinema was always able to find smaller and cheaper nodes than the default CA. Enhancing the accuracy and reliability of the resource forecasting model is paramount to ensuring the efficient and safe operation of the Kinema placement optimizer. Further research and development are required to improve the existing forecasting algorithms employed in Kinema. Incorporating advanced statistical and machine learning techniques, such as time series analysis or predictive modeling, can enhance the precision of resource forecasting, leading to even better resource allocation decisions.

As presented in section 6.2, the scalability of Kinema greatly depends on the scalability of the internal placement optimizer. The current optimization algorithm utilized by Kinema shows signs of struggling with the curse of dimensionality. To overcome this challenge, alternative optimization techniques should be explored. Research on methods such as genetic algorithms, simulated annealing, or particle swarm optimization could provide valuable insights into improving the optimization process within Kinema. By evaluating and implementing these techniques, it is anticipated that the system's overall performance and efficiency can be significantly enhanced.

[1]When a node is cordoned, the scheduler cannot place any new pods onto it

Lastly, Kinema shall be put into production at various systems to gather valuable production data and real-world cases. This step involves deploying the system in different environments and collecting data on resource usage, autoscaling behavior, and optimization performance. Analyzing this data will provide crucial insights into Kinema's practical applicability, identify areas for further improvement, and validate the effectiveness of the proposed enhancements.

Abbreviations

AWS	Amazon Web Services
CA	Cluster Autoscaler
GKE	Google Kubernetes Engine
GPU	Graphics processing unit
gRPC	google Remote Procedure Call
HPA	Horizontal Pod Autoscaler
HTTP	Hypertext Transfer Protocol
LP	Linear Program
MIP	Mixed Integer Program
K8	Kubernetes
PDB	Pod Disruption Budget
REST	Representational State Transfer
RPS	Requests per second
RS	Replicaset
SLO	Service Level Objective

VPA	Vertical Pod Autoscaler
VU	Virtual user

List of Figures

1.1 Visualization of the utilization of CPU and memory resources of the vystem cluster in one week 2

2.1 Basic Kubernetes Architecture 7
2.2 Metric HPA architecture 9
2.3 Comparison of scaling a sample application solely based on CPU metrics vs. combining CPU and latency metrics 10
2.4 Teashop VPA CPU observations 12
2.5 Teashop VPA memory observations 13

4.1 Trimaran load aware scheduling plugin architecture [29]. 25
4.2 Trimaran scoring based on a target utilization of 50% [29] 26
4.3 Comparison of the Trimaran target load packaging plugin vs. the default scheduler using a balance utilization CA profile 29
4.4 Comparison of the Trimaran target load packaging plugin vs. the default scheduler using a optimize utilization CA profile 30

5.1 Kinema architecture overview 34
5.2 Rescheduling sequence 35

6.1 Kinema experiment setup 50
6.2 Beautyshop sample application architecture 52
6.3 Teastore sample architecture[9] 53
6.4 Beautyshop Linear ramp up 56
6.5 Teastore linear ramp-up experiment 57
6.6 Beautyshop spike load pattern 59
6.7 Teastore spike load pattern 60
6.8 Beautyshop random load pattern 63
6.9 Teastore random load pattern 64
6.10 Comparison between the number of nodes and the total number of workloads distributed across these nodes based on the Alibaba 2017 cluster dataset. 66

6.11 Comparison between the number of nodes and the execution time of the MIP to find a feasible solution 67
6.12 Measurement of the memory consumption of the MIP for each iteration with a changing subset of nodes. 68

List of Tables

2.1 VPA recommendations for teastore application 14

3.1 Approach comparison to optimize cluster utilization and reduce costs . 21

4.1 Experimental comparison of the balance and optimize utilization autoscaling profiles with the default and load-aware scheduler 27
4.2 Statistical hypothesis test of a one-sided t-test to evaluate the significance of trimaran target load packaging on the CPU utilization levels 27

6.1 Selected Node pools available to the Kubernetes cluster in europe-west3 (Frankfurt) 51
6.2 Linear ramp-up results for sample applications 54
6.3 Spike load pattern results for tea store and beauty shop experiments . . 58
6.4 Experiment results for random load pattern 61

Bibliography

[1] B. Varghese and R. Buyya. *Next Generation Cloud Computing: New Trends and Research Directions*. Sept. 8, 2017. DOI: 10.48550/arXiv.1707.07452. arXiv: 1707.07452[cs]. URL: http://arxiv.org/abs/1707.07452 (visited on 11/27/2022).

