THIJS VAN DER KNAAP

REDUCING OPERATIONAL COSTS OF MICROSERVICES BY MEANS OF A DEPLOYMENT TUNER

university of groningen

faculty of science and engineering

REDUCING OPERATIONAL COSTS OF MICROSERVICES BY MEANS OF A DEPLOYMENT TUNER

THIJS VAN DER KNAAP

SUPERVISORS:

V. Andrikopoulos

A. Lazovik

Groningen, August 2019

Thijs van der Knaap: *Reducing operational costs of microservices by means of a deployment tuner* © August 2019

SUPERVISORS:
V. Andrikopoulos
A. Lazovik

LOCATION:
Groningen

TIME FRAME:
August 2019

ABSTRACT

Running a microservices architecture on the cloud can have significant cost advantages, especially when load on the system varies. Operational cost can be minimised by ensuring that just enough resources are requested, without harming performance. Autoscalers have been developed which analyse the system and scale accordingly, but even more cost can be saved by applying deployment tuning.

This work researches the financial benefits of expanding the standard Kubernetes autoscaler with an automated deployment tuner. The tuner reduces operational cost by actively searching for deployments that require less resources. The tuner is able to modify the current deployment and acts when the deployment is stable, this ensures that it does not compete with the autoscaler. It tunes based on the current load, deployment and stored previous deployments. When no cost gains can be made the tuner still expands its knowledge by performing small changes to the deployment, which also prevents the system getting stuck in a local minimum.

The evaluation shows that the tuner decreases operational cost of a cluster consisting of a single microservice. These savings do come with a decrease in performance. Further evaluation with a full microservice system is needed to judge the full cost saving potential of deployment tuning.

ACKNOWLEDGMENTS

First, I would like to thank my supervisor Vasilios Andrikopoulos for all the interesting discussions and useful insights. Without his guidance this research would not have been possible.

Second, I would also like to thank my colleague students Bogdan Petre and Patrick Vogel. They were always available to listen to my complaints and provide constructive feedback.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

1 INTRODUCTION

Microservices is an approach applied where there many services that have their own life cycle and collaborate together [1]. Microservices are becoming an industry standard for cloud based solutions. Their growth in popularity is still increasing [2]. Big companies like Netflix and Amazon [3] have already transitioned. As with all software, also each microservice eventually runs on hardware. This hardware is bought or rented which generates operational cost.

The aim of this research is to investigate if re-deployment capabilities and operational history awareness as a component working in collaboration with an autoscaler can lead to the reduction of operational costs. The proposed component is a deployment tuner which is introduced in Section 1.1. The process of re-deployment using history awareness will from now on be called deployment tuning. This results in the following research question:

> **Can deployment tuning reduce the operational cost of microservices?**

This research is applied on microservices, because one of the core abilities of microservices is the ability to scale each service independently [1]. This allows the system to acquire to correct scale for every component to deal with all the incoming load while minimising the required resources.

Autoscalers are applied to automate this scaling. An autoscaler is a program that dynamically allocates and deallocates resources for a particular application [4]. To perform this scaling the autoscaler needs to be aware of the workload that is put on the application. The goal of an autoscaler is to keep the application cost effective while ensuring that the quality of service is maintained. Autoscalers are covered in depth in Section 2.1.

Autoscalers are normally dedicated to scaling a single component. They can not optimise the system from a multiple component perspective. The deployment tuner, which is introduced in the next section, will be able to make decisions and changes using this perspective.

1.1 THE DEPLOYMENT TUNER

To perform the deployment tuning a tuner is needed. An important decision while building this tuner was to let the tuner run next to the original autoscaler. This allows the tuner to enhance the quality of the scaling performed by the autoscaler. The tuner will only act when the system is stable, meaning that the current deployment is able to handle the current incoming load. This ensures that it will not compete with the autoscaler, but will instead act when the autoscaler is inactive.

The design of the tuner is based on the MAPE-K loop proposed by IBM [5], providing an architecture for autonomic systems. The Knowledge base proposed in this architecture enables the awareness of deployments. The MAPE-K loop is further explained in Section 2.1.

The decision to maintain the normal autoscaler and only let the tuner act in stable situation has two large advantages.

The first advantage is that the tuner only has to focus on the actual redeployment for the current load. As scaling up and down due to load change does not have to be considered, it adheres to the principle set by Doug McIlroy, one of the founders of the UNIX tradition, "*Make each program do one thing well*" [6]

The second advantage gained by not replacing autoscaling, is that the tuner does not have to respond in real-time. The tuner should still be able to act in a reasonable amount of time, but this does not have to be instantaneous. This allows the tuner to perform a more advanced analysis, while the quick scalability is still ensured by the original autoscaler.

The goal of this work is to verify if a deployment tuner is able to reduce the cost of a microservice system and to give a clear explanation on how a tuner can be designed. By using the MAPE-K architecture also the design of single components of this loop can be useful for other self-optimising systems. Special care is taken in the design of the tuner to allow for the usage of previous deployments while ensuring that the tuner also tries to visit new states.

1.2 DOCUMENT STRUCTURE

This research thesis including the introduction consists in total of seven chapters. Chapter 2 discusses the background for the technologies used and the related work. Chapter 3 covers the requirements for the tuner and its design. Chapter 4 explains how the design is implemented and tested. The tuner is evaluated in Chapter 5. Conclusions concerning the whole research and interesting future work can be found in Chapter 6.

2 BACKGROUND AND RELATED WORK

This chapter consists of two sections. Section 2.1 covers all relevant technologies that where used in this work. Section 2.2 covers papers related to reducing cost in a scalable environment.

2.1 BACKGROUND

2.1.1 *Microservices*

Microservices are autonomous services that work together [1]. Each service should be exactly focused on only one thing. All functionality in a service should be directly concerned with its purpose, ensuring high cohesion. Low coupling is acquired by standardising the communication between services, this allows for the modification of one service without affecting others. High cohesion and low coupling are only attained when the boundaries between services are set up correctly.

When these boundaries are correctly constructed and maintained you gain a lot of advantages. The most important ones and their implications are listed below [1].

- **Technology Heterogeneity**: Every service can use its own technologies, for example the programming language used. The communication between services is standardised, and as long as both parties keep to that standard the used local implementation has no effect outside of its service.
- **Resilience**: A single service can fail, but this does not have to bring down the whole system. The service in question can simply be removed and replaced by a new copy.
- **Scaling**: As stated before every service can be scaled independently, allowing single services to be scaled when their load increases.
- **Ease of deployment**: A change to a service only has to be deployed on that service, therefore versions can be iterated quickly. The change can also be rolled back quickly, easing the development even more.

2.1.2 *Containerisation*

To acquire many of the microservice benefits stated in the previous section it is essential that every service runs in its own environment. This ensures that services can only communicate using the intended means.

Running a (virtual) machine for every instance of a service results in a significant overhead. Therefore containerisation has been developed allowing each service to run in a lightweight container. This allows for the same benefits as a full fledged (virtual) machine per service, but only produces a fraction of the overhead [7]. Next to this reduction in overhead containerisation also aids in standardising the services, allowing general solutions to be build. The most used containerisation tool currently used is Docker [8]. Docker allows lightweight containerisation that can run on almost every machine.

2.1.3 *Orchestration*

Due to the standardisation acquired by containerisation more advanced tools could be constructed. One of these tools is orchestration. An orchestration tool allows the user to define the desired deployment and the tool will ensure that this deployment is created and maintained [9].

Two well know orchestration tools are Docker swarm and Kubernetes. Docker swarm allows the user to specify which containers should run in the system. Next to this one or multiple (virtual)machines can be provided to run these containers. Docker swarm will then place the given containers on the available machines.

Kubernetes allows the user to define per service how it should be deployed. Such a service definition runs then at least one pod. A pod is a grouping of one or multiple containers and is the smallest scalable unit in Kubernetes. This service definition states how many of the given pods should at all time be present in the system. This amount of pods is then scheduled on the machines available in the system. These machines are called nodes.

Every pod can have a requested and a limit CPU and memory value set. The requested value indicates how much of this resource the pod needs to function properly. The limit value states the maximum amount that the pod is allowed to use. The requested value for both CPU and memory is used to schedule pods on nodes. The sum of all requested values of pods present on a node will never succeed the amount of resources present on the node.

There are three types of deployments possible in Kubernetes, each having their own characteristic behaviour.

- **Deployment**: This is the most basic option and is used for stateless containers. Pods being part of a deployment can be killed and later restarted.
- **Stateful set**: As the name already suggests these pods have a state. Therefore they should be treated with more care and can not simply be removed from the system. This is used for example for a database.
- **Daemon set**: A daemon set specifies pods that have to run on every node in the system. This is normally not used for real services aimed at the client, but are pods that support the actual services. A use cases for a daemon set is for example monitoring, as metrics from every node are required.

2.1.4 *Autoscaling*

The autoscaler is responsible for scaling the deployment of a system, so that it is able to cope with the load that is put on the system. A good autoscaler does not only allow the system to cope with the load, but also ensures that it is doing this while not wasting resources. This scaling can be performed both vertical and horizontal [10]. Vertical scaling means that the resources available for the already running container(s) of a service changes. With these acquired resources the changed load can be handled. Horizontal scaling means that the amount of containers that provide the service is scaled. Every container requests a certain amount of CPU and when you scale the number of containers you also change the amount of CPU dedicated to this service.

There are two types of autoscalers, Reactive and proactive [4]. Reactive scaling means that the autoscaler only looks at the current state and based on this the deployment is scaled. The downside to this approach is that the scaling will always be too late, as the necessity of scaling is only noticed when the extra resources are already required. Examples of reactive scaling are the Kubernetes pod [11] and cluster [12] autoscaler or work by Soundararajan et al. [13]. Proactive scaling tries to solve this problem by predicting the required resources in the future. This allows the autoscaler to scale the system before the load increase actually arrives. Of course this prediction can also be wrong, resulting in either wasted resources or under provisioning. Examples of a proactive autoscaler can be found in work by Chen et al. [14] and Ashraf et al. [15].

2.1.5 *MAPE-K*

The proposed tuner will be an autonomic system striving for cost optimisation. It will be active when the autoscaler is not making

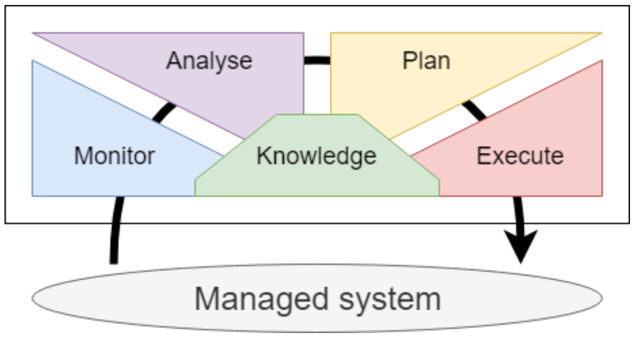

Figure 2.1: MAPE-K loop [5]

changes due to the load being constant. To attain this autonomic behaviour the MAPE-K loop architecture [5] is used. The architecture discusses all the elements needed to build a self-optimising system.

The MAPE-K loop consists of five components: monitor, analyse, plan, execute and knowledge. All these components are needed to build the self-optimising system and every step is performed in sequence as can be seen in Figure 2.1. The next list covers every component and their purpose.

- **Monitor** is concerned with retrieving the current state of the managed system and aggregating these results.
- **Analyse** uses the results found during the monitor-step and combines this with the knowledge stored in the knowledge base and generates a change request if change is desired. A change request is only generated when the resulting change will improve the performance of the managed system.
- **Plan** receives the change request and creates a change plan. This represents the desired set of changes that have to happen to reach the requested state.
- **Execute** is responsible for actually changing the state of the managed resource, it receives the change plan and executes its desired steps.
- **Knowledge** stores the knowledge gained by the system. Every iteration of the loop this knowledge will be extended making the system smarter as it runs longer.

2.2 RELATED WORK

This section covers four papers that each take a different approach on how deployment selection should be performed. All the papers use their deployment selection strategy to minimise the operational cost of the deployment.

Sharma et al. [16] present in their paper a provisioning framework called Kingfisher. This framework is designed to reduce the cost of running a system consisting of multiple virtual machines. The framework uses replication and migration of virtual machines to perform scaling while minimising the cost.

Their decision logic is based on an initial analysis where the load handling of each component is analysed on the different available machines in the system. Due to this static analysis the Kingfisher framework only performs actions when the load changes. the resulting deployment should already be optimal. The static analysis does yield a downside, as the analysis has to be performed again when a component gets modified. They limit the search space in the paper by supplying only one small, one medium and one large machine as possible options for running a virtual machine.

The framework does not only decide what the ideal configuration is, but it also generates and executes the required changes to acquire the configuration. In addition to rental costs of the hardware they also take into account supplementary costs induced by transitioning from the current deployment to the preferred deployment. Their results show that especially the migration option is essential for attaining efficient use of resources in a heterogeneous environment. The migration option replaces multiple smaller virtual machines for fewer bigger machines. The paper only contains an evaluation for increasing the load on the system, but the concept should also work for down-scaling.

Chen et al. [14] propose an autonomic provisioning system based on machine learning to ensure that their database meets the set Service Level Agreements (SLA) while minimising over-provisioning. The provisioning system performs pro-active scaling by using K-nearest-neighbours (KNN) classifier as the prediction algorithm. This work is a continuation on their previous paper [13], where they create and verify a reactive system for the same use-case that scales when SLA are broken. The pro-active provisioning breaks less SLAs than their reactive version.

The KNN classifier is trained using data generated by the same source as is also used for the verification. They found that using a KNN classifier gives more stable results than direct aggregation of the learning data. This is interesting, because our proposed tuner uses the historical data directly. It might be beneficial to apply machine learning on the knowledge base to enhance the knowledge extracted from it.

The proposed solution does have the same disadvantage as the scaling solution proposed by Sharma et al. in the previous paper. The constructed machine learning algorithm requires training data. This makes changes to the system more costly as the KNN classifier has to be retrained. In our use-case online machine learning would be required to enhance the knowledge base data while the data is being gathered. The investigation in an online machine learning algorithm for provisioning is also listed by the paper as interesting future work.

Andrikopoulos [17] opts that cloud systems should have dynamic topology selection. The option space when working with topologies explodes quickly, therefore he argues that it could be dealt with by treating it as a graph and performing transitions on them. This same property was also used in designing the tuner presented in this thesis, Going over every possible option is impossible. The only options considered are a result of a direct change to the current deployment to reach new unknowns or going back to a remembered stored option which was better.

Andrikopoulus stresses that it is really important to not only generate and use different topologies, but also to evaluate their performance after usage. This also holds for deployments, as keeping track of previous deployments without being able to evaluate them reduces the value of the knowledge dramatically.

Townend et al. [18] discuss achieved savings on electrical cost by implementing a holistic Kubernetes scheduler. In their research a private cloud with a constant amount of machines is used. The cost saving is achieved by trying to put the ideal load to electricity balance on as many nodes as needed, and idling the rest of the nodes.

The idling cost almost no power and can be compared to our case of shutting down a machine. Due to their idling they are able to quickly spin up a node again, since it is still available. Their scheduler therefore is able to respond quicker to the changing situation than our tuner combined with an autoscaler that has to start and stop machines.

3 ARCHITECTURE

The tuning system presented in this chapter is build to work with Kubernetes, but the concepts covered can be applied on any orchestration tool. The chosen orchestrations tool does have to allow the tuner control over the placement of the containers.

3.1 REQUIREMENTS

The requirements are divided into functional and non-functional requirements.

Functional requirements

FR01 The system must be able to retrieve the current load on the deployment.

FR02 The system must be able to retrieve the current deployment.

FR04 The system is able to check that the current deployment is able to handle the incoming load and therefore is stable.

FR05 The optimisation and knowledge addition are only performed when the deployment is stable.

FR06 The structure of a stable deployment must be stored.

FR07 The system must never select a more expensive distribution if a valid cheaper one is available.

FR08 The system must remain searching for better options to circumvent getting stuck in a local minimum concerning cost.

FR09 The system must be able to add/migrate/remove specific pods on specific nodes.

FR10 The system must be able to remove a node from the deployment.

FR11 The system must verify the success of every action.

FR12 The system must not perform actions with stateful containers.

FR13 The system should reach the desired state in the least amount of changes possible.

Non-functional requirements

NFR01 The system must minimise the required cost to run the deployment.

NFR02 The system should not harm performance of the deployment.

3.2 DESIGN

The architecture presented in this section is designed to fill the requirements set in the previous section. The tuner is designed using the MAPE-K loop (Section 2.1) as the backbone of the tuner system.

The logical view which can be seen in Figure 3.1 shows all the main components of the system. All the components of the MAPE-K loop are present and they are managed by the Moderator. The Moderator performs the interactions between the components and runs the actual loop. The Kubernetes system is treated as an external component from which information can be extracted and changes can be performed on.

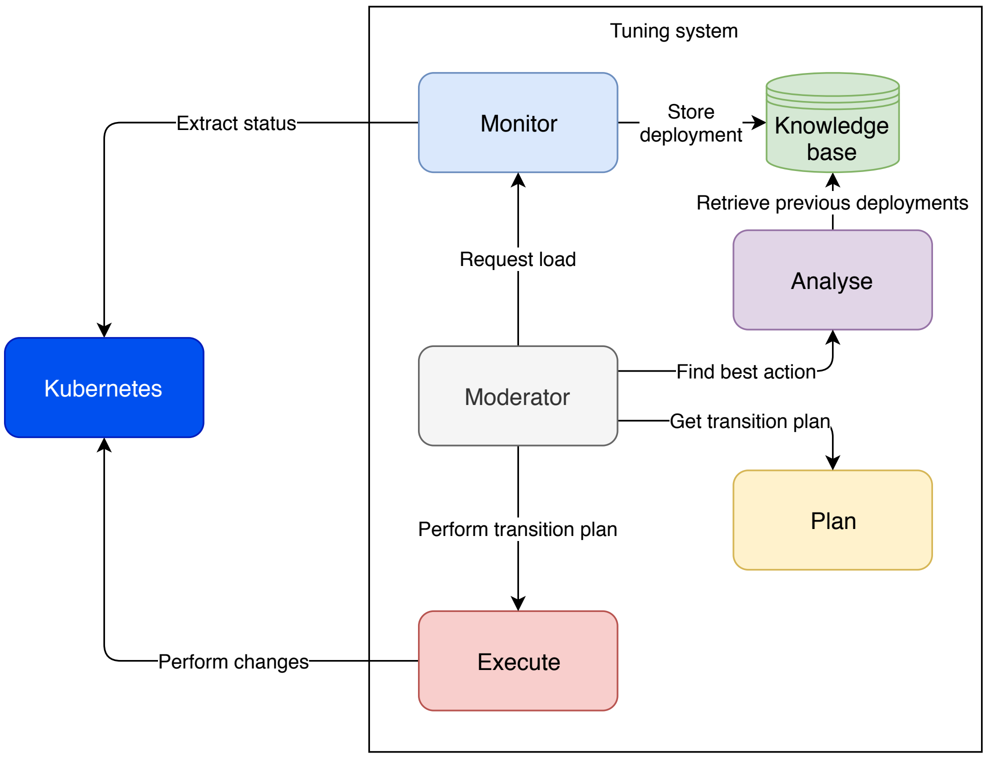

Figure 3.1: Logical view

The process view, Figure 3.2, shows the actual steps that are performed in the tuner's MAPE-K loop. The colour coding indicates for every step to which component it belongs. The following sections will discuss what actually happens at each of these steps.

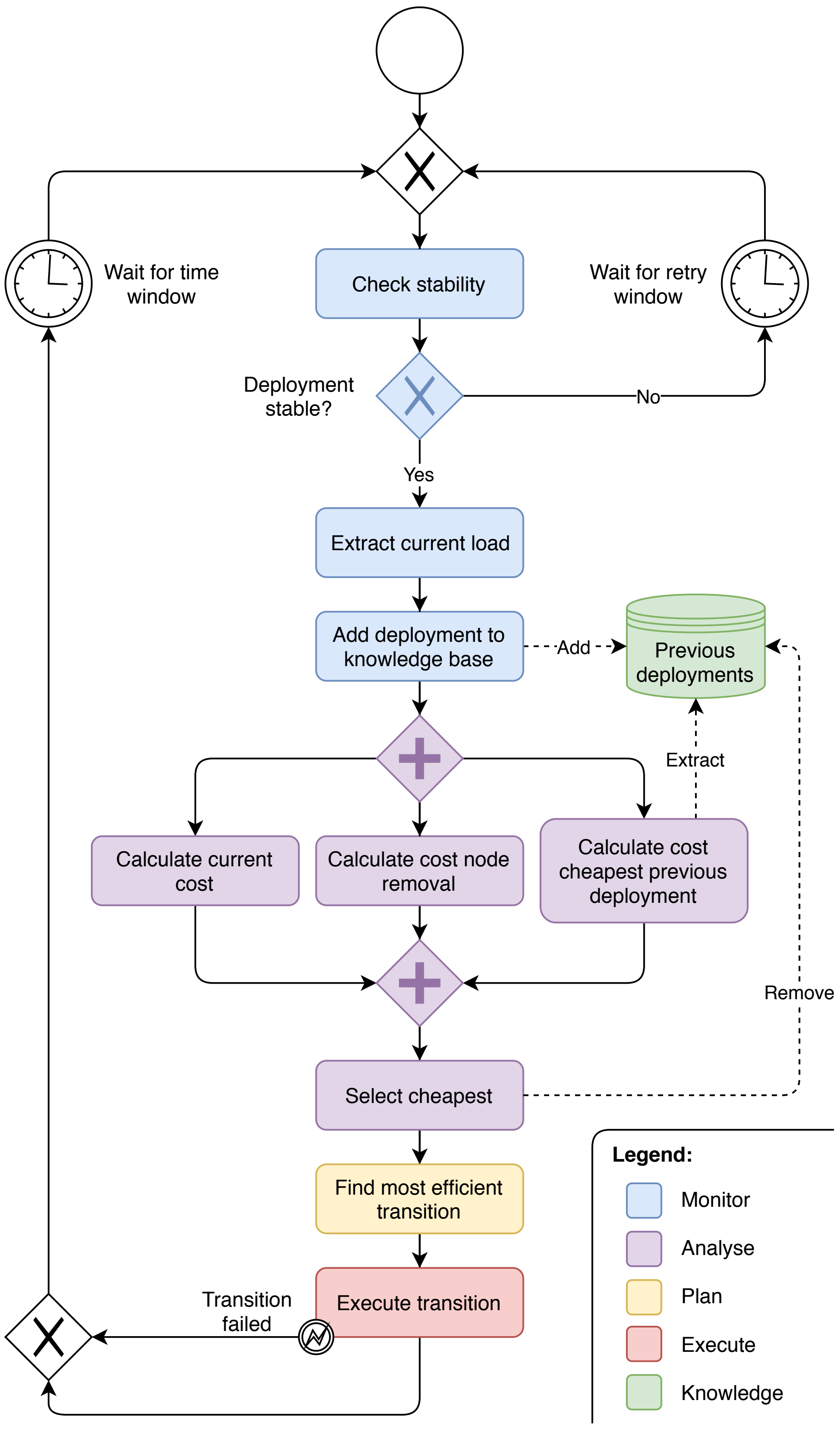

Figure 3.2: Process view, using BPMN

3.2.1 *Knowledge*

The knowledge base consists of a database that holds the actual structure of deployments of previous execution time windows and the load these deployments dealt with. It is important to store this information as it allows the system to return to this deployment when it sees that it is favourable over the current deployment.

3.2.2 *Monitor*

The monitoring is concerned with retrieving the actual current metrics concerning the containers and machines of the current deployment and the load that is applied on this deployment. More specifically, it performs the following functionalities:

Checking stability

The stability check is performed by comparing the deployment of the system at the start and end of the time window. The amount of machines and the amount of containers of every different group are compared and only when both have not changed the system is deemed stable.