[2] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes. "Borg, Omega, and Kubernetes." In: *Communications of the ACM* 59.5 (Apr. 26, 2016), pp. 50–57. ISSN: 0001-0782, 1557-7317. DOI: 10.1145/2890784. URL: https://dl.acm.org/doi/10.1145/2890784 (visited on 11/28/2022).

[3] C. Carrión. "Kubernetes Scheduling: Taxonomy, ongoing issues and challenges." In: *ACM Computing Surveys* (June 2, 2022), p. 3539606. ISSN: 0360-0300, 1557-7341. DOI: 10.1145/3539606. URL: https://dl.acm.org/doi/10.1145/3539606 (visited on 11/16/2022).

[4] *Why Large Organizations Trust Kubernetes*. URL: https://tanzu.vmware.com/content/blog/why-large-organizations-trust-kubernetes (visited on 11/28/2022).

[5] L. M. Ruíz, P. P. Pueyo, J. Mateo-Fornés, J. V. Mayoral, and F. S. Tehàs. "Autoscaling Pods on an On-Premise Kubernetes Infrastructure QoS-Aware." In: *IEEE Access* 10 (2022). Conference Name: IEEE Access, pp. 33083–33094. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2022.3158743.

[6] 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)*. 2019 IEEE 25th International Conference on Parallel and Distributed Systems (ICPADS). Tianjin, China: IEEE, Dec. 2019, pp. 193–200. ISBN: 978-1-72812-583-1. DOI: 10.1109/ICPADS47876.2019.00037. URL: https://ieeexplore.ieee.org/document/8975761/ (visited on 11/16/2022).

[7] *Considerations for large clusters*. Kubernetes. Section: docs. URL: https://kubernetes.io/docs/setup/best-practices/cluster-large/ (visited on 06/10/2023).

[8] *autoscaler/vertical-pod-autoscaler at master · kubernetes/autoscaler*. GitHub. URL: https://github.com/kubernetes/autoscaler (visited on 05/02/2023).

[9] J. von Kistowski, S. Eismann, N. Schmitt, A. Bauer, J. Grohmann, and S. Kounev. “TeaStore: A Micro-Service Reference Application for Benchmarking, Modeling and Resource Management Research.” In: *2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. 2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS). ISSN: 2375-0227. Sept. 2018, pp. 223–236. DOI: `10.1109/MASCOTS.2018.00030`.

[10] *Scheduling Framework*. Kubernetes. Section: docs. URL: `https://kubernetes.io/docs/concepts/scheduling-eviction/scheduling-framework/` (visited on 06/10/2023).

[11] *Cluster-Autoscaling | Google Kubernetes Engine (GKE)*. Google Cloud. URL: `https://cloud.google.com/kubernetes-engine/docs/concepts/cluster-autoscaler?hl=de` (visited on 06/10/2023).

[12] *Karpenter*. Karpenter. URL: `https://karpenter.sh/` (visited on 05/14/2023).

[13] R. J. Vanderbei. *Linear programming: foundations and extensions*. 3rd ed. International series in operations research and management science. Boston: Springer, 2008. 464 pp. ISBN: 978-0-387-74387-5.

[14] F. P. Vasilyev and A. Y. Ivanitskiy. *In-Depth Analysis of Linear Programming*. Dordrecht: Springer Netherlands, 2001. ISBN: 978-90-481-5851-5 978-94-015-9759-3. DOI: `10.1007/978-94-015-9759-3`. URL: `http://link.springer.com/10.1007/978-94-015-9759-3` (visited on 05/02/2023).