When the deployment is unstable the tuner will wait for the specified amount of time (`retry window`) to check the stability again. The `retry window` is added to allow the user to specify the retry frequency of the sliding window.

Extract the current load

When the current deployment is stable the system needs to find the load on the system that this deployment was able to deal with. The load extraction is project dependent as different type of applications will receive different types of load. When implementing the load extraction it is important to ensure that the load measured is directly representative for the size of the deployment. A load extraction variable can be found by first analysing the scaling of the system, and then finding a variable that follows a similar pattern. Section 4.4 provides an example.

Add deployment to knowledge base

The current deployment is stored in the knowledge base using the load value the deployment dealt with as its key. This allows the tuner system to later query the knowledge base for deployments that where able to deal with a certain amount of load.

3.2.3 *Analyse*

The analysis component actually is the real brains of the system. The rest of the components are simply there to provide data and tools to aid the analysis. The tuner is able to perform three different actions. The analysis component predicts the resulting cost of these actions and decides which action requires least operational cost. The resulting transitions, needed to execute the chosen action, are then forwarded to the planner and executor to be performed.

The three paragraphs below explain each one of the possible actions of which the resulting cost has to be predicted. The paragraph covers the selection of the cheapest action using the predicted costs.

Calculate current cost

The cost per hour of the current deployment is calculated by multiplying the amount of cores and memory in the deployment by their appropriate costs. The user sets the chance that a random pod migration is performed. For this random migration space has to be available on a receiving node to ensure that the migration can be successful. This does not reduce the cost of the system, but does allow the system to enhance its knowledge base and reduces the chance that the system gets stuck in a local minimum as the deployment keeps on changing.

Calculate cost node removal

To remove a node from the current deployment there has to be a enough space available on the other nodes to hold the pods currently present on the node that is to be removed. This check is performed using a Depth First Search (DFS). The depth of the search tree is equal to the amount of pods that have to be rescheduled, therefore DFS will always provide the most efficient option. In addition there should also be no stateful sets on the node as these can not be migrated. The cost is calculated of the deployment with the node removed. When it is impossible to remove a node the resulting cost will be infinite.

Calculate cost cheapest previous deployment

To find the cheapest previous deployment able to deal with the load the knowledge base is used. All previous deployments that dealt with a similar load are retrieved. From these the cheapest is selected and the nodes are matched to nodes in the current deployment. This can not be simply done using the name of the nodes, but is performed using node contents.

The current nodes therefore have to be matched to the desired nodes. Every current node is compared to every desired node using a scoring scheme. The scoring scheme is designed to give preference to matching stateful sets, as these can not be moved. A pod being part of a stateful set is worth 100 points and a normal pod 1 point. This distinction is made to give major preference to stateful sets as these can not be moved. When two nodes are compared plus points are given for each matching pod. Points are subtracted when pods do not match and therefore have to be added or removed. The mapping from each current node to nothing is also added, as this is the idea of going to an previous deployment, reducing the amount of nodes needed. Using the found scores each current node is matched to an desired node resulting in the highest overall score.

The matched nodes are then inspected to check if no stateful sets have to be moved. If they have to be moved the chosen previous deployment becomes unattainable. Its entry will be removed from the knowledge base and the search will be performed again to find the new best candidate. When the cheapest attainable previous deployment is found its running cost is calculated.

Select cheapest

Each of the before explained actions have a resulting cost. The action resulting in the cheapest deployment is selected, as the goal of the tuner is to minimise the the cost of the deployment. Of course it is possible that more then one action results in the cheapest deployment. When there is a draw the following rules are used to decide between them. These rules are based on the principle that impact of the tuner on the deployment has to be minimised, therefore the least amount of changes are preferred. The following listing will consider all possible draw cases and explain why the resulting action is preferred.

- $cost_{current} \neq cost_{node\_removal}$
  When it is possible to remove a node from the current deployment, it will always result in a lower cost. If this is not possible the cost of node removal will be infinite.

- $cost_{node\_removal} = cost_{previous\_cheapest} \rightarrow node\_removal$
  When the deployment with a node removed has the same cost as a the cheapest previous deployment, it means that the cheapest previous deployment also clears one node. Getting to the previous deployment might require additional actions next to emptying the to be removed node. The node removal option will not require this and therefore needs less actions. In addition this also enhances the knowledge base further as a deployment that was not yet in the knowledge base might be reached.

- $cost_{current} = cost_{previous\_cheapest} \rightarrow current$
  When the current cost is equal to the cheapest previous deployment then changing to this previous deployment will not yield any cost benefits. Staying with the current deployment and

maybe migrating one pod is always less changes then changing to the found previous deployment. Therefore the current deployment is preferred.

Using these comparison rules an algorithm can be constructed to ensure that always the correct action is chosen. This pseudo code for the comparison can be found in Algorithm 1.

```
   Data: $deployment_{current}$, $load_{current}$
   Result: $transitions_{cheapest}$
 1 $cost_{current}, transitions_{current} = CalculateCurrentCost(deployment_{current})$
 2 $cost_{node\_removal}, transitions_{node\_removal} = CalculateCostNodeRemoval(deployment_{current})$
 3 $cost_{previous\_cheapest}, transitions_{previous\_cheapest} = ExtractPreviousCheapestDeployment(load_{current})$
 4 if $cost_{node\_removal} \leq cost_{previous\_cheapest}$ then
 5     $transitions_{cheapest} = transitions_{node\_removal}$
 6 else if $cost_{current} \leq cost_{previous\_cheapest}$ then
 7     $transitions_{cheapest} = transitions_{current}$
 8 else
 9     $transitions_{cheapest} = transitions_{previous\_cheapest}$
10 end
```

Algorithm 1: Select cheapest action

When the cheapest previous deployment action gets selected as being the preferred option the selected deployment its entry is removed from the knowledge base. Removing the entry ensures that a single good run does not cause the system to keep striving for this deployment. It can actually be seen as a verification step, because the system is changing back to the found deployment and if the load does not change it should be able to endure the whole next time window. When the time window is finished with the deployment still being stable it will be added back to the knowledge base. This keeps our knowledge base reliable.

3.2.4 *Plan*

The plan component receives the desired changes that have to happen to each of the nodes currently in the system. These changes consist of a list of additions and a list of removals that have to be performed. The goal of the plan step is translate this list of changes into the least amount of actions possible which will eventually be performed in the execution step. The system is able to perform four operations on the deployment of the Kubernetes cluster:

- Adding a pod to a node.

- Removing a pod from a node.
- Migrating a pod from one node to another.
- Deleting a node.

Find most efficient transition

When a pod has to be removed from one node and that same type has to be added to another it is preferred to perform this action using a migration. This affects the system less then performing a separate removal and addition. Therefore the goal of the planning step is to perform the transition with the maximum amount of migrations possible. Sometimes migrations are not possible, for example in case where two nodes are totally filled and two pods have to be migrated between them. This would be possible by first removing one pod then migrating the other and after that adding the first one again.

Constructing the planning starts with identifying all possible migrations. All possible migration combinations and orders are then found using a recursive loop over all found migrations. Before performing this loop all migrations are shuffled, to ensure that the migrations are evenly spread out over the whole cluster and not one node at a time is changed a lot. A migration is deemed possible when the current requested CPU on the node of the already present pods plus the requested CPU of the pod is less then the actual CPU cores available. The migration plan containing the most migrations is chosen.

When the best migration plan is known the pod additions and removals can be aggregated, by removing the migrations from the initial desired change. All the pod additions, pod migrations, pod removals and node removals are formatted such that the execute step can perform them.

3.2.5 *Execute*

The execute component is responsible for actually realising all the plans that the planner suggests. It can perform the four actions that the planner introduced. After an action has been performed the result should also be verified. If the action succeeds the transition can continue with the next step. If the action failed, for example a pod was scheduled on the wrong node, the transition is stopped, because the desired deployment can not be reached anymore using the provided plan. The tuner will be idle till the end of the next time window, because the transition might also have failed due to the autoscaler scaling the deployment while the transition is happening.

3.2.6 *Reiterating the loop*

After the execution step is completed an iteration of the MAPE-K loop is finished. The tuner will wait for the specified time window, before starting a new iteration of the loop with the monitor step.

4 IMPLEMENTATION AND TESTING

This chapter will discuss how the tuner is implemented and its features are tested.

Figure 4.1 shows the physical view of the implementation. There is a Kubernetes cluster present that contains a variable amount of nodes and pods. On the right of the cluster the local machine can be seen. It runs the actual tuner software and holds the knowledge base. In the following sections all the components will be explained in more detail and the implementation of the tuner will be discussed.

4.1 KUBERNETES CLUSTER

Initial development of the tuner was performed using Minikube[1] (Version `1.0.0`) which allows the user to create a Kubernetes cluster with a single node on a private machine. As the system expands multiple nodes are needed to test and verify the behaviour of the system. Therefore the decision was made to transfer to Google Kubernetes Engine[2]. Here virtual machines are used that contain 2 virtual CPU cores (Intel Xeon Scalable Processor running with a 2.0 GHz base frequency) and 7.5 Gigabytes of virtual Random Access Memory. Automatic node scaling is used to allow the system to dynamically react to increase and decrease of required resources. The system runs with Kubernetes version `1.11.8` and the nodes are located in zone `europe-west3-c`.

4.1.1 *Autoscaler*

The standard autoscaler of Kubernetes which is used next to the tuner is a reactive horizontal autoscaler. An autoscaler is defined per service. This autoscaler normally scales on CPU usage, but can be configured to scale on a custom value like size of a job queue [11]. If you set the value of the autoscaler to 80 percent of CPU then the autoscaler will make/remove as many pods such that each remaining pod is using 80 percent of its requested CPU amount.

The scaling of nodes in Kubernetes is done indirectly. The amount of nodes will scale up when all nodes in the cluster are full and a new pod needs to be added. Scaling down is performed by checking whether there are nodes in the system that do not run pods anymore, except for daemon set pods. These nodes are only removed after a certain time of inactivity.

1 `https://github.com/kubernetes/minikube`
2 `https://cloud.google.com/kubernetes-engine/`

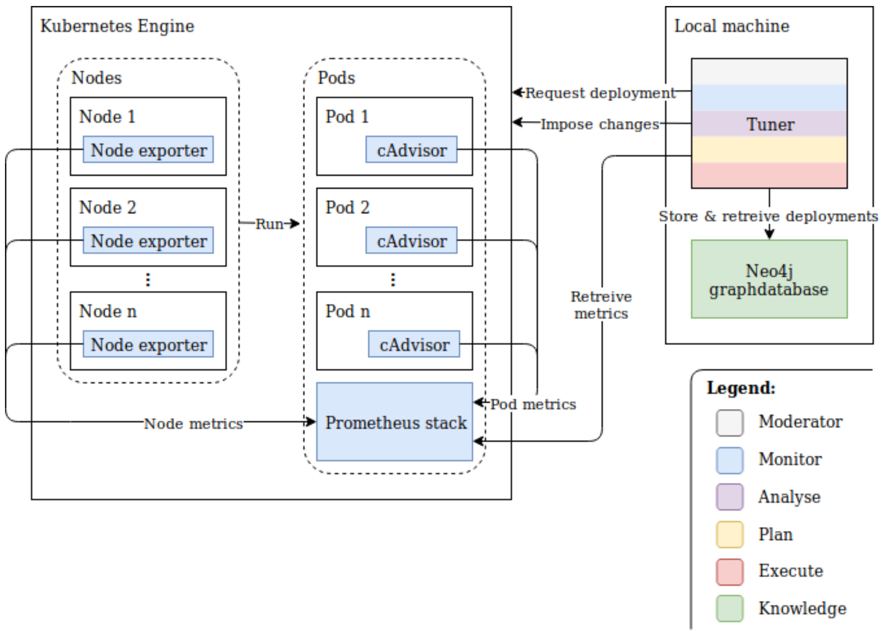

Figure 4.1: Physical view

4.2 LOCAL MACHINE

The local machine runs the actual tuner system and a Neo4j graph database which functions as the knowledge base (Section 4.3). The local machine that is used for testing is a HP ZBOOK Studio G5. It contains an Intel(R) Core(TM) i7-8750H CPU @ 2.20GHz containing 6 cores. There is 16 gigabyte DDR4 RAM present, consisting of two 8 gigabyte cards running at 2667 MHz. The long term storage is a 256 gigabyte SSD.

4.3 KNOWLEDGE BASE

The knowledge base is implemented using the graph database Neo4j[3] (Version `3.5.8`) running locally in a docker container. It holds the actual structure of previous execution time windows. A graph database is chosen because it allows to directly store the actual structure of the deployments. The graph structure could also allow for future structure mining. Cypher[4] is the query language used by Neo4j, which is a powerful and expressive language that allows for intuitive usage. Figure 4.2 shows the structure of multiple execution windows stored in a graph structure. On the top left of the figure you can see the legend, the first row concerns the vertexes and the second covers the edges. The green vertexes are executions, they represent an execution window. This execution window has properties like start and end time. It also had certain Kubernetes nodes that where running during

3 https://neo4j.com/
4 https://neo4j.com/docs/cypher-manual/current/

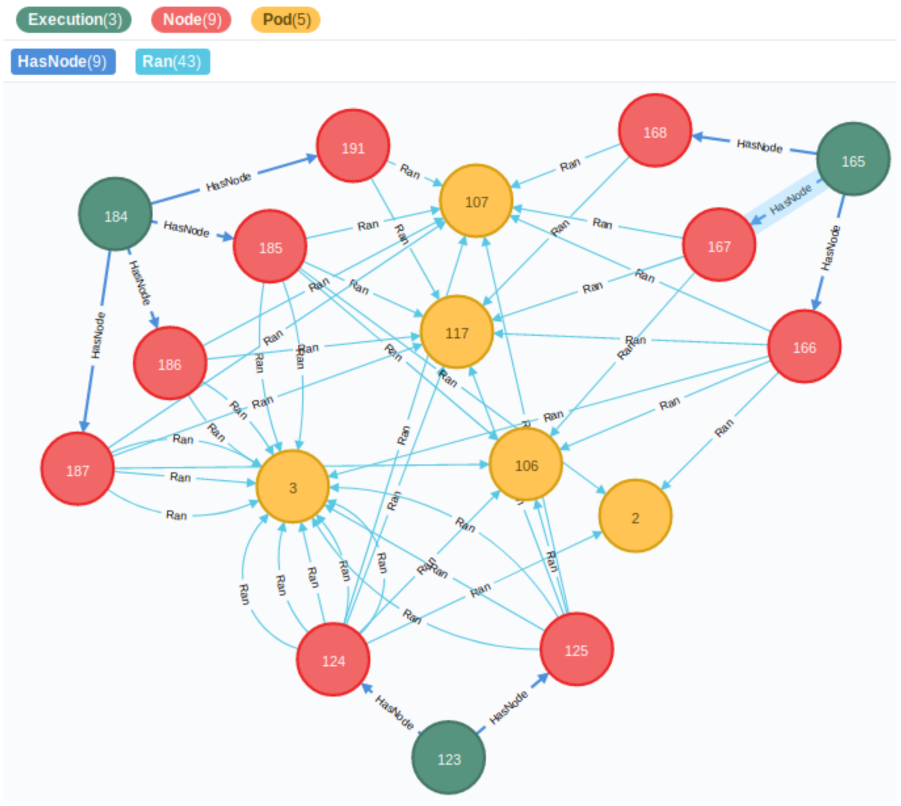

Figure 4.2: Graph database structure

its execution, those are the red vertexes that are connected with a HasNode relationship. Every Kubernetes node ran pods which are represented by the Ran relationship. These Ran relationships point to a deployment pod description which are the yellow vertexes. The whole structure is shaped like an centre core with multiple mantels. The pod descriptions form the centre. The Kubernetes nodes are the first shell and the executions are the final layer.

To further clarifying the structure, one of the executions in Figure 4.2 is used as an example. The example focuses on the execution represented by vertex 184 (top left). The colour of the vertex is green, identifying it as an execution. When this execution is examined the start and end time of the execution window can be found. In this case the window started on 2019-06-18, 15:05:46.39 and ended on 2019-06-18, 15:05:51.39. This can not be seen in the figure, but are properties stored in the vertex. Vertex 184 has four edges labelled with HasNode that point to four node vertexes, this shows that the execution had four nodes running during the execution window. Vertex 186 represents one of the nodes running in execution vertex 184. When inspecting the properties of this vertex the CPU and RAM can be found. This node contained 2 cores CPU and 7.5 Gigabytes of RAM (again not visible in the figure). The node ran four pods which can be seen by the Ran edges connecting this vertex to Pod vertexes. Every Ran edge represents one pod of the vertex Pod type. So node vertex 186 ran one pod of type 107, one of type 117 and two pods of type 3. Looking into Pod vertex 3 the deployment name and type of deployment (StatefulSet, DaemonSet or Deployment) can be found. All this data can be retrieved using the Cypher query shown in Listing 4.1 by providing the `execution_id` of an previous deployment.

```
MATCH (e:Execution) WHERE ID(e) = execution_id
MATCH (e) -[:HasNode]-> (n:Node)
MATCH (n) -[r:Ran]-> (p:Pod)
WITH e, n, COLLECT({name:r.name, generate_name:p.generate_name,
    kind:p.kind, deployment_name:p.deployment_name}) AS pods
RETURN ID(e) AS id, e.start_time.epochMillis AS start_time, e.end
    _time.epochMillis AS end_time, COLLECT({name:n.name, cpu:n.
    cpu, pods:pods, memory:n.memory}) AS nodes",
```

Listing 4.1: Retrieve deployment from knowledge base

4.4 TUNER

The tuner is implemented in Python[5] `3.7.3` using the native requests package (Version `2.22.0`) to perform all API requests. The knowledge base is queried using the neo4j package (Version `1.7.4`) for Python. The tuner runs on the local machine. The following sections will cover the implementation of the tuner using again the process view displayed in the design section Figure 3.2.

4.4.1 *Monitor*

As already discussed in the monitor section of the architecture chapter (Section 3.2) the monitor needs to extract two different types of data, performance data and structural data. To retrieve the performance data cAdvisor[6] (Version `0.32.0`) and Node Exporter[7] (Version `0.18.1`) are used as shown in the physical view (Figure 4.1). cAdvisor monitors the internals of nodes and retrieves metrics concerning pods and containers in these pods. Node Exporter retrieves metrics covering the performance of nodes and their status. cAdvisor is also used for monitoring of Kubernetes itself [19] and therefore is automatically already present in every pod. The Node Exporter has to be present on every node and by using a daemon set this is ensured. Both monitors are industry standards and have shown to work very well with the chosen performance metrics database Prometheus. Prometheus[8] (Version `2.2.1`) is a time series database. The data from both sources are directly piped to Prometheus for storage.

5 https://www.python.org/
6 https://github.com/google/cadvisor
7 https://github.com/prometheus/node_exporter
8 https://prometheus.io/

To retrieve the actual structure of the current deployment the Kubernetes web API is queried using GET requests. The received data is then prepared for usage in the system by storing it in a standardised dictionary structure. Initially the tuner system was build to use the Kubernetes API[9] specially built for Python, but this proved unreliable and would fail after a random amount of time for no apparent reason.

Monitoring stability

The stability of the system is verified by retrieving the current deployment from the Kubernetes API and comparing this to the deployment at the start of the time window. The deployment at the start of the time window is retrieved from Prometheus.

Extract current load

The load extraction is project dependent. The chosen load value retrieved should represent the size of the current deployment, for example when the deployment scales up due to an increase in requests, the retrieved load value should also increase.

To explain the load extraction the testing application that is introduced in Section 4.5 is used as an example. Testing is performed with a PHP-Apache container that performs an action for each received request. The load of this application therefore increases when the amount of requests increase. To get an insight in this relationship the test application was put under a load pattern where each two minutes the amount of requests per second increased with two requests. This pattern started at 2 requests per second and continued till 40 requests per second. Figure 4.3 shows the found relation between requests per seconds and the amount of scaling present that is required to handle the incoming load.

The next step is to find a metric that shows the same behaviour as the scaling when the amount of requests changes. For our use-case it was decided that the amount of bytes received over the network might be a good fit as every request received by our service would be of the same size. The verify this assumption the relation between bytes received and requests per seconds was analysed, using the same load pattern as before. The resulting plot can be seen in Figure 4.4. Comparing this to Figure 4.3, it shows that both the scaling and the bytes received show a linear growth when the amount of requests per second increases. Therefore bytes received can be used as an indicator to establish the amount of scaling required in this use-case.

9 https://github.com/kubernetes-client/python

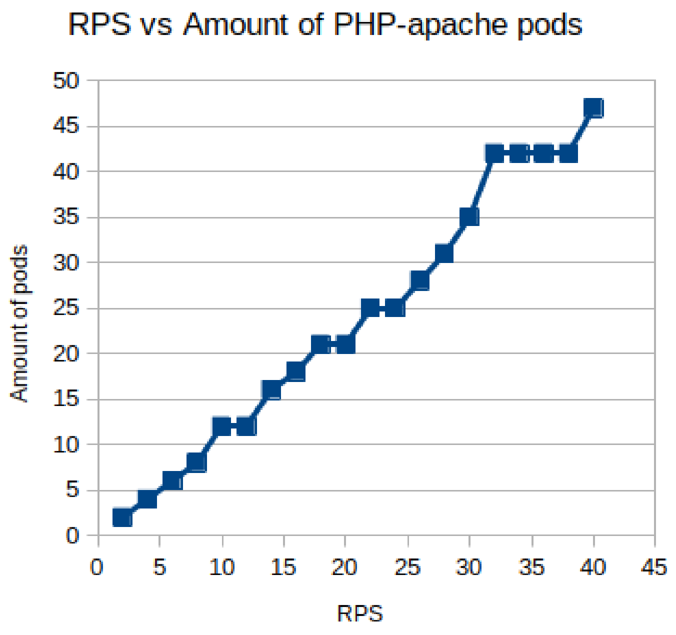

Figure 4.3: Increase of PHP-Apache pods with increasing rps

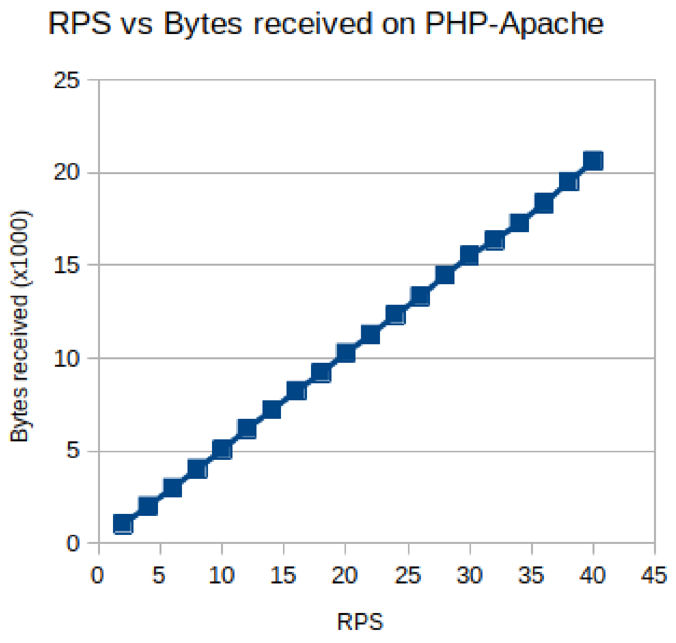

Figure 4.4: Load extraction measure Bytes received

4.4.2 *Analyse*

Calculate current cost

To calculate the current cost the price of a core and of a gigabyte of memory is given in the run settings. The current deployment is retrieved from Kubernetes and the resources present in every node are extracted. The resources are multiplied by their given price as can be seen in function `calc_cost` in Listing 4.2 and the resulting cost is found.

```python
def calc_cost(nodes, settings):
    """
    Calculates the operational cost of running the given nodes
        using the prices provided in the settings.
    """
    cost = 0.0
    for node_info in nodes.values():
        cost += node_info["cpu"] * settings["price_per_core"]
        cost += node_info["memory"] * settings["price_per_gb"]
    return cost
```

Listing 4.2: Calculate current cost

The current cost case has a chance to perform a random migration to ensure that the deployment does not get stuck in a local minimum. The probability for this migration is set in the settings and is set to 0.5. The random migration is performed by first selecting a pod. A check is performed to verify that the pod can be moved, and therefore is not stateful. If this is the case a new random pod is selected. A random destination node is selected and the available CPU space is checked to verify that the pod fits on the node. If the pod fits a random migration has been found. If it does not fit the whole process restarted by selecting first a new random pod and then a destination. The tuner performs 1000 attempts to find a suitable pod to migrate. The 1000 was set because the probability of covering all interesting cases is very high. The tuner does not perform an exhaustive search as there are $num_{pod} * (num_{node} - 1)$ possibilities and with large systems this can grow very large. If this has failed for 1000 times the tuner concludes that no random migration can be performed as all the nodes are already totally filled.

Calculate cost node removal

To find the running cost of the deployment where one node is removed the tuner first has to find if and which node can be removed. The code performing this analysis can be found in the Appendix on page 65 in Listing A.1. Finding the best option for node removal is initiated by calling `empty_node_transitions`. This function first retrieves the current state of the system. The nodes that can not be removed are filtered by `select_removable_nodes`, it looks at every node and checks that it does not contain pods that can not be moved. After filtering all the nodes a check is performed to verify that there are still potential nodes remaining that can be removed. `least_transitions_removable` counts the pods that have to be moved for the remaining nodes, daemonset pods are ignored in this count. The node that requires the least amount of movements to be emptied is selected.

The next step is to find a new node for every movable pod present on the node that is selected to be removed. To find all possible resulting deployments the function `recursive_find_new_distributions` iterates through all possibilities using a depth first search. Each iteration one pod from the `reschedule_pods` is assigned to one of the remaining nodes if possible. Every time all pods have a new assigned location the found deployment gets added to the list of possible deployments. If the list containing possible resulting deployments is empty then the pods can not be rescheduled and the node can not be removed. If the list has one or more entries the node can be removed as all its pods can find a new place on a different node.

To select between multiple found deployments function `select_lowest_max_requested` selects the one with the most even spread of requested CPU. This is implemented by retrieving the node with the largest CPU requested value from each possible deployment and comparing them. The lowest is selected. This should allow the system to be more resilient to a sudden increase in load.

The found deployment still has to be put in the standardised format that the planner step prefers. This is performed by `change_selected_distribution_into_transitions`. Specific information of this format can be found in the plan step.

The resulting operational cost of the node removal is calculated using `calc_removal_resulting_cost` provided in Listing 4.3. Here the current state of the system has to be provided together with the name of the node that has to be removed. This function removes the appropriate node and then uses the cost calculation function given in Listing 4.2.

```python
def calc_removal_resulting_cost(nodes, node_removed, settings):
    copy_nodes = copy.deepcopy(nodes)
    del copy_nodes[node_removed]
    return calc_cost(copy_nodes, settings)
```

Listing 4.3: Calculate cost node removal

Calculate cost cheapest previous deployment

Using the current load on the deployment the knowledge database is queried to retrieve all applicable deployments. The query can be seen in Listing 4.4 and returns the used resources and the id of the execution. The user can set the delta load difference in the settings, this is used to specify the load offset allowed for it to still be retrieved. For example if the deployment is currently receiving a load of 3000 and the load delta is 0.10 then all previous deployments are selected that have a load between $3000 * (1 - 0.10) = 2700$ and $3000 * (1 + 0.10) = 3300$.

```
MATCH (e:Execution) -[:HasNode]-> (n:Node)
WHERE $load_min <= e.load <= $load_max
RETURN ID(e) AS execution_id, COLLECT({name:n.name, cpu:n.cpu,
    memory:n.memory}) AS nodes
```

Listing 4.4: Deployment retrieval query

The next step is to select the cheapest option from the retrieved previous deployments. Every previous deployment contains the nodes and the resources those ran with, therefore the cost can easily be calculated. When multiple previous deployments are the cheapest, the one with the on average lowest CPU usage is selected. This is based on the same argument as used for the node removal: adding to resilience to load increase.

Now the cost of the desired deployment is found and the total deployment is retrieved from the knowledge base by providing the `execution_id` to the query in Listing 4.1. Verification is needed to ensure that the current deployment can be modified to acquire the desired deployment. To perform this verification each node present in the current deployment has to be matched to a node in the desired deployment.

The matching is performed by code that can be found in the Appendix on page 69 in Listing A.2. The matching depends on a scoring system that distinguishes removable pods and pods that can not be removed as can be seen in function `scoring`. The `get_scores` function iterates over all possible combinations of current and desired nodes and calculates for each combination the resulting scoring using function `calc_score_per_node`. This function iterates over all the pods present in the current node and compares these to the destination node. Every matching pod provides an increase in score and every mismatch decreases the score.

When all the possible node combinations have their appropriate score it is time to find the best mapping from current nodes to desired nodes. This is performed by an recursive iteration over all possible mapping combinations which each time returns the best found option for its iteration. When the last step of this recursion returns, the result is a mapping where all the current nodes are mapped to desired ones. This is performed by `rec_find_highest_score`.

The next step is to analyse our found best mapping and verify that pods that can not be moved are supposed to move in this mapping. This is performed by `valid_transition` by iterating over all the required changes and inspecting the provided info of the pod.

When the mapping has been verified it still has to be put into a standardised format for the planner. This is performed by the second part of `find_transitions_execution_change` using a loop over all the required changes.

The cost of this cheapest previous deployment was already calculated in the first step when it was compared to other deployments and this cost is used as its resulting price.

Select cheapest

This component simply compares the three found costs and selects the cheapest. The if statement present in the pseudo code of Algorithm 1 in the analysis section (Section 3.2) is implemented. The resulting transitions required to reach the cheapest deployment are returned and can be passed to the planning step.

4.4.3 *Plan*

Find most efficient transition

The plan step receives all the changes that have to be performed per node. These changes include which pods have to be added and removed from the node and if the node has to be terminated. The goal of the planning step is to perform these changes with the most migrations possible.

The first step in finding the most migrations possible is to extract all possible migration combinations. The code performing this first step can be found in the appendix on page 74 in Listing A.3. The extraction of possible migration is performed using recursion performed by function `recurse_find_all_migrations_sets`. This function removes the first add action in the transition it can find. If no add action could be found and therefore is empty the base case has been found and the whole transition has been processed and empty migration set can be returned. If the add action is not empty it is used to find all possible migration by matching it to pods listed in the `removes` of nodes using `find_suitable_migration_sets`. The loop is then repeated with one add action less resulting eventually in the base case. `find_all_migrations_sets` initiates this recursive loop and sorts the resulting migration sets on decreasing length. This ensures that the largest possible number of migrations will always be evaluated first.

The next step is to find an order to perform the largest possible migration set. This process starts with `find_suitable_migrations` which can be found together with the other functions discussed in the appendix on page 76 in Listing A.4. It takes the first entry of the generated migration sets and checks if there is an order in which all the given migrations could be executed. When the check fails the next migration set is selected from the provided list and the check is performed again until it succeeds or the list becomes empty.

The check is performed using the recursive function called `recursive_construction_migration_order`. The base case of this function is receiving an empty list, then every migration present in the initial migration set could be executed. When the function receives a non empty list it will select the first migration in the list and simulate performing the migration using `helper_recursive_construction_migration_order`. This helper function then calls `recursive_construction_migration_order` again but then with a deployment where the selected migration from the previous iteration is performed. This process is repeated until all migrations are performed, reaching the base case. A possible migration order is then found. If the list is not yet empty but the rest of the migrations can not be executed anymore the recursive function will backtrack and try a new branch of the search tree. The actual simulation of the migration to modify the provided deployment is performed by `simulate_migration` and the deployment is afterwards verified using `verify_deployment` to ensure no node holds more pods than its capacity.

The largest possible migration is now know, indicating that any other changes in the given transition have to be performed by add and remove actions. These are all extracted by removing the found migrations from the provided transitions. All the actions including the node removals are prepared to be passed to the execution step.

4.4.4 *Execute*

The current standard when building a cluster with Kubernetes is to use deployments. A deployment specifies how many pods should run of a specific pod description; more information concerning Kubernetes deployments can be found in Section 2.1. Due to the specified use of deployments a newly created pod can not simply be assigned to its destination node. The system has to be massaged in scheduling the pod on the desired node. To achieve this node affinity [20] is used. Each deployment gets an unique label added to its description of the pods. When there is a node having that same label the Kubernetes system will put preference on placing the new pod on the labelled node. This does mean that the system is only able to move pods that are added by the cluster owner, pods that are present to ensure the functioning of Kubernetes itself can not be labelled and therefore can not be scheduled.

Node affinity prefers the presence of a label, but does not require it. The weak binding was deliberately chosen, as it does not effect the behaviour of the deployment when a label is not present. This allows Kubernetes to still perform its normal scaling without effects. Because Kubernetes is able to ignore the weak binding, it is uncertain that the pod is scheduled on the correct node. Therefore an verification is added to every action to verify that the preferred action was performed as expected. When this is not the case, the whole transition is stopped.

All the actions described in the sections below are enforced using command-line calls performed using the subprocess package in Python. The commands are all performed using kubectl[10] version 1.14.

Execute transition

This section will cover all four actions that the execute component can perform, starting with pod deletion.

Before the pod can be deleted the initial deployment has to be stored, this is later used to verify that the deletion was successful. After the initial state has been stored the pod is deleted. The deployment scale of the pod type is decreased by one to ensure that no new pod gets scheduled. The stored initial deployment is compared to the current deployment and the deletion is verified. If the deletion failed, for example because a different pod was deleted an exception is raised. This also stops further transitions that were still planned. The following enumeration lists all the steps performed for a pod deletion.

1. Delete the designated pod
2. Reduce the deployment scale of the given pod type by one
3. Verify that the correct pod is deleted

The migration action does not change the deployment scale, but it uses it to its advantage. As with every action first the initial deployment is stored. Then the destination node of the migrated pod is labelled with the label of the deployment. The pod is deleted from its original node, and the deployment will therefore automatically create a new pod to meet required number of pods. The applied node-label will give this new pod preference to the labelled node. A verification check is ran to verify that the new pod was actually assigned to the correct node. When the movement has been verified the label is removed from the node. Even if the verification fails and an exception is raised the system still removes the label from the node.

1. Label the destination node using the pod type
2. Delete the designated pod
3. The pod should be scheduled on the labeled node
4. Verify if the pod is scheduled on the correct node
5. Remove the label from the destination node

10 https://kubernetes.io/docs/reference/kubectl/kubectl/

The add action is very similar to the migration action. First it stores the initial deployment for verification. Then it labels the destination node with the corresponding label of the deployment of the pod that has to be moved. Now the add action deviates from the migration, because instead of deleting a pod, the deployments required number of pods is simply increased by one. This causes the deployment to create a new pod, which again has preference for a node with the deployment label. The addition on the correct node is verified by comparing it to the initial state and after success of failure the label is removed again from the node.

1. Label the destination node using the pod type
2. Increase deployment scale of the given pod type by one
3. The pod should be scheduled on the label node
4. Verify if the pod is scheduled on the correct node
5. Remove the label from the destination node

The node removal action is fairly simple, as the node should already be totally emptied. The first step of removing a node is to drain it. This is redundant as it is already confirmed that it does not contain any pods, but ensures that no new pods get scheduled on the node. When the node is drained successfully it can be deleted using the delete command. This command is passed with the `-ignore-daemonsets` flag, to allow the removal while there are still daemonsets present. These daemonsets are only present for the monitoring and killing them will not affect performance. The next step is to verify that the node is actually deleted, when this has been verified the next node can be removed.

1. Drain the given node
2. Remove the node
3. Verify that the node has been removed

4.4.5 *Running the tuner*

Before the tuner can be started the Neo4j container must be running and the cluster including Prometheus is available and can be reached from the machine that is running the tuner. Also the local Kubectl, command line interface with Kubernetes, must be linked to the appropriate cluster as the tuner uses this to perform its actions. The Kubernetes API should also be reachable by the tuner. Lastly, ensure that the addresses of all the previously mentioned components are correctly set in the `settings.json`.

The tuner can be started using `Python3 TopologyGenerator/main.py <Path to settings.json>`. An additional logging flag `-l <custom log path>` can be passed to provide a custom destination for the log file.

4.5 TESTING

Most of the components are tested using unit-testing these test-cases can be found in the Github repository[11]. Some components that interact with Kubernetes can not be tested using unit-testing and are therefore covered in the following sections.

4.5.1 *Test application: PHP-Apache*

To develop all the required components and to verify that they actually work a testing application is needed. The decision was made to opt for the simplest testing application possible to ease the development. Therefore the PHP-Apache deployment was chosen. The application consists of only one microservice. The microservice consists of a pod that runs a Apache server on which PHP code is running. The PHP code performs a million square root calculations for every request it receives, therefore generating CPU load. The deployment files can be found on the Github repository[12]

4.5.2 *Visualisation*

To visualise the state of the deployment Grafana[13] (Version `5.3.4`) is used. This is a visualisation tool that is specialised in creating online dashboards filled with plots and other metrics. It can use Prometheus as a data source and perform API calls retrieving real time data. The created dashboard used for visualising the state of the deployment can be retrieved from the Github repository[14]. The plots used in the testing are all extracted from this dashboard.

4.5.3 *Execute actions*

All the specific pod and node names are shortened to their unique part of the name to make the testing more readable.

11 https://github.com/Gezzellig/DynamicTopologySelection/tree/master/TopologyGenerator/SmartKubernetesSchedular/tests
12 https://github.com/Gezzellig/DynamicTopologySelection/tree/master/demo
13 https://grafana.com/
14 https://github.com/Gezzellig/DynamicTopologySelection/blob/master/visualisation/grafana_dashboard.json

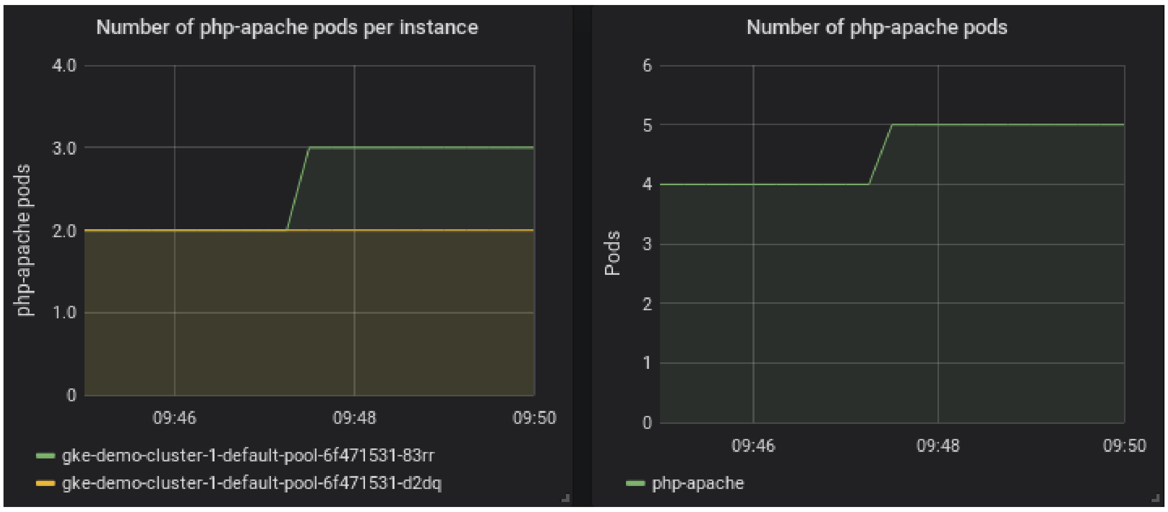

Figure 4.5: Add pod

Add pod

To verify that a pod can be added to the deployment the pod addition was run. As can be seen in Figure 4.5 the initial state contained four pods present in the deployment. Both node `83rr` and `d2dq` contain two pods.

The add pod part of the tuner is executed to add podtype `php-apache` to node `83rr`.

As can be seen in the same figure, the amount of pods increases from four to five, and the new php-apache pod is scheduled on node `83rr`. More detailed output of the test can be found in the Appendix on page 79. The pod has been successfully added to the node, therefore the test succeeded.

Migrate pod

To verify that the tuner can migrate a pod a pod migration was executed to move pod `h6fkb`, being part of a php-apache deployment, from node `83rr` to node `d2dq`. Figure 4.6 shows the results of the migration. The initial deployment consisted of four php-apache pods. Three of these are present on node `83rr` and one is on node `d3dq`

The migration succeeded. This can also be seen in the figure, the amount of php-apache pods stays constant but now both nodes contain two of those pods. More detailed output of the test can be found in the Appendix on page 79. The pod was successfully migrated.

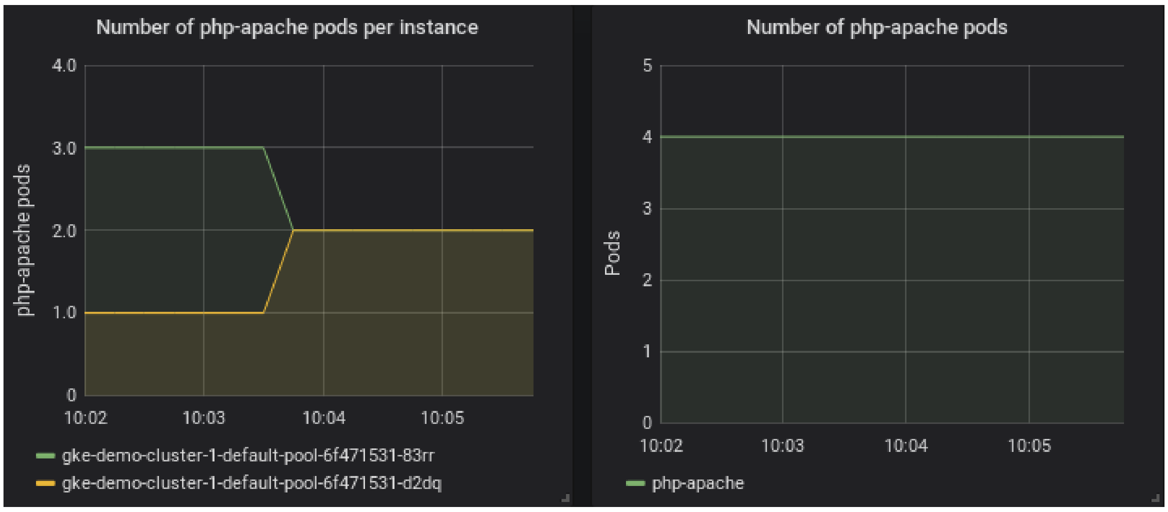

Figure 4.6: Migrate pod

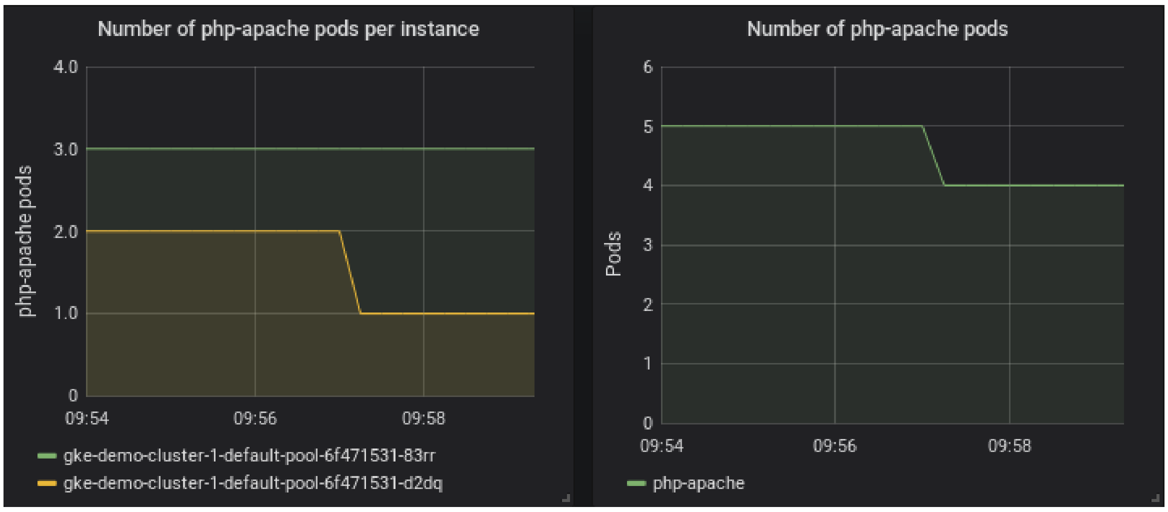

Figure 4.7: Remove pod

Remove pod

To verify that the tuner can delete a pod from the deployment, the pod removal to remove pod `bmtcb` was run. Figure 4.7 shows the whole test. Initially the system was running five php-apache pods as can be seen in the graph on the right. node `83rr` contained three of those pods and node `d3dq` had two.

Pod `bmtcb`, which is part of the php-apache deployment, was situated on node `d3dq` was removed using the removal part of the tuner.

As can be seen in the figure the total amount of php-apache pods decreases and the total amount of pods on node `d3dq` also decreases. More detailed output of the test can be found in the Appendix on page 80. The pod was succesfully removed from the deployment.

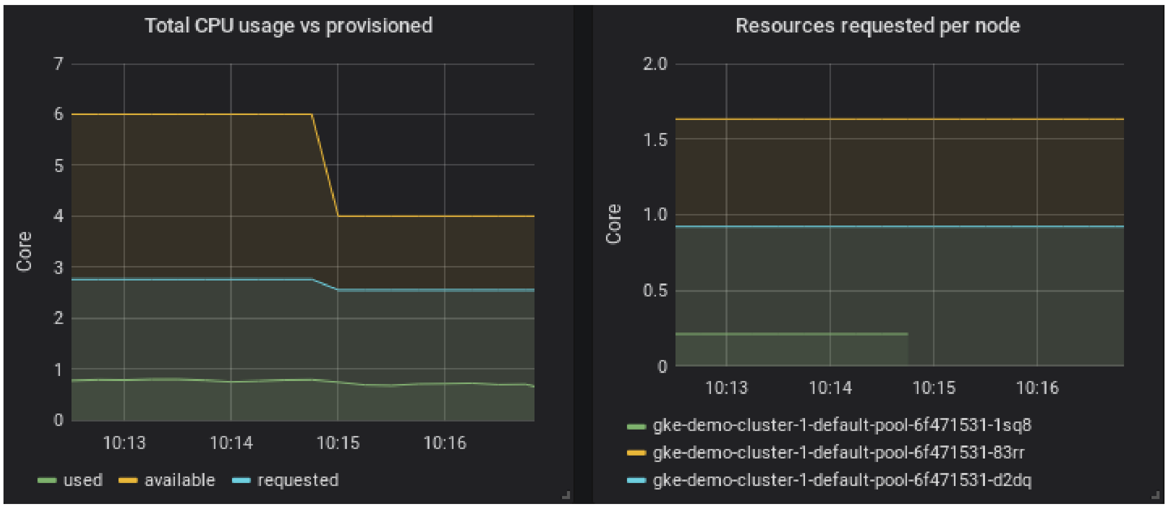

Figure 4.8: Remove node

Remove node

To verify that the tuner can remove a node from the deployment, the node removal was ran to remove node `1sq8`. Figure 4.8 shows the whole test of node removal. The left graph displays the total sum of available, requested and used CPU cores in the system. The right shows the requested amount for every separate node. Initially the system was running three nodes.