[15] J. Matoušek and B. Gärtner. *Understanding and using linear programming*. Universitext. Berlin ; New York: Springer, 2007. 222 pp. ISBN: 978-3-540-30697-9 978-3-540-30717-4.

[16] “Mixed-Integer Linear Programming.” In: E. Castillo, A. J. Gonejo, P. Pedregal, R. Garciá, and N. Alguacil. *Building and Solving Mathematical Programming Models in Engineering and Science*. Hoboken, NJ, USA: John Wiley & Sons, Inc., Oct. 28, 2011, pp. 25–46. ISBN: 978-0-471-22529-4 978-0-471-15043-5. DOI: `10.1002/9780471225294.ch2`. URL: `https://onlinelibrary.wiley.com/doi/10.1002/9780471225294.ch2` (visited on 06/14/2023).

[17] P.-Q. Pan. *Linear Programming Computation*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014. ISBN: 978-3-642-40753-6 978-3-642-40754-3. DOI: `10.1007/978-3-642-40754-3`. URL: `https://link.springer.com/10.1007/978-3-642-40754-3` (visited on 05/02/2023).

[18] R. Bellman. "Dynamic Programming." In: *Science* 153.3731 (July 1966). Publisher: American Association for the Advancement of Science, pp. 34–37. DOI: 10.1126/science.153.3731.34. URL: https://www.science.org/doi/10.1126/science.153.3731.34 (visited on 06/14/2023).

[19] J. L. Higle and S. W. Wallace. "Sensitivity Analysis and Uncertainty in Linear Programming." In: *Interfaces* 33.4 (Aug. 2003), pp. 53–60. ISSN: 0092-2102, 1526-551X. DOI: 10.1287/inte.33.4.53.16370. URL: https://pubsonline.informs.org/doi/10.1287/inte.33.4.53.16370 (visited on 06/14/2023).

[20] Z. Wang, S. Zhu, J. Li, W. Jiang, K. K. Ramakrishnan, Y. Zheng, M. Yan, X. Zhang, and A. X. Liu. "DeepScaling: microservices autoscaling for stable CPU utilization in large scale cloud systems." In: *Proceedings of the 13th Symposium on Cloud Computing*. SoCC '22: ACM Symposium on Cloud Computing. San Francisco California: ACM, Nov. 7, 2022, pp. 16–30. ISBN: 978-1-4503-9414-7. DOI: 10.1145/3542929.3563469. URL: https://dl.acm.org/doi/10.1145/3542929.3563469 (visited on 11/16/2022).

[21] M. Wang, D. Zhang, and B. Wu. "A Cluster Autoscaler Based on Multiple Node Types in Kubernetes." In: *2020 IEEE 4th Information Technology, Networking, Electronic and Automation Control Conference (ITNEC)*. 2020 IEEE 4th Information Technology, Networking, Electronic and Automation Control Conference (ITNEC). Chongqing, China: IEEE, June 2020, pp. 575–579. ISBN: 978-1-72814-390-3. DOI: 10.1109/ITNEC48623.2020.9084706. URL: https://ieeexplore.ieee.org/document/9084706/ (visited on 11/16/2022).

[22] A. Chung, J. W. Park, and G. R. Ganger. "Stratus: cost-aware container scheduling in the public cloud." In: *Proceedings of the ACM Symposium on Cloud Computing*. SoCC '18: ACM Symposium on Cloud Computing. Carlsbad CA USA: ACM, Oct. 11, 2018, pp. 121–134. ISBN: 978-1-4503-6011-1. DOI: 10.1145/3267809.3267819. URL: https://dl.acm.org/doi/10.1145/3267809.3267819 (visited on 11/16/2022).