Node `1sq8` is removed using the node removal part of the tuner. This node contained only daemonset pods. These are required for Kuberenetes and the monitoring. These are the same daemonset pods as would be present when the tuner is running and is removing a node. These daemonset pods request in total a 0.211 from the CPU.

As can be seen in the left graph of the figure the total amount of cores present in the deployment decreases by two. The graph on the right shows that the line representing the amount requested for node `1sq8` stops. More detailed output of the test can be found in the Appendix on page 80. The node was successfully removed.

5 EVALUATION

To evaluate the ability of the tuner to reduce operational cost it needs to run in a controlled environment. Section 5.1 will explain the setup used for the evaluation and provides all the necessary information to reproduce the results. Section 5.2 will show these results and analyse them to evaluate the performance of the tuner.

5.1 EXPERIMENTAL SETUP

5.1.1 *Benchmark application*

A lot of effort was spend on finding a proper scalable benchmark application running in Kubernetes on which we could generate load. Preferably this application would be one consisting of many services that allow scaling to simulate a proper microservice architecture under load. After multiple failed attempts trying the Robotshop[1], the Sockshop[2] and Gitlab[3] eventually the decision was made to use the `PHP-Apache` application introduced in the testing section (Section 4.5). This application only consists of a single pod, but it is scalable and allows for easy load generation. Therefore it does allow the tuner to show that it can save cost, however the advantages acquired by moving different type of containers around can not be evaluated.

An autoscaler was applied on the `PHP-Apache` deployment to enable its scaling. The autoscaler tries to keep the total percentage of CPU usage of `PHP-Apache` pods on 80 percent of the requested value. The requested value for an `PHP-Apache` pod is 0.2 core. The autoscaler will scale the deployment between 1 and 50 instances. The scaling in the following evaluation will never come close to 50, therefore the system behaves like there is no upper bound. The deployment file of the autoscaler can be found on the Github repository[4].

5.1.2 *Load generator*

To generate load on the deployment a load generator is needed. Artillery[5] (Version `1.6.0-28` running on Node.js Version `10.15.2`) was chosen as it is able to generate load on API endpoints using web

1 https://github.com/instana/robot-shop
2 https://microservices-demo.github.io/
3 https://gitlab.com
4 https://github.com/Gezzellig/DynamicTopologySelection/blob/master/demo/php-apache-hpa.yaml
5 https://artillery.io/

requests. Artillery allows the user to provide a load pattern using a YAML file. This allows an easy and robust way of running the same load pattern multiple times. It keeps track of every request and reports on its response time and success or failure.

5.1.3 *Hardware*

The hardware is the same as used in the testing section (Section 4.5). The Kubernetes cluster is running on Google Kubernetes Engine. The virtual machines used contain 2 virtual CPU cores (Intel Xeon Scalable Processor running with a 2.0 GHz base frequency) and 7.5 Gigabytes of virtual Random Access Memory. Automatic node scaling is used to allow the system to dynamically react to increase and decrease of required resources. The node scaling was set to a minimum of one and a maximum of eight nodes. As all experiments run in this evaluation use seven nodes maximum this should behave like there was no upper limit. While receiving no load on the `PHP-Apache` and therefore only having one pod the system needed two nodes to place all the support pods for Prometheus and Grafana. The system runs with Kubernetes version `1.11.8` and the nodes are located in zone `europe-west3-c`.

The tuner and knowledge base are located on the local machine which is an HP ZBOOK Studio G5. Further specifications can be found in the testing section (Section 4.5).

The load generator is also located on the local machine. The load generation and the tuner both require few resources and as there are plenty available, this should not effect the results.

5.1.4 *Experiment*

Multiple interesting load patterns will be ran to perform the evaluation. These load patterns will be introduced in the Results section (Section 5.2). To circumvent explaining the processing of the patterns multiple times this section will explain the processing and the sources of the data used.

Per interesting load pattern the whole pattern will be run multiple times with and without the tuner enabled. A single pattern is run multiple times to ensure that the results are representative. The found operational cost can then be compared between running the pattern with and without the tuner, giving an insight in the operational cost the tuner saves.

The cost metrics are retrieved from Prometheus by multiplying the used resources in the used machines by their appropriate cost. The prices used in the evaluation to calculate the operational costs are €0.0280 per CPU core and €0.0038 per Gigabyte of memory. These prices are the current prices for Google Compute Engine [21] and are transferred to euros using the exchange rate of the first of July 2019 where 1 dollar is 0.8859 Euro.

Reducing operational cost is the eventual goal of this research, but it is also very important to establish at what quality cost these reductions are attained. The performance of the deployment will be measured using the response time of every request as it shows how quickly each user could be served. The resulting response times are retrieved from the output file generated by Artillery.

5.2 RESULTS

The evaluation of the tuner was performed in three phases. The first phase was used to get an insight in the performance of tuner. The second phase was used to analyse some anomalies found in the first phase. The third phase is the final evaluation where the load patterns are all set to a comparable load to allow for comparison of performance between the different patterns.

Every load pattern that is used will be explained and visualised in its appropriate phase. For every load pattern ran will start with displaying the results. These results consist of a comparison of operational cost where the same load pattern is run with and without the tuner. The effect on performance is measured using the average latency needed to handle all incoming requests. The analysis section uses the presented results and identifies interesting trends.

5.2.1 *Phase 1*

All the tests shown in phase 1 where ran at least two times and the results shown are the average results of the different runs. The evaluation in phase 1 consists of two different load patterns, one with wide peaks and one with narrow peaks and have the same width of the valleys. These two patterns where chosen to see if the width of the peak has any impact on the performance of the tuner. The load pattern with wide peaks can be seen in Figure 5.1 and the pattern with small peaks can be seen in Figure 5.2. The settings used in this phase can be found in the Github repository[6]

6 https://github.com/Gezzellig/DynamicTopologySelection/blob/master/TopologyGenerator/settings-phase1.json

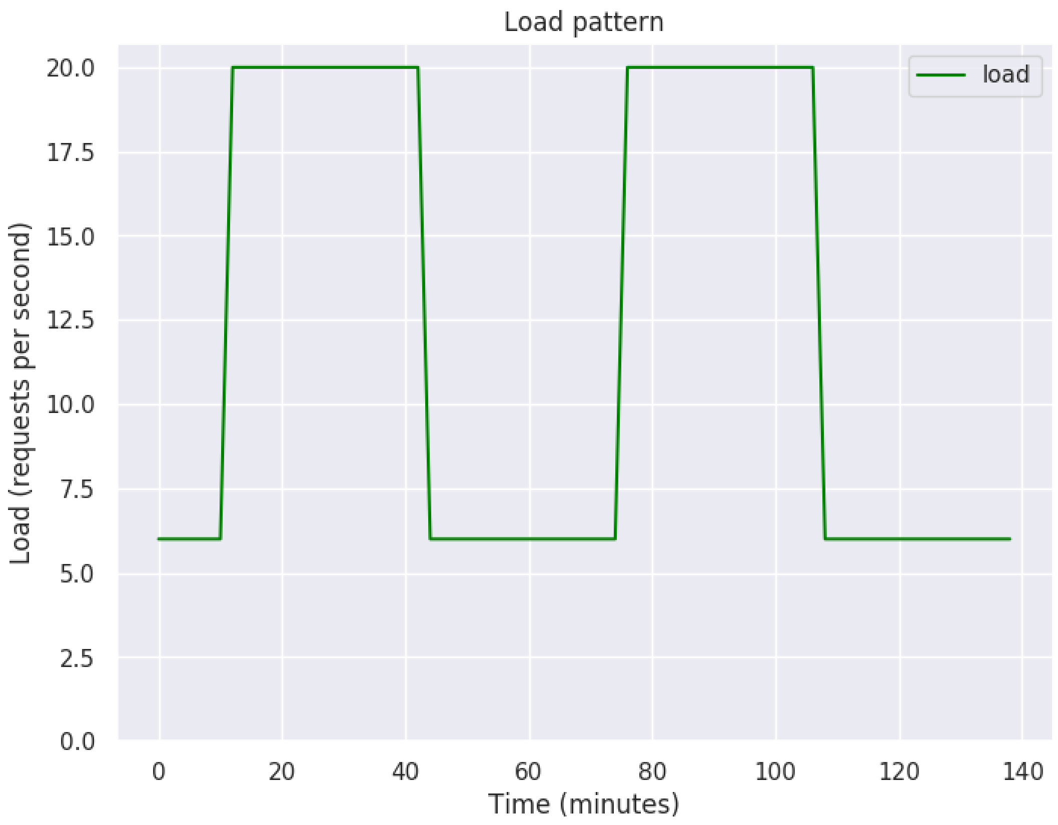

Figure 5.1: Load pattern 1

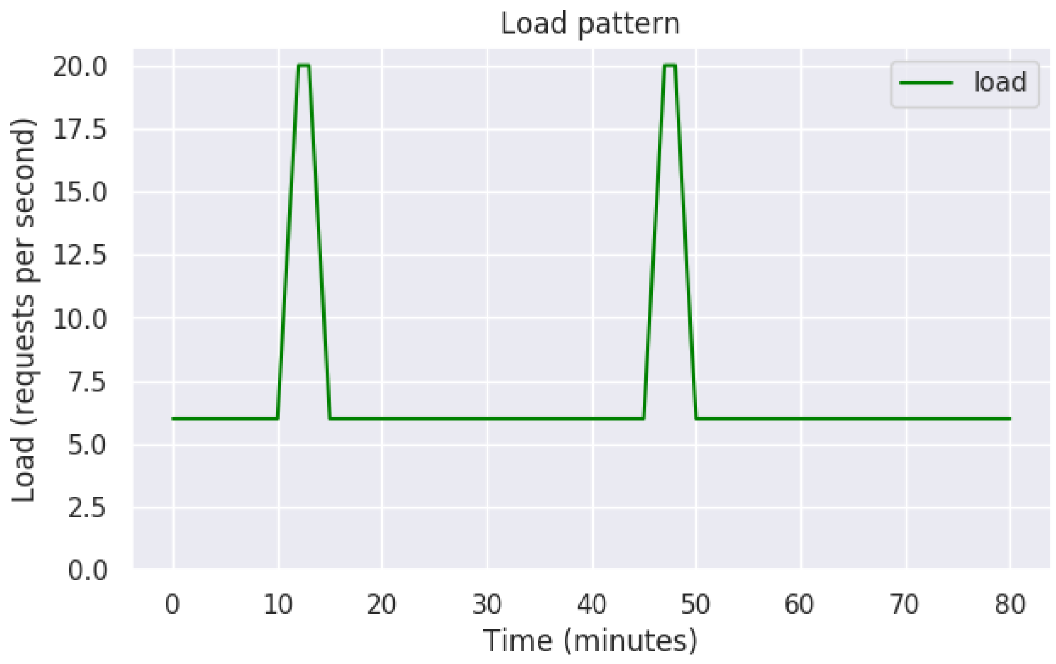

Figure 5.2: Load pattern 2

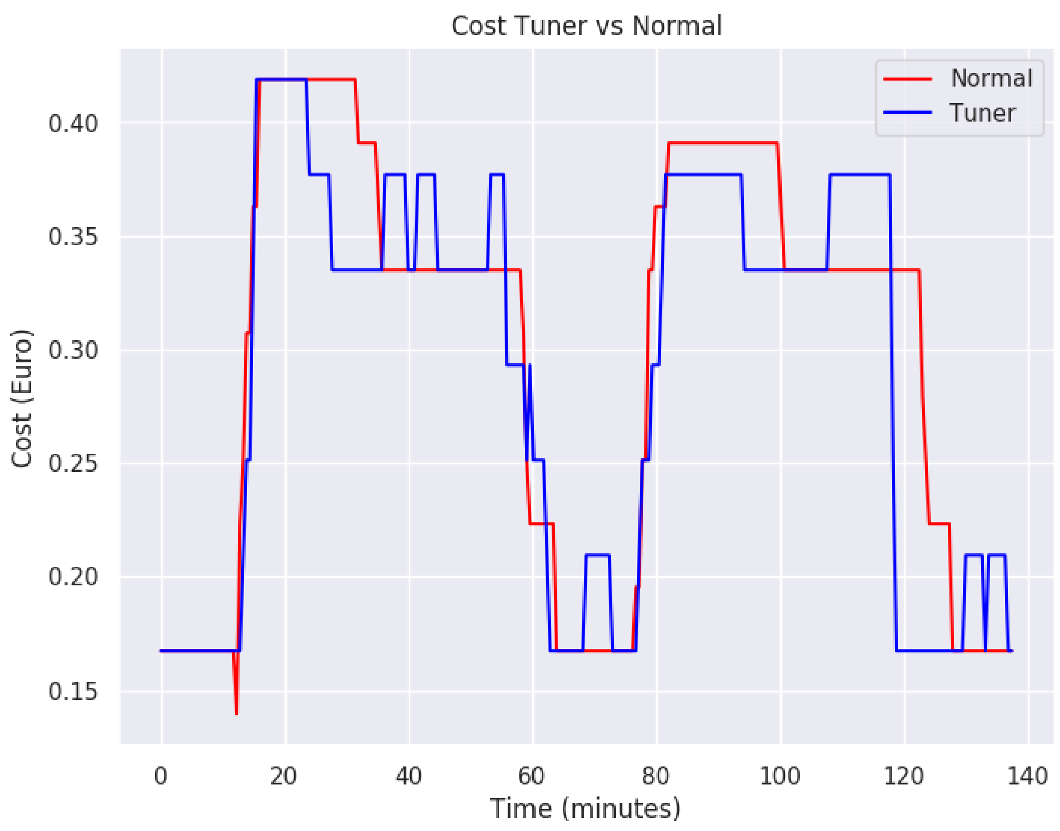

Figure 5.3: Operational cost pattern 1

Load pattern 1: Wide peaks

Figure 5.3 shows the cost per hour needed for running the deployment. Normal means running the system without the tuner active, Tuner are the costs when the tuner is active. The average execution costs for running the load with and without tuner can be seen in Table 5.1. The latency for both the normal runs and runs with tuner can be seen in Figure 5.4. The figure shows for both with and without tuner the median latency for every time step and the 95th percentile. As these results only show two large peaks Figure 5.5 provides the same data but then with a logarithmic scale, to allow better insight in the performance on other moments than the peaks.

Total cost with tuner active	€0.669
Total cost without tuner active	€0.691
Saved by tuner	€0.022
Percentage saved by tuner	3.2%

Table 5.1: Savings pattern 1

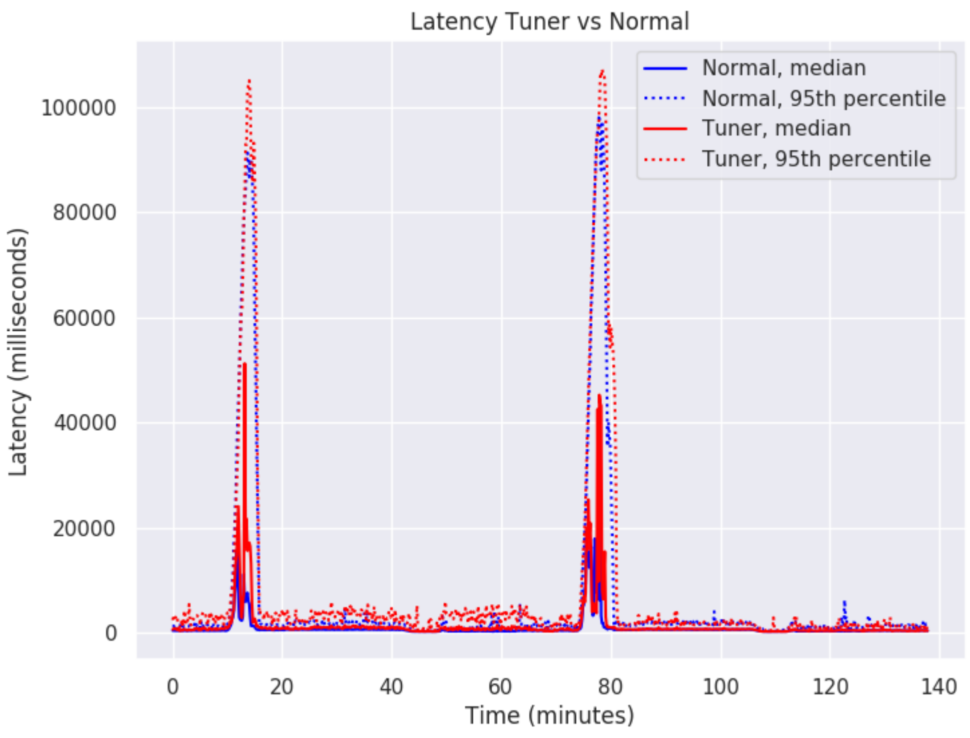

Figure 5.4: Latency pattern 1

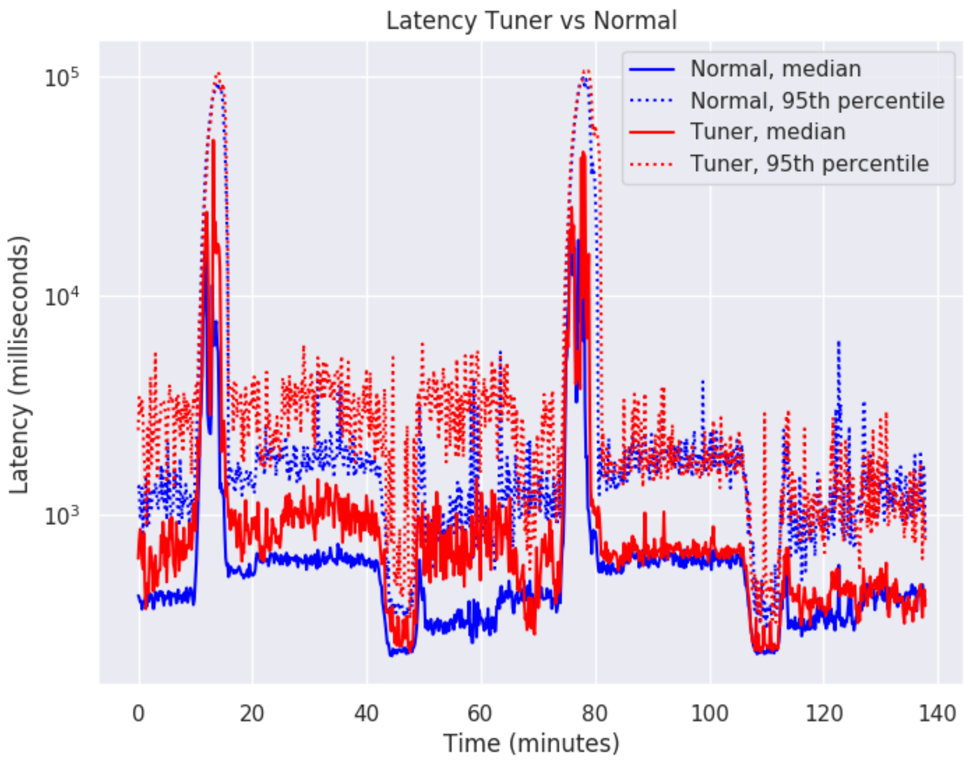

Figure 5.5: Logarithmic latency pattern 1

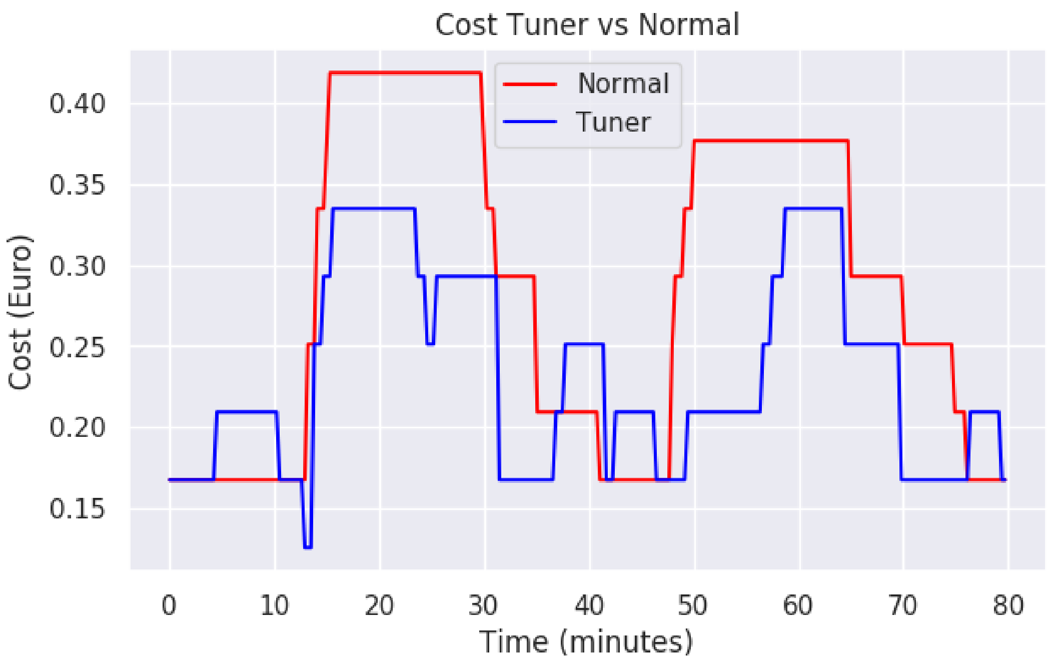

Figure 5.6: Operational cost pattern 2

Load pattern 2: Narrow peaks

Figure 5.6 shows the cost per hour needed for running the deployment. The average execution costs can be found in Table 5.2. The latency for both the normal runs and runs with tuner can be seen in Figure 5.7. The latency on a logarithmic y-axis can be found in Figure 5.8.

Total cost with tuner active	€0.311
Total cost without tuner active	€0.381
Saved by tuner	€0.069
Percentage saved by tuner	18.2%

Table 5.2: Savings pattern 2

Analysis

As can be seen in the results of both load patters their savings tables (Tables 5.1 and 5.2), the tuner reduced operational costs. The performance in both tests did decrease as can be seen in Figures 5.5 and 5.8.

Looking closer into cost over time (Figures 5.3 and 5.6) both patterns show some weird bumps where the tuner is active, indicating that more nodes are requested on those moments. Investigation into the cause of these anomalies is required.