[23] Z. Ding, S. Wang, and C. Jiang. "Kubernetes-Oriented Microservice Placement with Dynamic Resource Allocation." In: *IEEE Transactions on Cloud Computing* (2022), pp. 1–1. ISSN: 2168-7161, 2372-0018. DOI: 10.1109/TCC.2022.3161900. URL: https://ieeexplore.ieee.org/document/9741392/ (visited on 11/28/2022).

[24] Z. Jian, X. Xie, Y. Fang, Y. Jiang, T. Li, and Y. Lu. *DRS: A Deep Reinforcement Learning enhanced Kubernetes Scheduler for Microservice-based System*. preprint. Preprints, Jan. 4, 2023. DOI: 10.22541/au.167285897.72278925/v1. URL: https://www.authorea.com/users/572171/articles/617148-drs-a-deep-reinforcement-

learning-enhanced-kubernetes-scheduler-for-microservice-based-system?commit=4c93783210431d97d7dcf3bb8b0c80d33a5b5baa (visited on 05/12/2023).

[25] M. Chima Ogbuachi, C. Gore, A. Reale, P. Suskovics, and B. Kovacs. "Context-Aware K8S Scheduler for Real Time Distributed 5G Edge Computing Applications." In: *2019 International Conference on Software, Telecommunications and Computer Networks (SoftCOM)*. 2019 International Conference on Software, Telecommunications and Computer Networks (SoftCOM). Split, Croatia: IEEE, Sept. 2019, pp. 1–6. ISBN: 978-953-290-088-0. DOI: 10.23919/SOFTCOM.2019.8903766. URL: https://ieeexplore.ieee.org/document/8903766/ (visited on 11/16/2022).

[26] D. Li, Y. Wei, and B. Zeng. "A Dynamic I/O Sensing Scheduling Scheme in Kubernetes." In: *Proceedings of the 2020 4th International Conference on High Performance Compilation, Computing and Communications*. HP3C 2020: 2020 4th International Conference on High Performance Compilation, Computing and Communications. Guangzhou China: ACM, June 27, 2020, pp. 14–19. ISBN: 978-1-4503-7691-4. DOI: 10.1145/3407947.3407950. URL: https://dl.acm.org/doi/10.1145/3407947.3407950 (visited on 11/16/2022).

[27] A. Horn, H. M. Fard, and F. Wolf. "Multi-objective Hybrid Autoscaling of Microservices in Kubernetes Clusters." In: *Euro-Par 2022: Parallel Processing*. Ed. by J. Cano and P. Trinder. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2022, pp. 233–250. ISBN: 978-3-031-12597-3. DOI: 10.1007/978-3-031-12597-3_15.

[28] H. M. Fard, R. Prodan, and F. Wolf. "Dynamic Multi-objective Scheduling of Microservices in the Cloud." In: *2020 IEEE/ACM 13th International Conference on Utility and Cloud Computing (UCC)*. 2020 IEEE/ACM 13th International Conference on Utility and Cloud Computing (UCC). Dec. 2020, pp. 386–393. DOI: 10.1109/UCC48980.2020.00061.

[29] *scheduler-plugins/kep/61-Trimaran-real-load-aware-scheduling at master · kubernetes-sigs/scheduler-plugins*. URL: https://github.com/kubernetes-sigs/scheduler-plugins/tree/master/kep/61-Trimaran-real-load-aware-scheduling (visited on 11/20/2022).

[30] M. M. Rovnyagin, S. O. Dmitriev, A. S. Hrapov, and V. K. Kozlov. "Algorithm of ML-based Re-scheduler for Container Orchestration System." In: *2021 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (ElConRus)*. 2021 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (ElConRus). St. Petersburg, Moscow, Russia: IEEE, Jan. 26, 2021, pp. 613–617. ISBN: 978-1-66540-476-1. DOI: 10.1109/ElConRus51938.2021.

9396294. URL: https://ieeexplore.ieee.org/document/9396294/ (visited on 11/16/2022).

[31] *Descheduler for Kubernetes*. original-date: 2017-07-28T17:11:38Z. Nov. 19, 2022. URL: https://github.com/kubernetes-sigs/descheduler (visited on 11/20/2022).