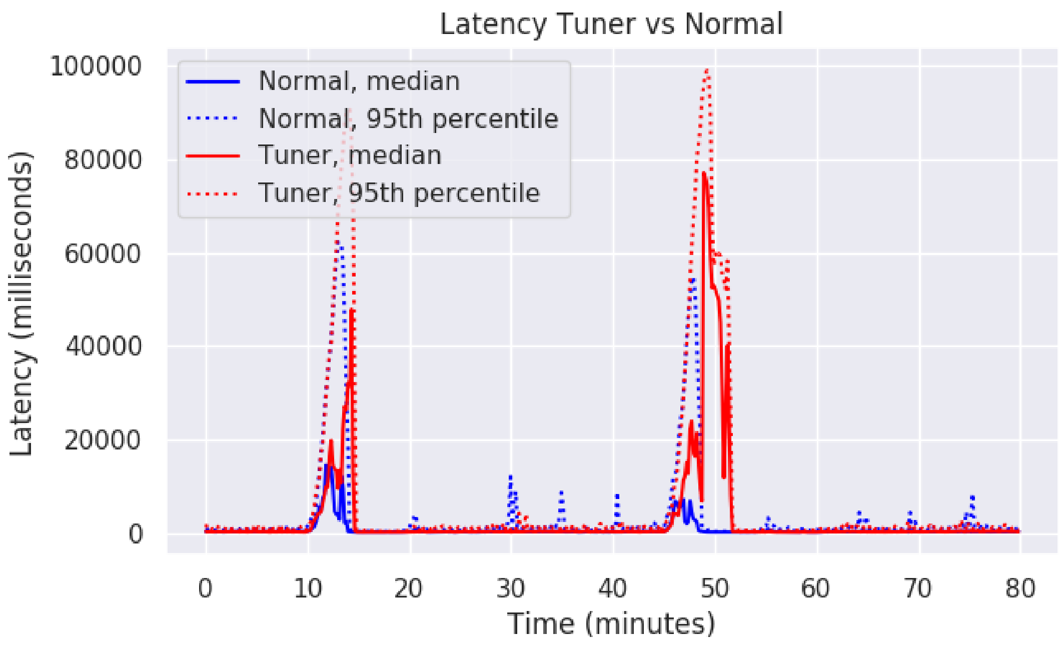

Figure 5.7: Latency pattern 2

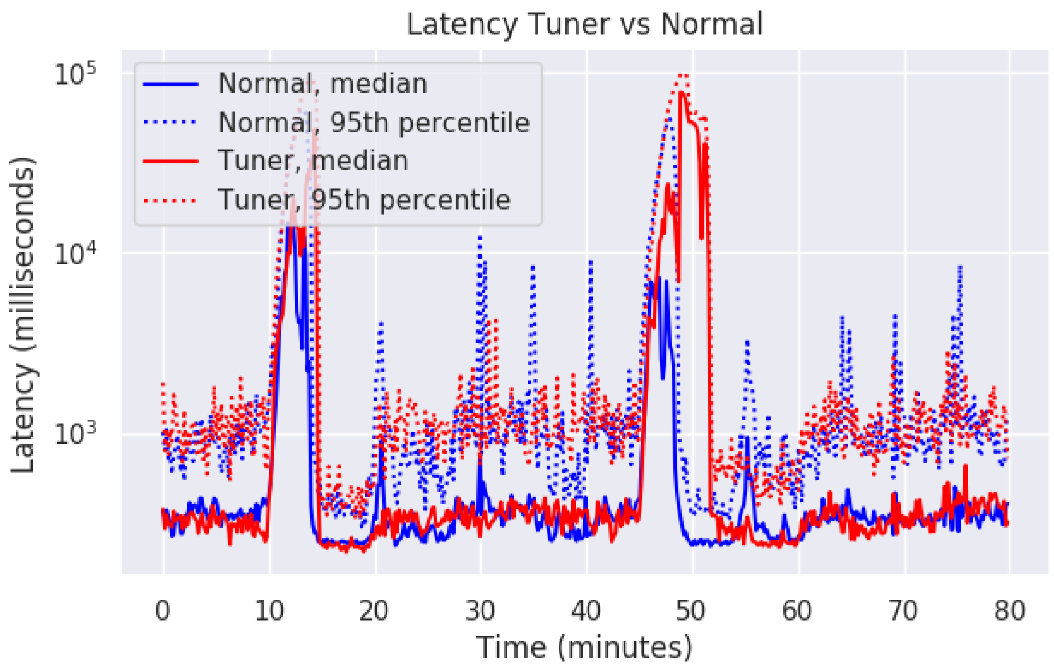

Figure 5.8: Logarithmic latency pattern 2

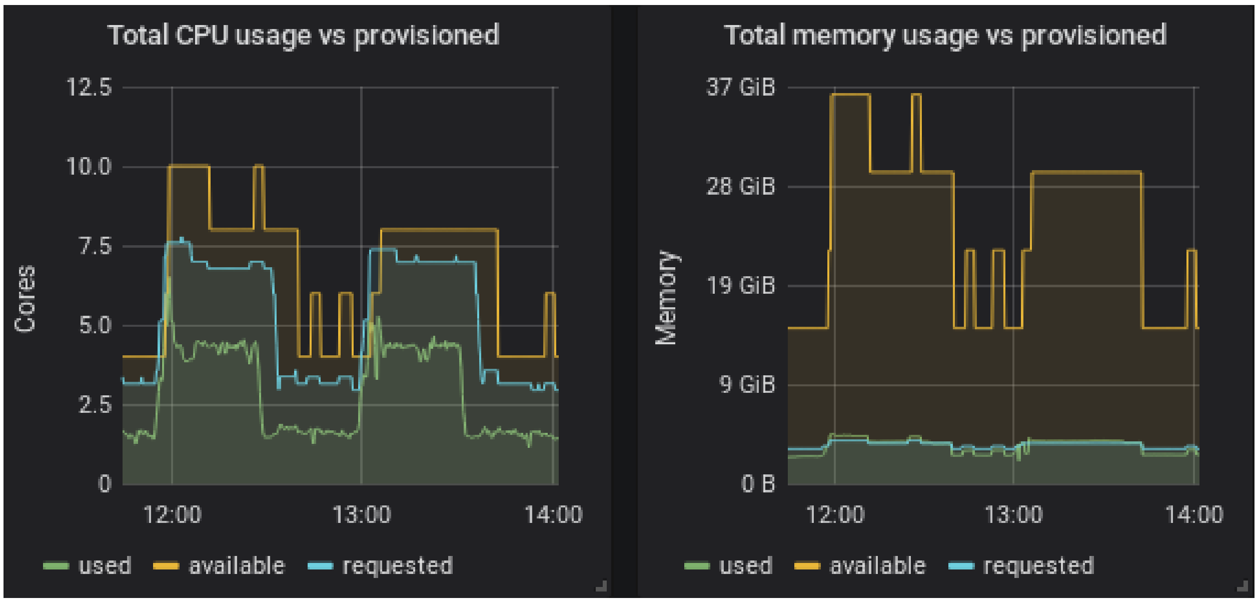

Figure 5.9: CPU and memory utilisation

This increase might be caused by the system lacking resources at those moments. Figure 5.9 shows the total available, requested and used CPU and memory in the deployment for one of the executions for load pattern 1. This data is retrieved from Prometheus and shows resource usage on cluster level. This figure shows that both CPU and memory are abundantly present compared to what is requested and being used. After further investigation it shows no pods are being scheduled on newly added nodes. This behaviour was found over all runs with both load patterns. Therefore it was concluded that it is not caused by resource shortage.

Since we know this behaviour only appears when the tuner is running it is highly likely that the tuner throws the autoscaler off balance due to some actions. After some consideration two possible actions performed by the tuner were identified that might trouble the autoscaler. The first is the random migrations performed when no improvement can be found. As this might appear for the autoscaler as problems with a certain node and therefore it requests a replacement. The second hypothesis is that the active removal of nodes causes some kind of problem, as the autoscaler expects that it is the only one actively changing the deployment. The next phase of the analysis is concerned with investigating these phenomena.

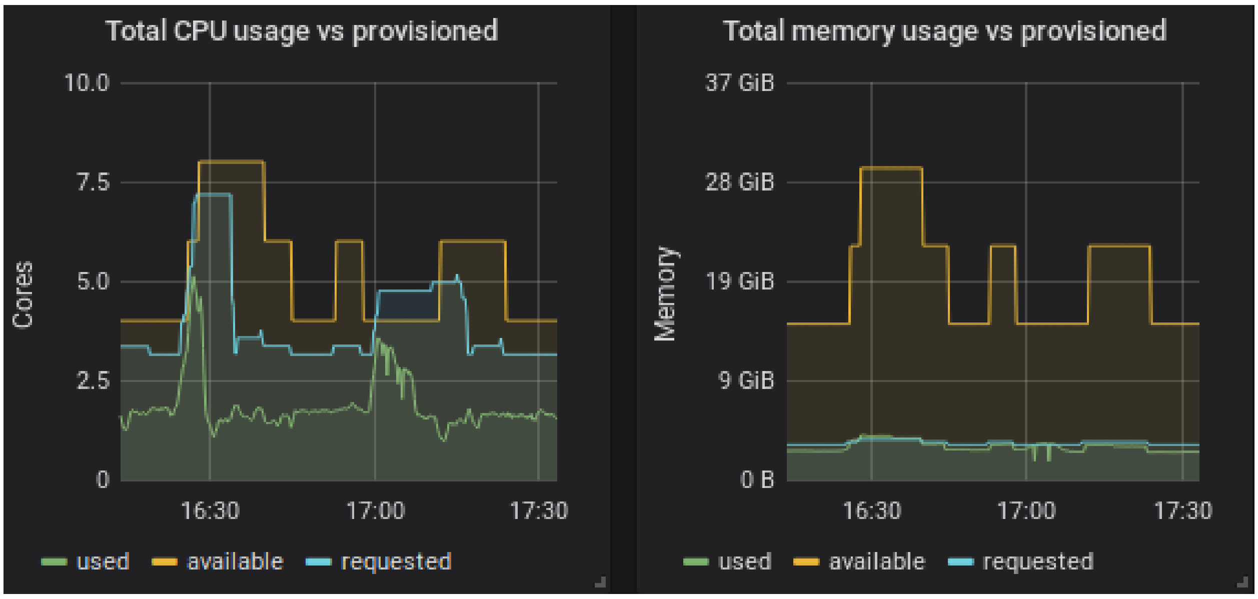

Figure 5.10: Resource utilisation, no random migrations

5.2.2 *Phase 2: Finding the source of node addition*

The first hypothesis was tested using the narrow peaks load pattern.

No random migration

The first hypothesis is that the unexpected scaling of nodes is caused by the random migrations performed when no cost could be saved. Therefore the tuner is run while putting the `migration_chance` in the settings to 0.0, as this will ensure that the random migrations are never performed. Figure 5.10 shows the resulting resource usage in the test. Again the anomalies are still present, indicating that the behaviour is not caused by random migration.

No node removals

To test the second hypothesis and get a clearer view of the problem it was decided that a load pattern with only a single narrow peak would be enough to analyse the problem. The load pattern can be seen in Figure 5.11. The advantages are that each run takes less time and enabling a longer valley to investigate the long term effect.

To verify that the unexpected behaviour also appears with this new load pattern, a control test was ran with the tuner running. Figure 5.12 shows the resource usage of this test. Again the anomalies are present and the used and requested line shows that the scaling is not caused by resource shortages.

The second hypothesis is that the scaling is caused by the hard removal of the nodes which are performed by the tuner. The removal of nodes was turned off by setting the `remove_nodes` value to `false`.

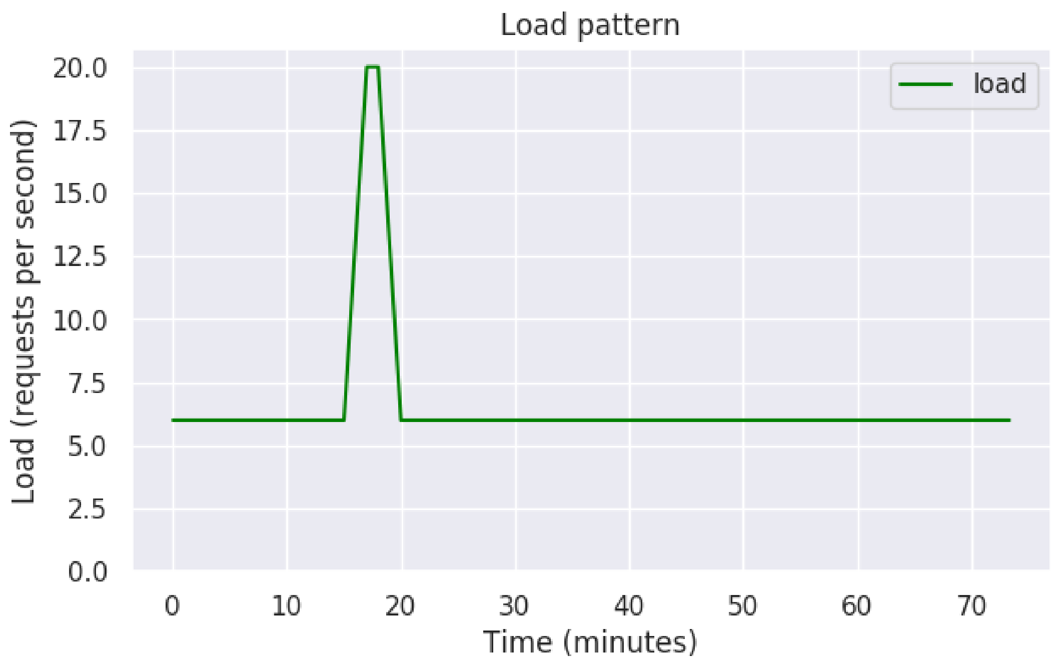

Figure 5.11: Load pattern single narrow peak

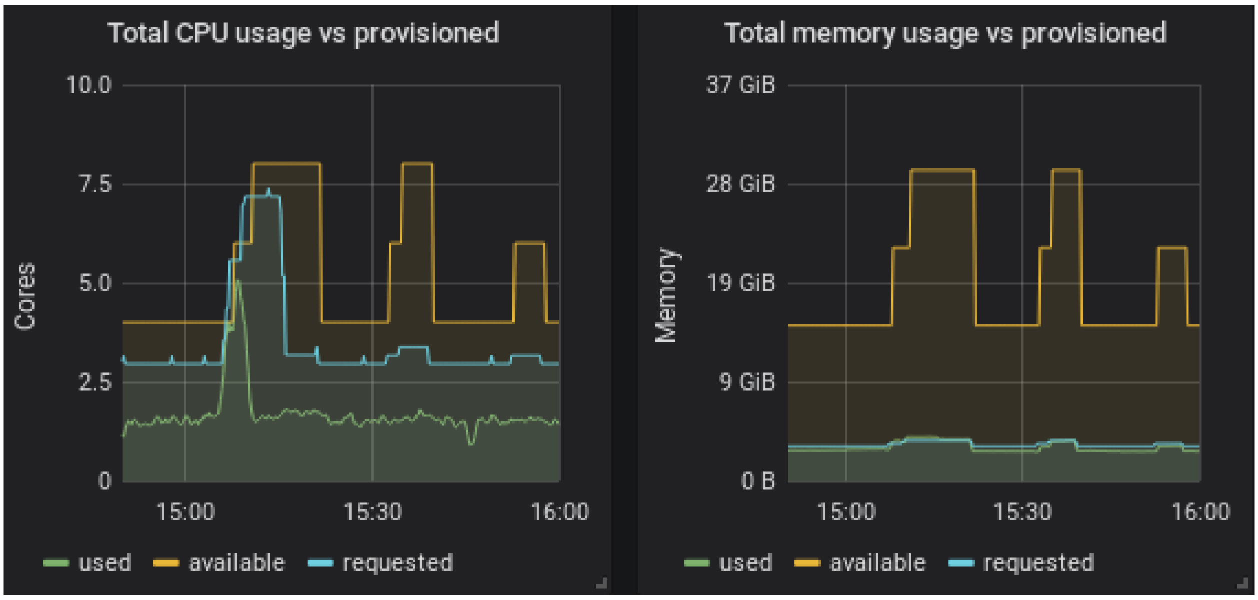

Figure 5.12: CPU and memory utilisation, single narrow peak

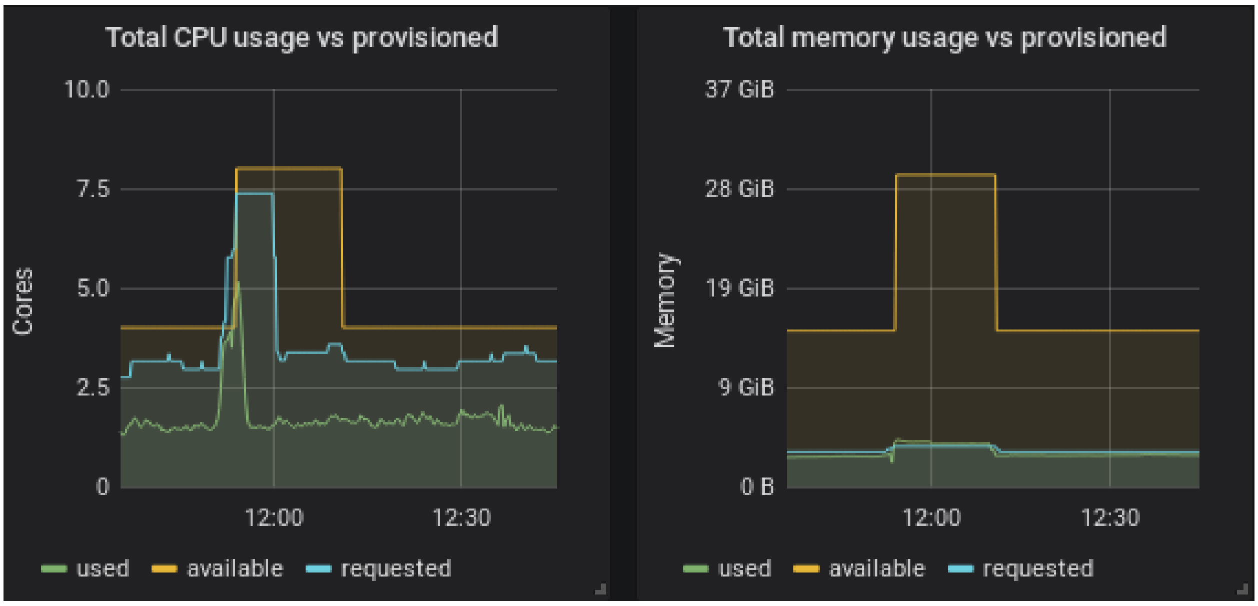

Figure 5.13: Resource utilisation, no node removals

Figure 5.13 shows the resource utilisation of running the single narrow peak pattern without node removal. As shown in this figure the anomalies are not present. Therefore the source of the unexpected scaling is identified.

Figure 5.14 shows the cost for running the deployment with the tuner while node removal is disabled and the cost of running without tuner. Both cost lines in the figure are almost identical. This can also be seen in Table 5.3. The cost for running with and without the tuner are exactly the same.

Total cost with tuner active	€0.252
Total cost without tuner active	€0.252
Saved by tuner	€0.000
Percentage saved by tuner	0.0%

Table 5.3: Savings no node removal

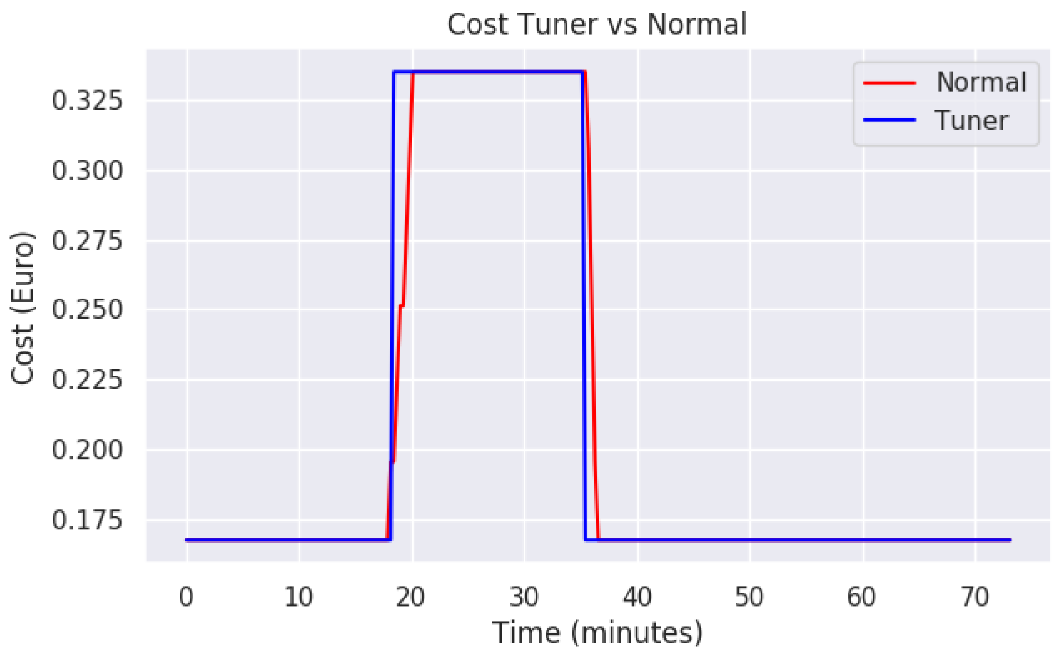

Figure 5.14: Cost no node removal

5.2.3 *Phase 3*

This phase used three different load patterns to analyse the cost-saving performance of the tuner. These three patterns are chosen because together they cover most general load behaviour that can appear on a deployment. The first pattern, shown in Figure 5.15 contains wide peaks and valleys, representing slow change of deployment load. The second pattern (Figure 5.16) has three sharp peaks and long valleys, representing sudden short bursts of load on the system. The third pattern (Figure 5.17) represents a highly variable load where every 10 minutes the load increases or decreases.

Every load pattern lasts 3.5 hours and in total 151200 requests are sent in every pattern. As this is equal for all three patterns the results found can also be compared between them, this gives an insight in which use-cases the tuner is more effective. All the presented results are the average of three runs.

As in phase 1 the found results are compared to running the deployment on the same load without the tuner. In addition to running it just with the tuner, also the results of running the tuner with the node removal is disabled are ran and analysed. All the node removal actions that where supposed to be executed in these runs are logged. These logs are used to simulate the cost reduction of the tuner when Kubernetes would not restart the nodes. A "deleted" node is added back to the cost calculation when pods are assigned again on the node.

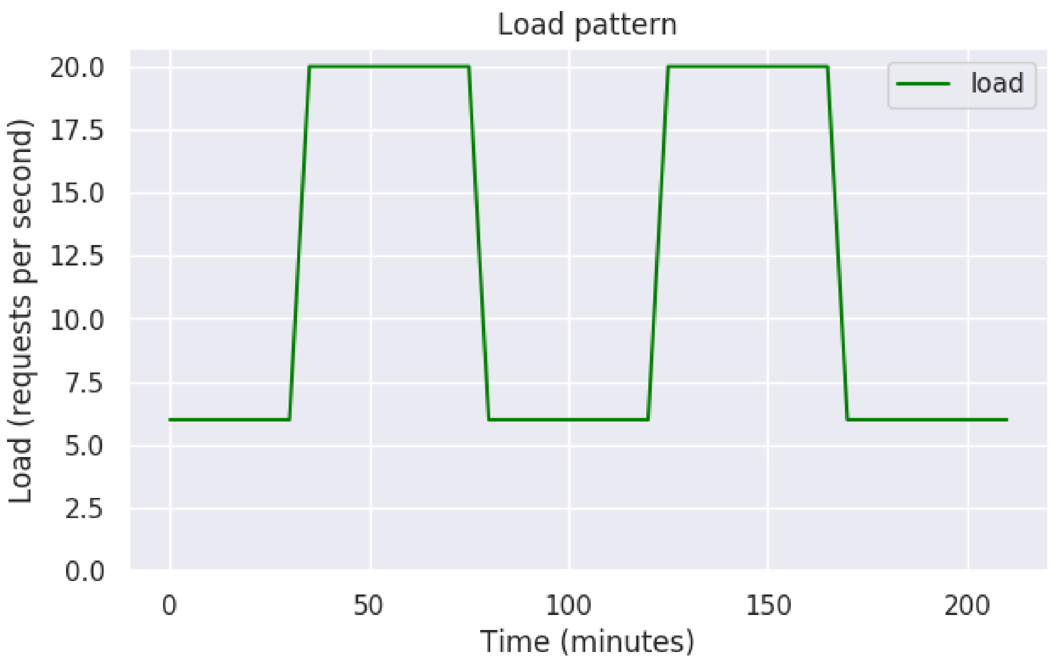

Figure 5.15: Load pattern 1

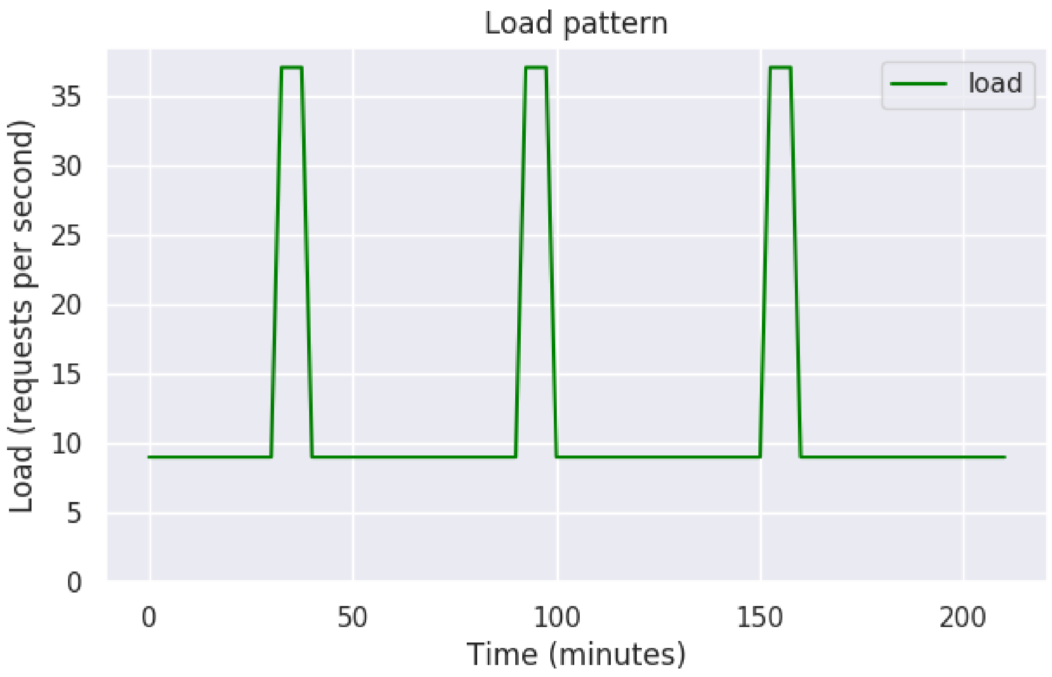

Figure 5.16: Load pattern 2

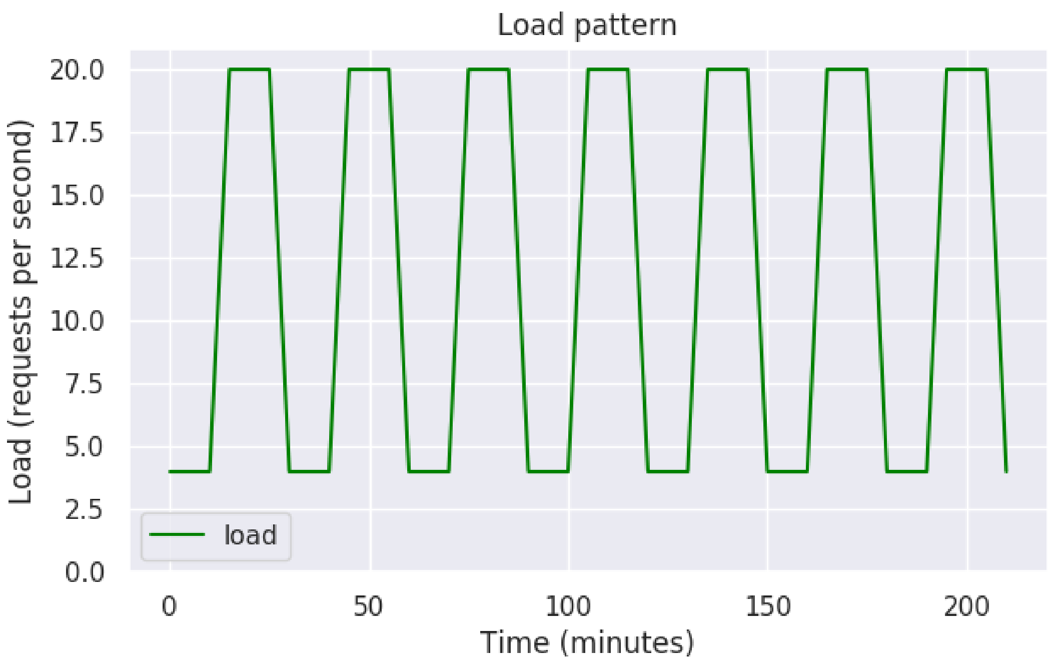

Figure 5.17: Load pattern 3

Improvement over previous phases

The tuner now uses a sliding window to try again when the deployment is deemed unstable instead of a tumbling windows. The sliding window is set to 10% of the normal window size, increasing the chance that the tuner will quickly spot a stable deployment. The warm-up time of the load is increased from 5 minutes to 30 minutes, since it was noticed that anomalies can be caused up till 15 minutes after the last scaling. The new settings file can be found in the repository[7].

Load pattern 1: Wide peaks

Figure 5.15 shows the load pattern that was used. The pattern consists of two wide peaks that have equal width to the valleys that follow. Both the peaks and valleys last for 40 minutes with a transition period of 5 minutes.

The cost of running the tuner with node removal can be seen in Figure C.1. Figure C.2 shows the cost when running the tuner without node removal. Figure C.3 shows the cost for the simulated node removal. In Figure 5.18 are all these cost combined to allow for easy comparison. Table 5.5 provides an overview of the total cost for the three options.

7 https://github.com/Gezzellig/DynamicTopologySelection/blob/master/TopologyGenerator/settings-phase3.json

Node removal	With	Without	Simulated
Total cost with tuner active	€1.014	€1.005	€0.992
Total cost without tuner active	€1.012	€1.012	€1.012
Saved by tuner	€−0.002	€0.007	€0.020
Percentage saved by tuner	−0.19%	0.67%	1.95%

Table 5.4: Savings overview pattern 1

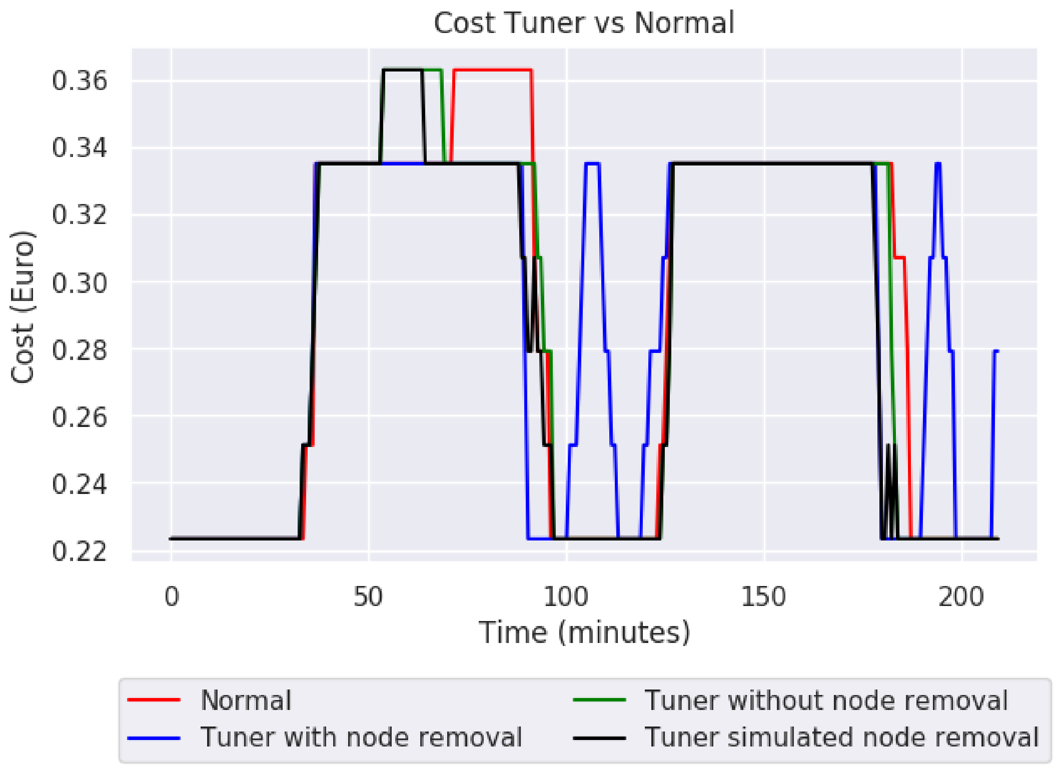

Figure 5.18: Cost all combined pattern 1

The latency for running the system with and without node removal compared to running the system without the tuner can be seen in Figures C.4 and C.5.

Load pattern 2: Narrow peaks

The load pattern used can be found in Figure 5.16. Here you can see that the load pattern consists of three narrow peaks separated by valleys of 50 minutes.

The cost for running the system with and without the tuner while having node removal enabled can be found in Figure C.6. Figure C.7 showing the cost when node removal is disabled and Figure C.8 shows the cost for the simulated node removal. Figure 5.19 merges the previous figures and allows for comparison between the different

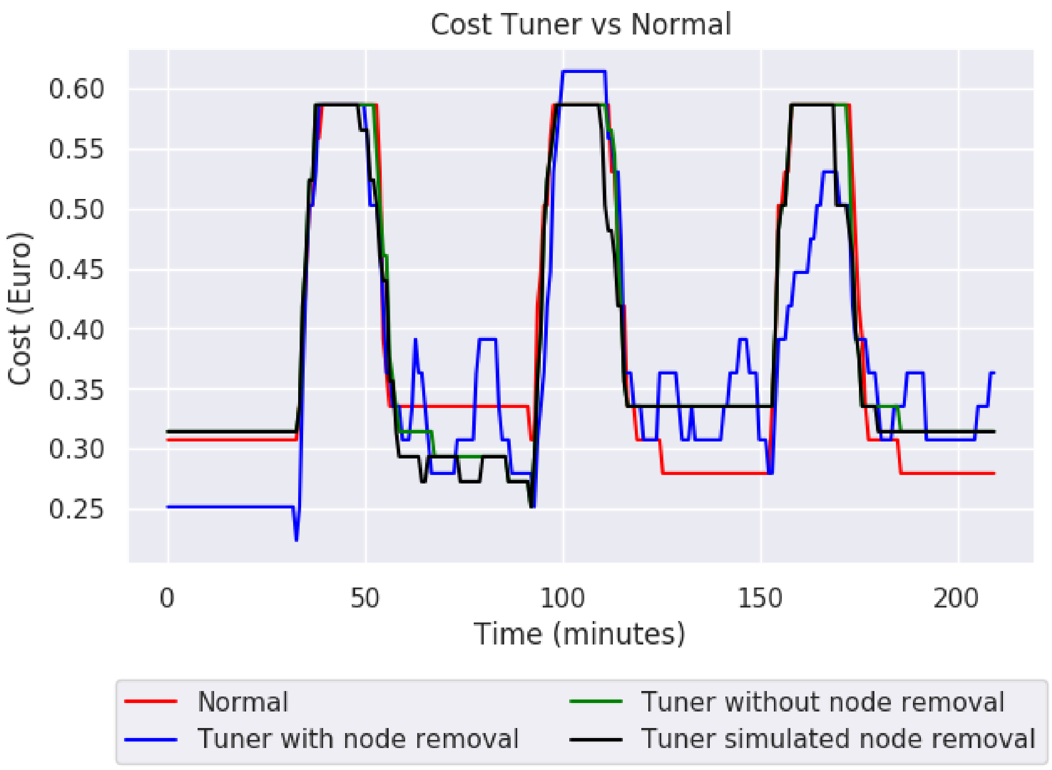

Figure 5.19: Cost all combined pattern 2

options. The total cost of running the system using the different modes can be found in Table 5.5.

Node removal	With	Without	Simulated
Total cost with tuner active	€1.308	€1.367	€1.343
Total cost without tuner active	€1.342	€1.342	€1.342
Saved by tuner	€0.034	€−0.025	€0.001
Percentage saved by tuner	2.54%	−1.84%	−0.11%

Table 5.5: Savings overview pattern 2

The latency of running the deployment with a tuner that allows for node removal can be seen in Figure C.9. Figure C.10 shows the latency when the tuner was not allowed to remove nodes.

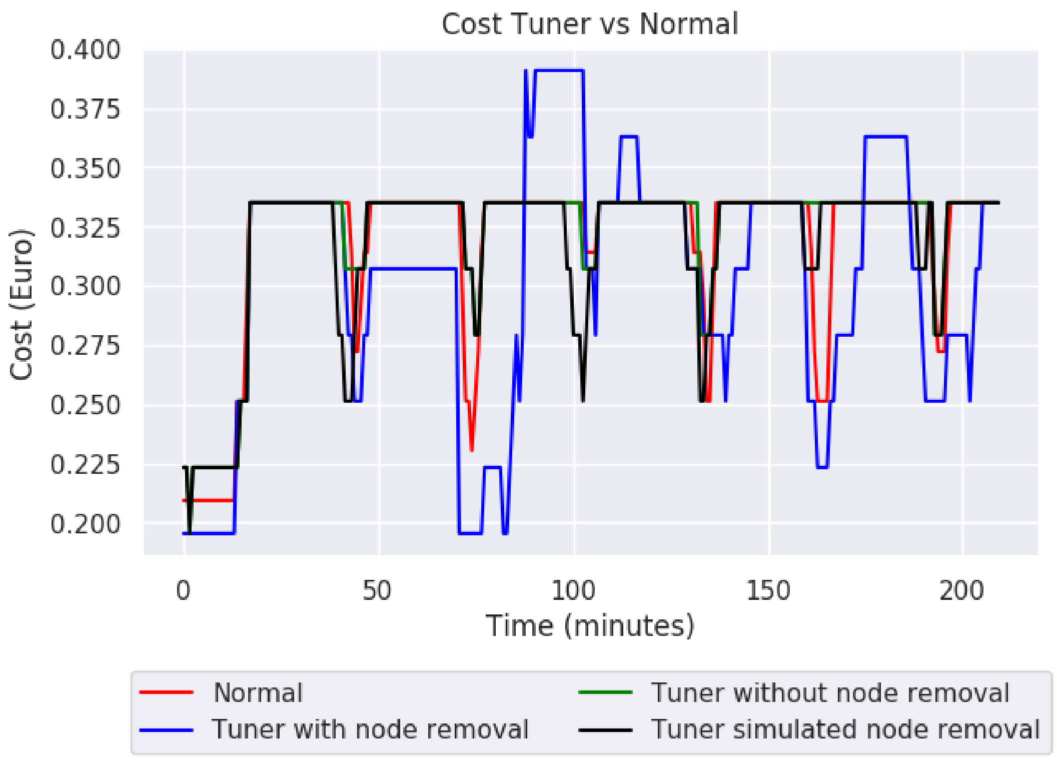

Figure 5.20: Cost all combined pattern 3

Load pattern 3: High frequency

As Figure 5.17 shows this load pattern consist of many peaks and valleys which last 10 minutes. In total there are seven oscillations.

Figure C.11 shows the cost for running the deployment using the tuner with node removal enabled. The cost of running the deployment when node removal is not allowed can be seen in Figure C.12. Figure C.13 shows the cost for the simulated node removal. All the costs are also combined for easy comparison in Figure 5.20. The total costs for running the different options can be found in Table 5.6.

Node removal	With	Without	Simulated
Total cost with tuner active	€1.057	€1.127	€1.113
Total cost without tuner active	€1.115	€1.115	€1.115
Saved by tuner	€0.058	€−0.013	€0.001
Percentage saved by tuner	5.16%	−1.16%	0.10%

Table 5.6: Savings overview pattern 3

The latency for running the deployment using the tuner with node removal can be seen in Figure C.14. Figure C.15 shows the latency for applying the tuner without node removal.

Analysis

The first observation that can be made is that the node reappearing after deletion is still present in this analysis as can be seen in the valleys of Figures C.1, C.6 and C.11.

When looking at the total cost results for every load pattern (Tables 5.4, 5.5 and 5.6) it is observed that the tuner with node removal allowed is only able to make a minor improvement, and in the third load pattern this does cause a decrease in performance as can be seen in Figure C.14. The other two load patterns did not have this decrease in performance as can be seen in Figure C.4 and C.9. In these load patterns the valleys where wide enough for the node reappearing anomalies to happen without negatively affecting the scaling needed for the next increase of load. Using this knowledge it is concluded that the tuner should not be applied on any deployments where the frequency of changing load is significantly larger than the time to scale a node up and down. In addition, it can be concluded that the tuner should not be applied on systems where performance is critical.

The resulting cost for applying the tuner without node removal (Tables 5.4, 5.5 and 5.6) did not yield a significant reduction in cost as was already expected from the results found in phase 2.

The total cost results of running the tuner while simulating the node removals (Tables 5.4, 5.5 and 5.6) resulted in less cost savings than initially expected. When looking at the end of every peak in Figures C.3, C.8 and C.13 the tuner does allow for a slightly faster down-scaling and therefore decrease in cost, but this effect is smaller then expected. This shows that the current autoscaler of Kubernetes is already very capable of scaling a single service actively.

The initial use-case where the tuner would be applied on a more complex microservice system could not be evaluated, due to the absence of appropriate microservice applications. In this use-case the tuner might still yield improvements over the current Kubernetes autoscaler as it is able to evaluate the total deployment of all different pods and can apply redistribution. Each Kubernetes autoscaler can only optimise its own type of pods while lacking the knowledge of the overall deployment. To verify this future evaluation and analysis is needed.

6 CONCLUSION AND FUTURE WORK

In this final chapter the findings of the thesis are concluded and future work is identified.

6.1 CONCLUSION

This thesis has shown that it is possible to create a deployment tuner that runs in parallel with an autoscaler. The tuner is able to monitor the state of the deployment. When the deployment is stable the tuner analyses how the deployment can be improved to make it more cost effective for the current load. The analysis is aided by a knowledge base that holds previous deployments and their cost effectiveness. A planning is constructed that contains the required modifications to reach the envisioned deployment using the least amount of actions. The planned modifications are performed on the deployment in the given order and each modification is verified before the next is performed until reaching the envisioned deployment. Subsequently the tuner will wait for the given time window, after which the next optimisation cycle is initiated.

The evaluation of the deployment tuner consisted of comparing the resulting operational cost of running a cluster with and without the tuner. The comparison was performed with different load patterns to simulate different types of systems. Using these results the research question can be answered.

> **Can deployment tuning reduce the operational cost of microservices?**

The short answer is yes. As is shown in the evaluation the tuner is able to reduce the operational cost, although only by a few percent. The long answer is a bit more nuanced. Yes, the tuner was able to save cost, but the current savings are mainly realised by harming the performance of the system as can be seen by the resulting increase in latency in the evaluation. Using this insight the tuner should only be applied on systems where high performance is not critical.

As the concept of deployment tuning is designed to work with larger deployments that allow the tuner to find beneficial combinations, the real potential of the tuner is not yet established. To properly evaluate the cost reduction that can be achieved, the tuner should be applied on a complete microservice system instead of a single service as part of future work.

6.2 FUTURE WORK

Five other interesting future works are identified and listed below. The first three focus on expanding and improving the tuner allowing it to make smarter decisions. The last two are focus on providing more evaluation for the concept of deployment tuning.

Improve deployment tuning

- The current implementation of the tuner only takes into account the CPU usage of pods, memory is not taken into account. Therefore the tuner struggles when memory heavy pods are present in the system. It would be really interesting to combine both metrics to ensure that both resources of nodes are used efficiently. A possible optimisation in such a system would be to put pods with high CPU requirements together with pods needing a lot of memory, thus making maximal use of the available resources and allowing less nodes to be required.
- It would be very interesting to enhance the redistribution capabilities of the tuner to include communication between pods. If two pods communicate a lot they should be placed on the same node thereby reducing communication overhead and increasing efficiency of the system.
- Implement the execute component directly into Kubernetes. This allows more control over the pod placement and precludes usage of labels and verification. Another large benefit is that this implementation will probably solve the problem with reoccurring nodes, as the node removal call can now be performed from inside Kubernetes.

Further analysis on the potential of deployment tuning

- As described in the architecture chapter (Chapter 3), the tuner is designed as not orchestration tool specific. It would be very interesting to implement the tuner for another orchestration tool and compare its operational cost saving abilities with this research.
- Currently, the purpose of the tuner is to reduce the operational cost of a deployment, but deployment tuning could also be applied to improve other metrics. An example of such a metric is balancing the usage of resource over all nodes. Many of the concepts used in this work can still be used, only the analysis component needs to be re-implemented to allow for the optimisation of the newly chosen metric. This is especially interesting for

private clouds as their hardware is purchased instead of rented, therefore it does not make sense to optimise on operational cost.

BIBLIOGRAPHY

[1] Sam Newman. *Building Microservices*. 1st. O'Reilly Media, Inc., 2015. ISBN: 1491950358, 9781491950357.

[2] Camunda. *New Research Shows 63 Percent of Enterprises Are Adopting Microservices Architectures Yet 50 Percent Are Unaware of the Impact on Revenue-Generating Business Processes*. https://www.globenewswire.com/news-release/2018/09/20/1573625/0/en/New-Research-Shows-63-Percent-of-Enterprises-Are-Adopting-Microservices-Architectures-Yet-50-Percent-Are-Unaware-of-the-Impact-on-Revenue-Generating-Business-Processes.html. [Online; accessed 24-July-2019]. 2019.

[3] Laura Mauersberger. *Why Netflix, Amazon, and Apple Care About Microservices*. https://blog.leanix.net/en/why-netflix-amazon-and-apple-care-about-microservices. [Online; accessed 24-July-2019]. 2019.

[4] R. S. Shariffdeen, D. T. S. P. Munasinghe, H. S. Bhathiya, U. K. J. U. Bandara, and H. M. N. D. Bandara. "Workload and Resource Aware Proactive Auto-scaler for PaaS Cloud." In: *2016 IEEE 9th International Conference on Cloud Computing (CLOUD)*. 2016, pp. 11–18. DOI: 10.1109/CLOUD.2016.0012.

[5] *An Architectural Blueprint for Autonomic Computing*. Tech. rep. IBM, June 2005.

[6] M. D. McIlroy, E. N. Pinson, and B. A. Tague. "Unix Time-Sharing System Forward." In: *The Bell System Technical Journal* 57.6, part 2 (1978), p.1902.

[7] L. Baresi, S. Guinea, G. Quattrocchi, and D. A. Tamburri. "MicroCloud: A Container-Based Solution for Efficient Resource Management in the Cloud." In: *2016 IEEE International Conference on Smart Cloud (SmartCloud)*. 2016, pp. 218–223. DOI: 10.1109/SmartCloud.2016.42.

[8] Datadog. *8 suprising facts about real docker adoption*. https://www.datadoghq.com/docker-adoption/. [Online; accessed 24-July-2019]. 2019.

[9] Emiliano Casalicchio. "Autonomic Orchestration of Containers: Problem Definition and Research Challenges." In: *Proceedings of the 10th EAI International Conference on Performance Evaluation Methodologies and Tools on 10th EAI International Conference on Performance Evaluation Methodologies and Tools*. VALUETOOLS&#39;16. Taormina, Italy: ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2017, pp. 287–290. ISBN: 978-1-63190-141-6. DOI: 10.4108/

eai.25-10-2016.2266649. URL: https://doi.org/10.4108/eai.25-10-2016.2266649.

[10] Luis M. Vaquero, Luis Rodero-Merino, and Rajkumar Buyya. "Dynamically Scaling Applications in the Cloud." In: *SIGCOMM Comput. Commun. Rev.* 41.1 (Jan. 2011), pp. 45–52. ISSN: 0146-4833. DOI: 10.1145/1925861.1925869. URL: http://doi.acm.org/10.1145/1925861.1925869.

[11] Kubernetes. *Horizontal Pod Autoscaler*. https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/. [Online; accessed 24-July-2019]. 2019.

[12] Google. *Cluster autoscaler*. https://cloud.google.com/kubernetes-engine/docs/concepts/cluster-autoscaler. [Online; accessed 22-August-2019]. 2019.