[32] *Use load-aware pod scheduling*. URL: https://www.alibabacloud.com/help/en/container-service-for-kubernetes/latest/use-load-aware-pod-scheduling (visited on 11/27/2022).

[33] J. Bi, Z. Yu, and H. Yuan. "Cost-optimized Task Scheduling with Improved Deep Q-Learning in Green Data Centers." In: *2022 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*. 2022 IEEE International Conference on Systems, Man, and Cybernetics (SMC). Prague, Czech Republic: IEEE, Oct. 9, 2022, pp. 556–561. ISBN: 978-1-66545-258-8. DOI: 10.1109/SMC53654.2022.9945426. URL: https://ieeexplore.ieee.org/document/9945426/ (visited on 11/23/2022).

[34] O.-M. Ungureanu, C. Vlădeanu, and R. Kooij. "Kubernetes cluster optimization using hybrid shared-state scheduling framework." In: *Proceedings of the 3rd International Conference on Future Networks and Distributed Systems*. ICFNDS '19: 3rd International Conference on Future Networks and Distributed Systems. Paris France: ACM, July 2019, pp. 1–12. ISBN: 978-1-4503-7163-6. DOI: 10.1145/3341325.3341992. URL: https://dl.acm.org/doi/10.1145/3341325.3341992 (visited on 11/16/2022).

[35] Z. Rejiba and J. Chamanara. "Custom scheduling in Kubernetes: A survey on common problems and solution approaches." In: *ACM Computing Surveys* (June 24, 2022), p. 3544788. ISSN: 0360-0300, 1557-7341. DOI: 10.1145/3544788. URL: https://dl.acm.org/doi/10.1145/3544788 (visited on 11/16/2022).

[36] *scheduler-plugins/install.md at master · kubernetes-sigs/scheduler-plugins · GitHub*. URL: https://github.com/kubernetes-sigs/scheduler-plugins/blob/master/doc/install.md (visited on 06/12/2023).

[37] *Specifying a Disruption Budget for your Application*. Kubernetes. Section: docs. URL: https://kubernetes.io/docs/tasks/run-application/configure-pdb/ (visited on 06/12/2023).

[38] *GKE-Standardknotengrößen planen | Google Kubernetes Engine (GKE)*. Google Cloud. URL: https://cloud.google.com/kubernetes-engine/docs/concepts/plan-node-sizes?hl=de (visited on 05/06/2023).

[39] *GKE release notes | Google Kubernetes Engine (GKE)*. Google Cloud. URL: https://cloud.google.com/kubernetes-engine/docs/release-notes (visited on 06/12/2023).

[40] *Kubernetes-Versionen für Amazon EKS - Amazon EKS*. URL: https://docs.aws.amazon.com/de_de/eks/latest/userguide/kubernetes-versions.html (visited on 06/12/2023).

[41] palma21. *Supported Kubernetes versions in Azure Kubernetes Service (AKS). - Azure Kubernetes Service*. June 8, 2023. URL: https://learn.microsoft.com/en-us/azure/aks/supported-kubernetes-versions (visited on 06/12/2023).

[42] Soma. *Top 10 Microservices Design Principles and Best Practices for Experienced Developers*. Javarevisited. May 7, 2023. URL: https://medium.com/javarevisited/10-microservices-design-principles-every-developer-should-know-44f2f69e960f (visited on 06/12/2023).

[43] W. R.O.F. *Answer to "GKE autoscaler 'optimize-utilization'"*. Stack Overflow. June 5, 2020. URL: https://stackoverflow.com/a/62218694/21796236 (visited on 06/13/2023).

[44] *Alibaba Cluster Trace Program*. original-date: 2017-09-05T03:16:34Z. June 2, 2023. URL: https://github.com/alibaba/clusterdata (visited on 06/03/2023).