[13] Gokul Soundararajan, Cristiana Amza, and Ashvin Goel. "Database Replication Policies for Dynamic Content Applications." In: *SIGOPS Oper. Syst. Rev.* 40.4 (Apr. 2006), pp. 89–102. ISSN: 0163-5980. DOI: 10.1145/1218063.1217945. URL: http://doi.acm.org/10.1145/1218063.1217945.

[14] Jin Chen, G Soundararajan, and C Amza. "Autonomic Provisioning of Backend Databases in Dynamic Content Web Servers." In: vol. 2006. July 2006, pp. 231 –242. DOI: 10.1109/ICAC.2006.1662403.

[15] A. Ashraf, B. Byholm, and I. Porres. "CRAMP: Cost-efficient Resource Allocation for Multiple web applications with Proactive scaling." In: *4th IEEE International Conference on Cloud Computing Technology and Science Proceedings*. 2012, pp. 581–586. DOI: 10.1109/CloudCom.2012.6427605.

[16] U. Sharma, P. Shenoy, S. Sahu, and A. Shaikh. "Kingfisher: Cost-aware elasticity in the cloud." In: *2011 Proceedings IEEE INFOCOM*. 2011, pp. 206–210. DOI: 10.1109/INFCOM.2011.5935016.

[17] Vasilios Andrikopoulos. "Engineering Cloud-Based Applications: Towards an Application Lifecycle." In: *Advances in Service-Oriented and Cloud Computing*. Ed. by Zoltán Ádám Mann and Volker Stolz. Cham: Springer International Publishing, 2018, pp. 57–72. ISBN: 978-3-319-79090-9.

[18] P. Townend, S. Clement, D. Burdett, R. Yang, J. Shaw, B. Slater, and J. Xu. "Invited Paper: Improving Data Center Efficiency Through Holistic Scheduling In Kubernetes." In: *2019 IEEE International Conference on Service-Oriented System Engineering (SOSE)*. 2019, pp. 156–15610. DOI: 10.1109/SOSE.2019.00030.

[19] Kubernetes. *cAdvisor standard in Kubernetes*. https://kubernetes.io/docs/tasks/debug-application-cluster/resource-usage-monitoring/#cadvisor. [Online; accessed 24-June-2019]. 2019.

[20] Kubernetes. *Assigning Pods to Nodes*. https://kubernetes.io/docs/concepts/configuration/assign-pod-node/. [Online; accessed 20-August-2019]. 2019.

[21] Google. *All pricing*. https://cloud.google.com/compute/all-pricing. [Online; accessed 1-July-2019]. 2019.

A CODE LISTINGS

A.1 FIND REMOVABLE NODE

```python
def movable(pod_info):
    """
    Returns if the given pod can be moved.
    """
    return pod_info["deployment_name"] is not None

def removable(pod_info):
    """
    Returns if the given pod can be removed.
    """
    return movable(pod_info) or pod_info["kind"] == "DaemonSet"

def node_removable(pods_info):
    """
    Checks if the given pods are removable.
    """
    for pod_info in pods_info:
        if not removable(pod_info):
            return False
    return True


def select_removable_nodes(nodes):
    """
    Filters the nodes that can't be removed from the provided
        nodes.
    """
    removable_nodes = []
    for name, node_info in nodes.items():
        if node_removable(node_info["pods"]):
            removable_nodes.append(name)
    return removable_nodes


def find_pods_to_be_rescheduled(pods):
    """
    Returns all pods that have to be rescheduled on a node.
    """
    reschedule = []
    for pod in pods:
        if extract_pods.movable(pod):
            reschedule.append(pod)
    return reschedule
```

```python


def least_transitions_removable(removable_nodes, nodes):
    """
    Selects the node that can be removed with the least amount of
        changes.
    """
    least_transitions = math.inf
    least_transitions_node = None
    for node_name in removable_nodes:
        num_pods = len(find_pods_to_be_rescheduled(nodes[
            node_name]["pods"]))
        if num_pods < least_transitions:
            least_transitions = num_pods
            least_transitions_node = node_name
    return least_transitions_node


def recursive_find_new_distributions(reschedule_pods, new_nodes):
    """
    Recursive function that tries all possible migrations and
        returns possible distributions.
    """
    #Base case no pods have to be rescheduled anymore
    if not reschedule_pods:
        return [new_nodes]

    #Recursive step
    suitable_distributions = []
    pod = reschedule_pods.pop()
    for node_name in new_nodes.keys():
        new_new_nodes = copy.deepcopy(new_nodes)
        new_new_nodes[node_name]["pods"].append(pod)
        if node_request_fits(new_new_nodes[node_name]):
            suitable_distributions +=
                recursive_find_new_distributions(copy.deepcopy(
                reschedule_pods), new_new_nodes)
    return suitable_distributions


def find_new_distributions(reschedule_pods, new_nodes):
    """
    Helper function to start the recursive search for possible
        new distributions.
    """
    return recursive_find_new_distributions(copy.deepcopy(
        reschedule_pods), copy.deepcopy(new_nodes))


def get_max_requested(distribution):
    """
    Return the requested value of the node the has the largest requested value of the given distribution
    """
    max_requested = 0.0
    for node_info in distribution.values():
        requested = node_sum_requested(node_info)
        if requested > max_requested:
            max_requested = requested
    return max_requested


def select_lowest_max_requested(distributions):
    """
    Return the distribution that has the lowest maximum requested value over all nodes from the given distributions.
    """
    lowest_max_requested = math.inf
    lowest_max_requested_distribution = None
    for distribution in distributions:
        max_requested = get_max_requested(distribution)
        if max_requested < lowest_max_requested:
            lowest_max_requested = max_requested
            lowest_max_requested_distribution = distribution
    return lowest_max_requested_distribution


def change_selected_distribution_into_transitions(candidate_node_name, selected_distribution, original_distribution):
    """
    Translate the node removal into a standardised format that the planner understands.
    """
    transitions = {
        candidate_node_name: {
            "delete": True,
            "add": [],
            "remove": []
        }
    }
    for node_name, node_info in selected_distribution.items():
        for pod in node_info["pods"]:
            if pod not in original_distribution[node_name]["pods"]:
                if pod["node_name"] not in transitions:
                    transitions[pod["node_name"]] = {"add": [], "remove": []}
                transitions[pod["node_name"]]["remove"].append(pod["pod_name"])
                if node_name not in transitions:
                    transitions[node_name] = {"add": [], "remove": []}
                transitions[node_name]["add"].append(pod["
                    pod_generate_name"])
    return transitions


def empty_node_transitions():
    """
    Find if any node can be emptied and removed
    """
    nodes = extract_nodes.extract_all_nodes_cpu_pods()
    removable_nodes = select_removable_nodes(nodes)
    candidate_node_name = least_transitions_removable(
        removable_nodes, nodes)
    if candidate_node_name is None:
        log.info("No node could be shutdown for improvement,
            because all nodes have a statefull set")
        return False, None, None

    reschedule_pods = find_pods_to_be_rescheduled(nodes[
        candidate_node_name]["pods"])
    nodes_node_removed = copy.deepcopy(nodes)
    del nodes_node_removed[candidate_node_name]
    distributions = find_new_distributions(reschedule_pods,
        nodes_node_removed)
    if not distributions:
        log.info("No node could be shutdown for improvement,
            because all the resources are needed")
        return False, None, None
    selected_distribution = select_lowest_max_requested(
        distributions)
    return True, candidate_node_name,
        change_selected_distribution_into_transitions(
        candidate_node_name, selected_distribution, nodes)
```

Listing A.1: Find removable node

A.2 NODE MATCHING

```python
def scoring(pod_info, not_movable_score, movable_score):
    """
    This function return the appropriate score for the given pod
        type.
    """
    if pod_info["kind"] == "DaemonSet":
        return 0
    if movable(pod_info):
        return movable_score
    else:
        return not_movable_score


def pod_score(pod_a_name, pod_a, node_b, copy_node_a, copy_node_b
    , not_movable_score, movable_score):
    """
    Tries to find a pod of the deployement of pod_a in the node_b
        .
    """
    for pod_b_name, pod_b in node_b["pods"].items():
        if pod_a["pod_generate_name"] == pod_b["pod_generate_name
            "]:
            if pod_b_name in copy_node_b["pods"]:
                del copy_node_b["pods"][pod_b_name]
                del copy_node_a["pods"][pod_a_name]
                return scoring(pod_a, not_movable_score,
                    movable_score)
    return 0


def remaining_pods_score(copy_node, not_movable_score,
    movable_score):
    """
    Returns the score that will be subtracted as it are pods that
         are not matched.
    """
    score = 0
    for pod_info in copy_node["pods"].values():
        score += scoring(pod_info, not_movable_score,
            movable_score)
    return score


def calc_score_per_node(node_a, node_b, not_movable_score,
    movable_score):
    """
    Returns the score of matching the contents of two nodes.
    """
    copy_node_a = copy.deepcopy(node_a)
    copy_node_b = copy.deepcopy(node_b)
    score = 0
    for pod_a_name, pod_a_info in node_a["pods"].items():
        score += pod_score(pod_a_name, pod_a_info, node_b, copy_node_a, copy_node_b, not_movable_score, movable_score)

    score -= remaining_pods_score(copy_node_a, not_movable_score, movable_score)
    score -= remaining_pods_score(copy_node_b, not_movable_score, movable_score)
    return score


def get_scores(current_state, desired_state):
    """
    Generates all possible combinations between nodes and calculates the score for each combination.
    """
    not_movable_score = 100
    movable_score = 1

    #Current -> Desired
    scores = {}
    for node_current_name, node_current_info in current_state.items():
        score = {None: -remaining_pods_score(node_current_info, not_movable_score, movable_score)}
        for node_desired_name, node_desired_info in desired_state.items():
            cur_score = calc_score_per_node(node_current_info, node_desired_info, not_movable_score, movable_score)
            score[node_desired_name] = cur_score
        scores[node_current_name] = score
    return scores


def rec_find_highest_score(current_state_list, desired_state_list, scores):
    """
    Uses recursion to find the highest scoring mapping of current nodes to desired nodes.
    """
    if not current_state_list:
        return 0, {}

    copy_current_state_list = list(current_state_list)
    node_name = copy_current_state_list.pop()
    max_score = -math.inf
    max_mapping = None
    for node_b_name in desired_state_list:
        score = scores[node_name][node_b_name]
        copy_desired_state_list = list(desired_state_list)
        copy_desired_state_list.remove(node_b_name)
        prev_score, prev_mapping = rec_find_highest_score(copy_current_state_list, copy_desired_state_list, scores)
        cur_score = prev_score + score
        prev_mapping[node_name] = node_b_name
        cur_mapping = prev_mapping
        if cur_score > max_score:
            max_score = cur_score
            max_mapping = cur_mapping
    # CHANGE IT Mapping is from current -> desired
    return max_score, max_mapping


def find_highest_score_mapping(current_state_list, desired_state_list, scores):
    """
    Helper function to start the recursion loop to find the highest scoring mapping of current nodes to desired nodes.
    """
    for i in range(0, len(current_state_list)-len(desired_state_list)):
        desired_state_list.append(None)
    score, mapping = rec_find_highest_score(current_state_list, desired_state_list, scores)
    return mapping


def match_nodes_desired_with_current_state(current_state, desired_state):
    """
    Retrieves first the matching scores and uses this to initiate the search for the best mapping. This mapping is returned.
    """
    scores = get_scores(current_state, desired_state)
    return find_highest_score_mapping(list(current_state.keys()), list(desired_state.keys()), scores)


def already_on_node(des_pod_info, current_state_pods, remove_list):
    """
    Checks if the given node is already present. Taking into account previously matched node using the remove_list.
    """
    for cur_pod_name, cur_pod_info in current_state_pods.items():
        if des_pod_info["pod_generate_name"] == cur_pod_info["pod_generate_name"]:
            if cur_pod_name in remove_list:
                remove_list.remove(cur_pod_name)
                return True
    return False


def remove_daemon_sets(state):
    """
    Deamonsets do not have to be taken into account when matching nodes, and therefore can be removed.
    """
    state_copy = copy.deepcopy(state)
    for node_name, node_info in state.items():
        for pod_name, pod_info in node_info["pods"].items():
            if pod_info["kind"] == "DaemonSet":
                del state_copy[node_name]["pods"][pod_name]
    return state_copy


def valid_transition(add_list, remove_list, pods):
    """
    Verifies that no unmovable pods have to be moved.
    """
    for name in add_list:
        for pod_info in pods.values():
            if pod_info["pod_generate_name"] == name:
                if not removable(pod_info):
                    return False
    for name in remove_list:
        if not removable(pods[name]):
            return False
    return True


def find_transitions_execution_change(current_state, desired_state):
    """
    First matches the nodes.
    Verifies the matching.
    Generates the transition description that can be used by the planner.
    """
    transitions = {}
    node_mapping = match_nodes_desired_with_current_state(current_state, desired_state)

    daemon_less_current_state = remove_daemon_sets(current_state)
    daemon_less_desired_state = remove_daemon_sets(desired_state)

    for cur_node_name, cur_node_info in daemon_less_current_state.items():
        des_mapped_node_name = node_mapping[cur_node_name]
        add_list = []
        remove_list = list(cur_node_info["pods"].keys())
        if des_mapped_node_name is not None:
            for des_pod_info in daemon_less_desired_state[des_mapped_node_name]["pods"].values():
                if not already_on_node(des_pod_info, cur_node_info["pods"], remove_list):
                    add_list.append(des_pod_info["pod_generate_name"])

        if not valid_transition(add_list, remove_list, cur_node_info["pods"]):
            return False, None

        transitions[cur_node_name] = {
            "add": add_list,
            "remove": remove_list
        }

        if node_mapping[cur_node_name] is None:
            transitions[cur_node_name]["delete"] = True
    return True, transitions
```

Listing A.2: Node matching

A.3 MIGRATION SET GENERATION

```python
def merge_found_migrations_sets(migration, migrations_sets):
    """"
    Adds the given migration to all elements present in migration_sets.
    """
    for other in migrations_sets:
        other.append(migration)
    migrations_sets.append([migration])
    return migrations_sets


def find_suitable_migrations_sets(selected_add, destination_node, transitions):
    """
    For one add action finds all possible suitable remove actions to and lists them in the migration sets.
    """
    migrations_sets = []
    for source_node, content in transitions.items():
        for pod_remove in content["remove"]:
            if re.match("{}*".format(selected_add), pod_remove):
                migration = {"pod_name": pod_remove, "source": source_node, "destination": destination_node}
                new_trans = copy.deepcopy(transitions)
                new_trans[source_node]["remove"].remove(pod_remove)
                migrations_sets += merge_found_migrations_sets(migration, recurse_find_all_migrations_sets(new_trans))
    return migrations_sets


def recurse_find_all_migrations_sets(transitions):
    """
    Recursively generates all possible migration sets that can be retrieved from the transitions.
    """
    selected_add = None
    selected_node = None
    for node_name, content in transitions.items():
        if content["add"]:
            selected_add = content["add"].pop(0)
            selected_node = node_name
            break

    # Base case
    if selected_add is None:
        return []

    # migrate
    migrations_sets = find_suitable_migrations_sets(selected_add,
         selected_node, transitions)

    # don’t migrate
    migrations_sets += recurse_find_all_migrations_sets(copy.
        deepcopy(transitions))
    return migrations_sets


def find_all_migrations_sets(transitions):
    """
    Starts the recursive function to retrieve all migration sets,
         and then orders them decreaslingy on length.
    """
    migrations_sets = recurse_find_all_migrations_sets(copy.
        deepcopy(transitions))
    migrations_sets.append([])

    # Sort the result so that the longest migration is in front,
        as we want to try this one the first
    migrations_sets.sort(key=len, reverse=True)
    return migrations_sets
```

Listing A.3: Migration set generation

A.4 VERIFY MIGRATION SET

```python
def extract_deployment(pods, nodes):
    """
    Transforms the given pods and nodes into a dictionary where
        the pods running on a node are added in a list under the
        node names key.
    """
    deployment = {}
    for node_name in nodes:
        deployment[node_name] = []
    for pod_name, info in pods.items():
        node = info["node_name"]
        deployment[node].append(pod_name)
    return deployment


def simulate_migration(cur_deployment, migration):
    """
    Generates the resulting deployment after the given migration
        would be performed
    """
    pod_name = migration["pod_name"]
    source = migration["source"]
    destination = migration["destination"]
    cur_deployment[source].remove(pod_name)
    cur_deployment[destination].append(pod_name)
    return cur_deployment


def verify_deployment(deployment, pods, nodes):
    """
    Checks for every node if the amount of requested cpu does not
         excel the available cpu.
    """
    for node_name, pod_names in deployment.items():
        cpu_available = nodes[node_name]["cpu"]
        cpu_needed = 0.0
        for pod_name in pod_names:
            cpu_needed += pods[pod_name]["total_requested"]
        if cpu_needed > cpu_available:
            return False
    return True


def helper_recursive_construction_migration_order(migration,
    migrations, cur_deployment, pods, nodes):
    """
    Helps with finding of the given set of migrations can be
        performed and return its found order
    """
    migrations.remove(migration)
    cur_deployment = simulate_migration(cur_deployment, migration)
    if not verify_deployment(cur_deployment, pods, nodes):
        return False, []
    return recursive_construction_migration_order(migrations, cur_deployment, pods, nodes)


def recursive_construction_migration_order(migrations, cur_deployment, pods, nodes):
    """
    Recursively searches for a possible order of migrations to perform the provided migrations.
    When this is found it is returned.
    """
    # Base case no more migrations to schedule
    if not migrations:
        return True, []

    for migration in migrations:
        success, result = helper_recursive_construction_migration_order(migration, list(migrations), copy.deepcopy(cur_deployment), pods, nodes)
        if success:
            return True, [migration] + result
    return False, []


def find_suitable_migrations(transitions, migrations_sets, pods, nodes):
    """
    Finds the first and therefore largest migrations set that can be executed and returns its order.
    """
    for migrations_set in migrations_sets:
        local_pods_removed = remove_non_migrated_remove_pods(transitions, migrations_set, pods)
        cur_deployment = extract_deployment(local_pods_removed, nodes)
        result, migration_order = recursive_construction_migration_order(migrations_set, cur_deployment, local_pods_removed, nodes)
        if result:
            return migration_order
```

Listing A.4: Verify migration set

B TEST OUTPUT

B.1 ADD POD

Initial state

```
NAME         DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
php-apache   4         4         4            4           12d

NODE                                            POD
gke-demo-cluster-1-default-pool-6f471531-d2dq   php-apache-5f657688bc-bmtcb
gke-demo-cluster-1-default-pool-6f471531-83rr   php-apache-5f657688bc-h6fkb
gke-demo-cluster-1-default-pool-6f471531-d2dq   php-apache-5f657688bc-pxbw8
gke-demo-cluster-1-default-pool-6f471531-83rr   php-apache-5f657688bc-zsj99
```

Log output

```
INFO -- Creating pod: php-apache on gke-demo-cluster-1-default-pool-6f471531-83rr
INFO -- Addition succeeded, pod name: php-apache-5f657688bc-mwtjl
```

Final state

```
NAME         DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
php-apache   5         5         5            5           12d

NODE                                            POD
gke-demo-cluster-1-default-pool-6f471531-d2dq   php-apache-5f657688bc-bmtcb
gke-demo-cluster-1-default-pool-6f471531-83rr   php-apache-5f657688bc-h6fkb
gke-demo-cluster-1-default-pool-6f471531-83rr   php-apache-5f657688bc-mwtjl
gke-demo-cluster-1-default-pool-6f471531-d2dq   php-apache-5f657688bc-pxbw8
gke-demo-cluster-1-default-pool-6f471531-83rr   php-apache-5f657688bc-zsj99
```

B.2 MIGRATE POD

Initial state

```
NAME         DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
php-apache   4         4         4            4           12d

NODE                                            POD
gke-demo-cluster-1-default-pool-6f471531-83rr   php-apache-5f657688bc-h6fkb
gke-demo-cluster-1-default-pool-6f471531-83rr   php-apache-5f657688bc-mwtjl
gke-demo-cluster-1-default-pool-6f471531-d2dq   php-apache-5f657688bc-pxbw8
gke-demo-cluster-1-default-pool-6f471531-83rr   php-apache-5f657688bc-zsj99
```

Log output

```
INFO -- moving: php-apache-5f657688bc-h6fkb to gke-demo-cluster-1-default-pool-6f471531-d2dq
INFO -- pod php-apache-5f657688bc-h6fkb is deleted
INFO -- Movement succeeded
```

Final state

```
NAME         DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
php-apache   4         4         4            4           12d

NODE                                             POD
gke-demo-cluster-1-default-pool-6f471531-83rr   php-apache-5f657688bc-mwtjl
gke-demo-cluster-1-default-pool-6f471531-d2dq   php-apache-5f657688bc-pxbw8
gke-demo-cluster-1-default-pool-6f471531-d2dq   php-apache-5f657688bc-vg87c
gke-demo-cluster-1-default-pool-6f471531-83rr   php-apache-5f657688bc-zsj99
```

B.3 REMOVE POD

Initial state

```
NAME         DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
php-apache   5         5         5            5           12d

NODE                                             POD
gke-demo-cluster-1-default-pool-6f471531-d2dq   php-apache-5f657688bc-bmtcb
gke-demo-cluster-1-default-pool-6f471531-83rr   php-apache-5f657688bc-h6fkb
gke-demo-cluster-1-default-pool-6f471531-83rr   php-apache-5f657688bc-mwtjl
gke-demo-cluster-1-default-pool-6f471531-d2dq   php-apache-5f657688bc-pxbw8
gke-demo-cluster-1-default-pool-6f471531-83rr   php-apache-5f657688bc-zsj99
```

Log output

```
INFO -- Deleting pod: php-apache-5f657688bc-bmtcb
INFO -- Deletion pod: php-apache-5f657688bc-bmtcb successful
```

Final state

```
NAME         DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
php-apache   4         4         4            4           12d

NODE                                             POD
gke-demo-cluster-1-default-pool-6f471531-83rr   php-apache-5f657688bc-h6fkb
gke-demo-cluster-1-default-pool-6f471531-83rr   php-apache-5f657688bc-mwtjl
gke-demo-cluster-1-default-pool-6f471531-d2dq   php-apache-5f657688bc-pxbw8
gke-demo-cluster-1-default-pool-6f471531-83rr   php-apache-5f657688bc-zsj99
```

B.4 REMOVE NODE

Initial state

```
NAME                                             STATUS   ROLES    AGE     VERSION
gke-demo-cluster-1-default-pool-6f471531-1sq8   Ready    <none>   4m22s   v1.12.8-gke.10
gke-demo-cluster-1-default-pool-6f471531-83rr   Ready    <none>   45m     v1.12.8-gke.10
gke-demo-cluster-1-default-pool-6f471531-d2dq   Ready    <none>   45m     v1.12.8-gke.10
```

Log output

```
INFO -- Deleting node: gke-demo-cluster-1-default-pool-6f471531-1sq8
INFO -- Deletion successful, node: gke-demo-cluster-1-default-pool-6f471531-1sq8
```

Final state

```
NAME                                            STATUS   ROLES    AGE   VERSION
gke-demo-cluster-1-default-pool-6f471531-83rr   Ready    <none>   48m   v1.12.8-gke.10
gke-demo-cluster-1-default-pool-6f471531-d2dq   Ready    <none>   48m   v1.12.8-gke.10
```

C EVALUATION PLOTS

C.1 PHASE 3: PATTERN 1

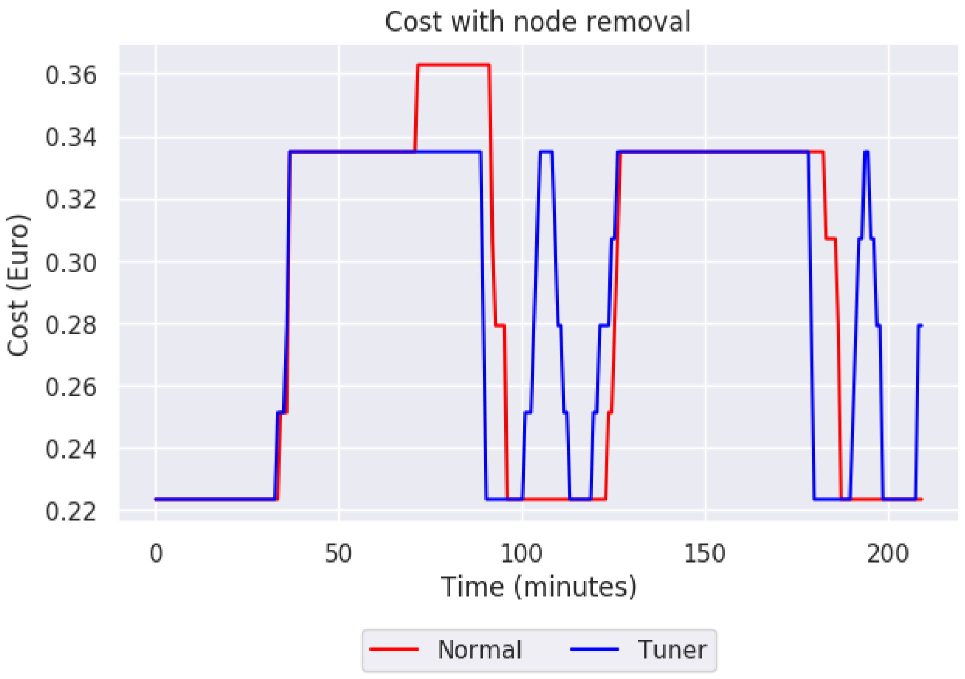

Figure C.1: Cost with node removal pattern 1

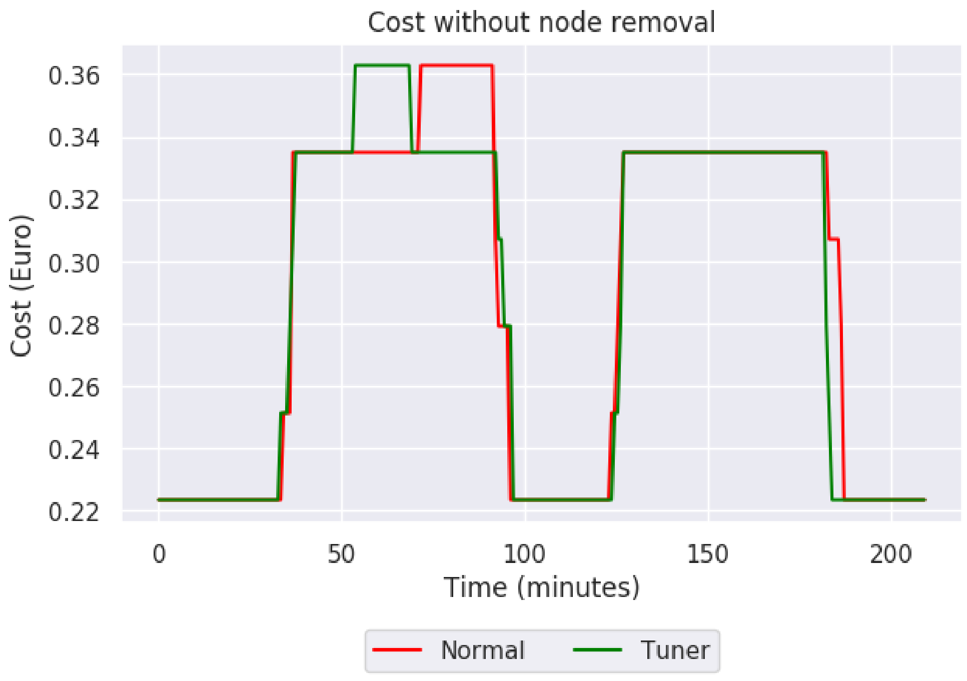

Figure C.2: Cost without node removal pattern 1

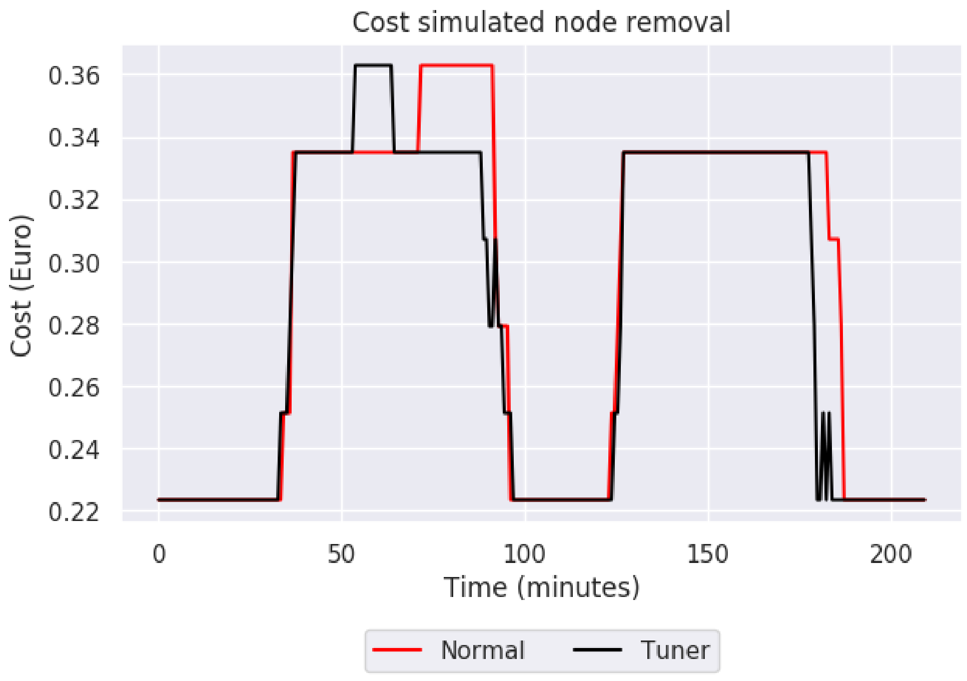

Figure C.3: Cost simulated node removal pattern 1

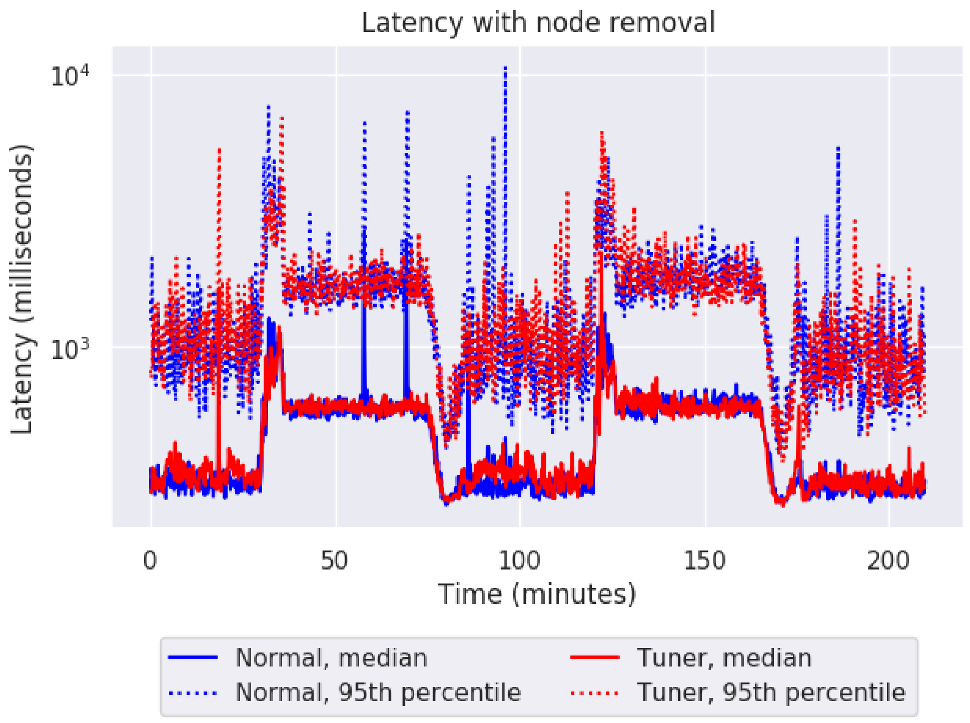

Figure C.4: Latency with node removal pattern 1

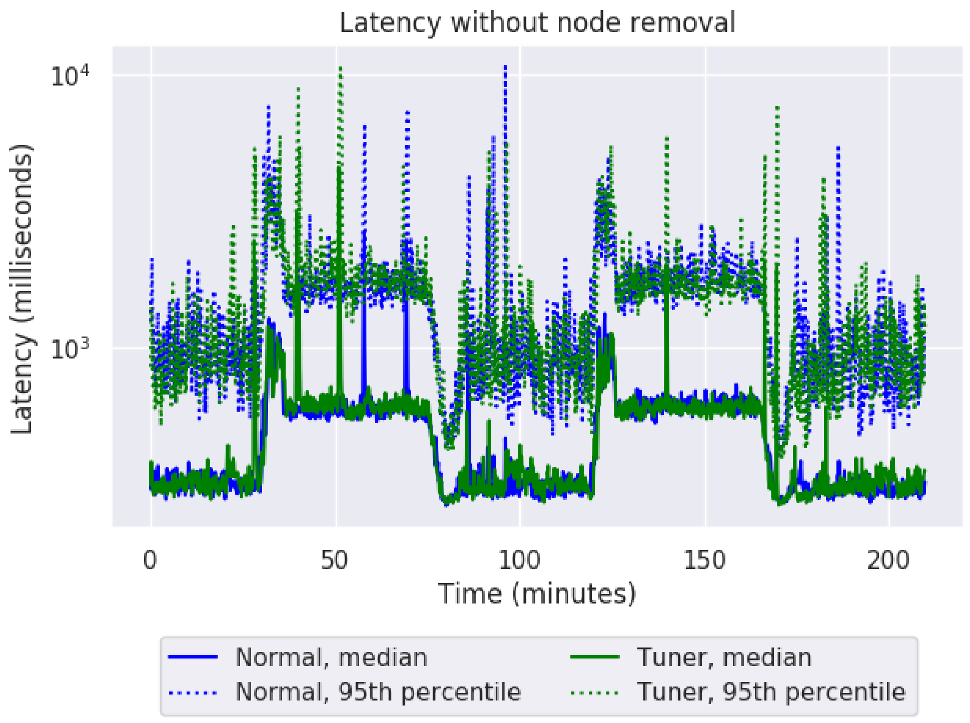

Figure C.5: Latency without node removal pattern 1

C.2 PHASE 3: PATTERN 2

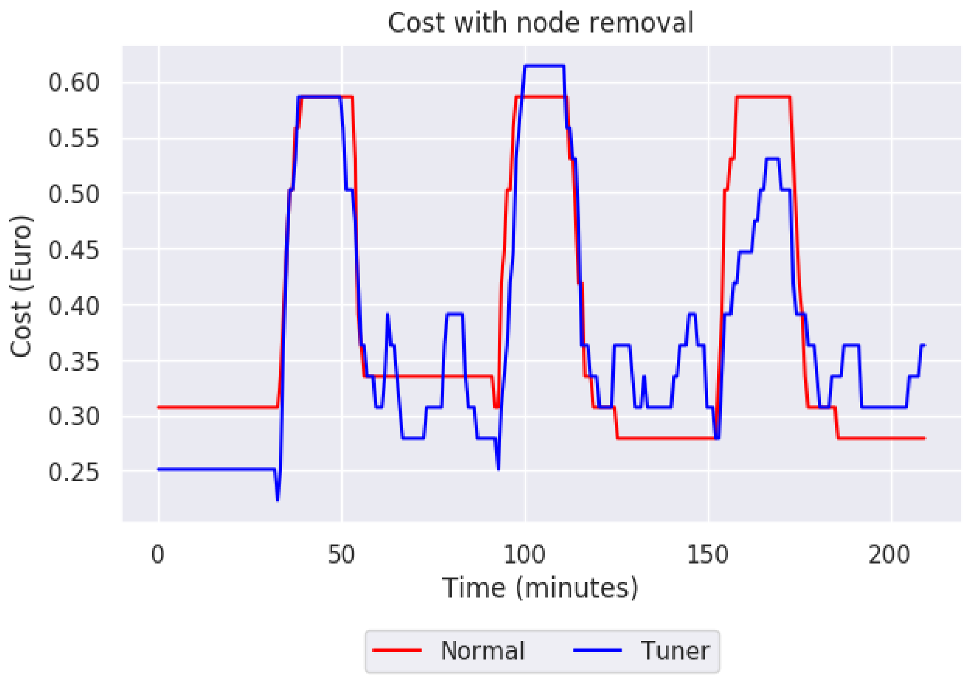

Figure C.6: Cost with node removal pattern 2

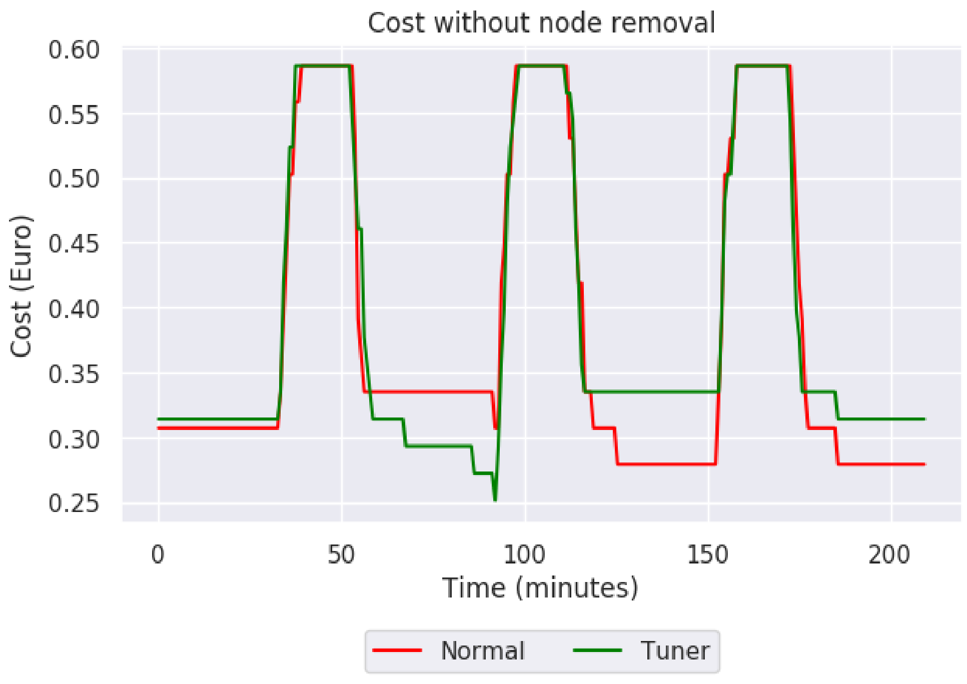

Figure C.7: Cost without node removal pattern 2

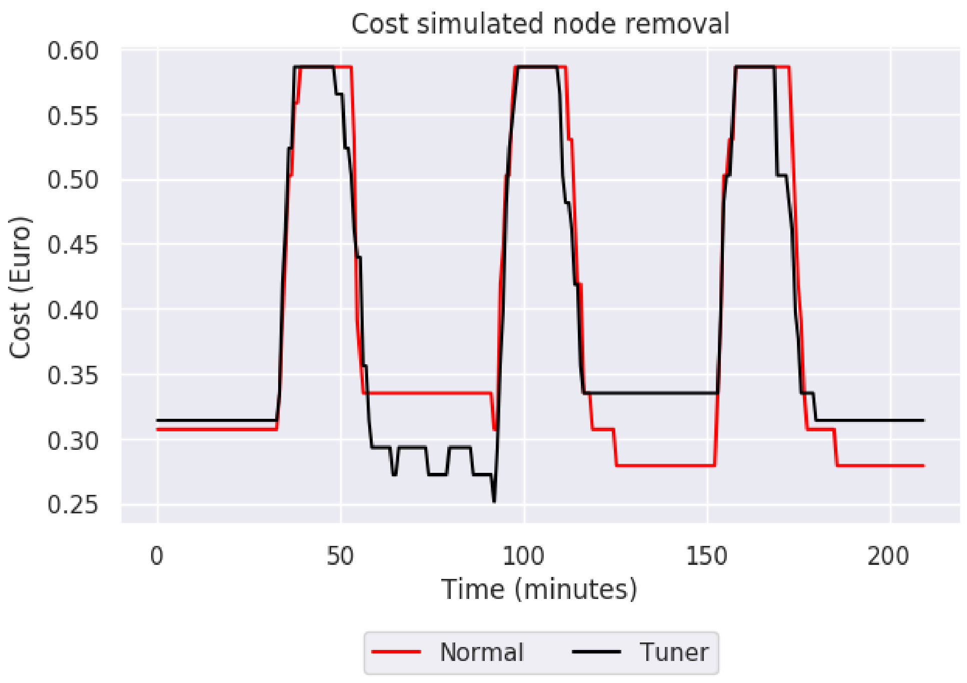

Figure C.8: Cost simulated node removal pattern 2

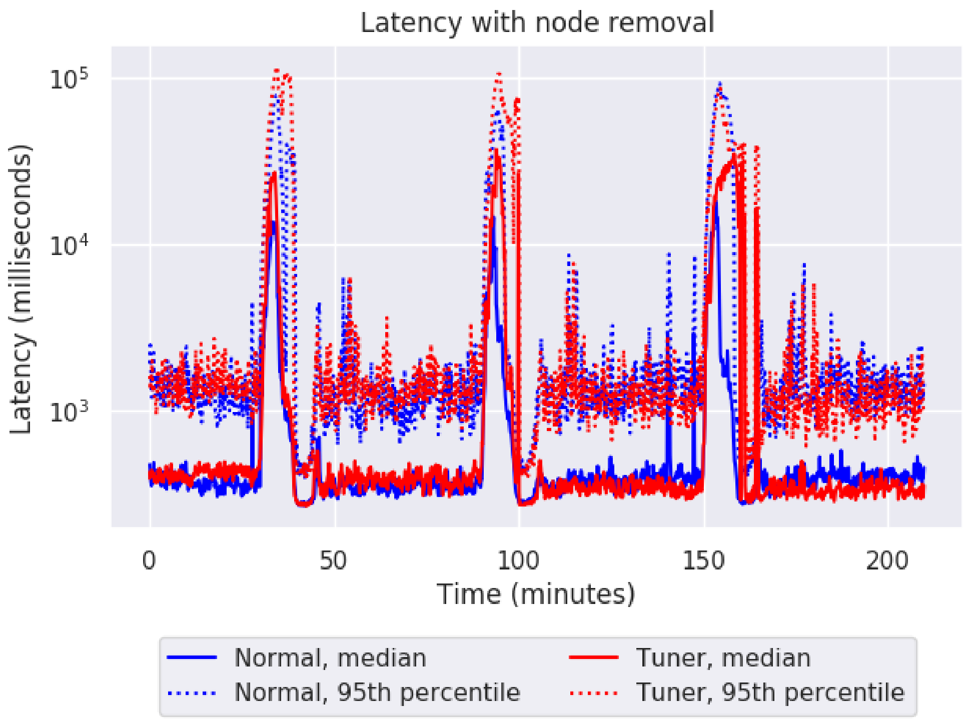

Figure C.9: Latency with node removal pattern 2

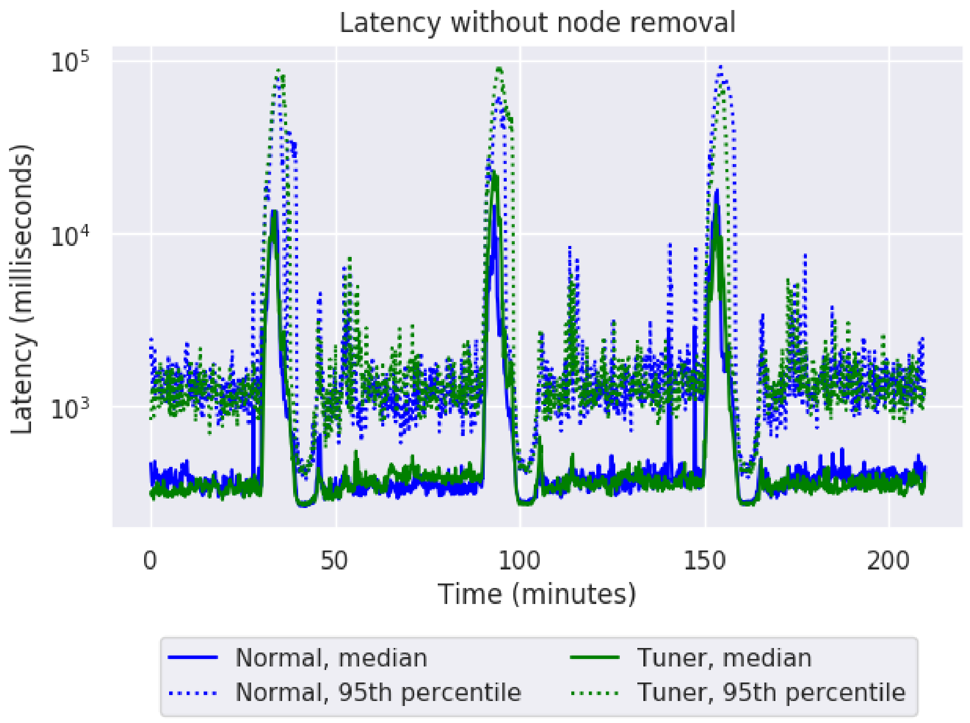

Figure C.10: Latency without node removal pattern 2

C.3 PHASE 3: PATTERN 3

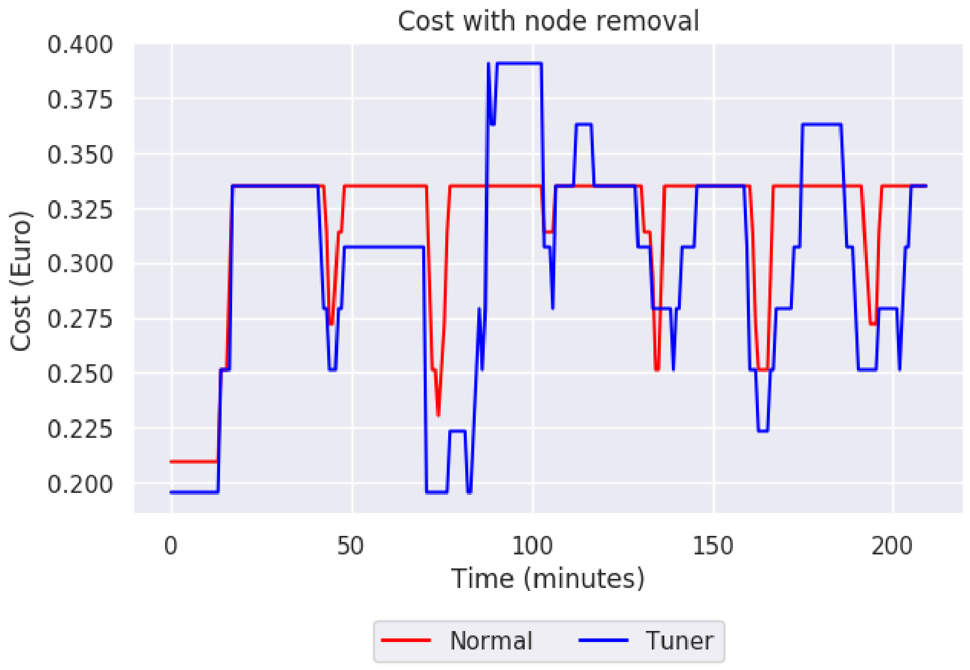

Figure C.11: Cost with node removal pattern 3

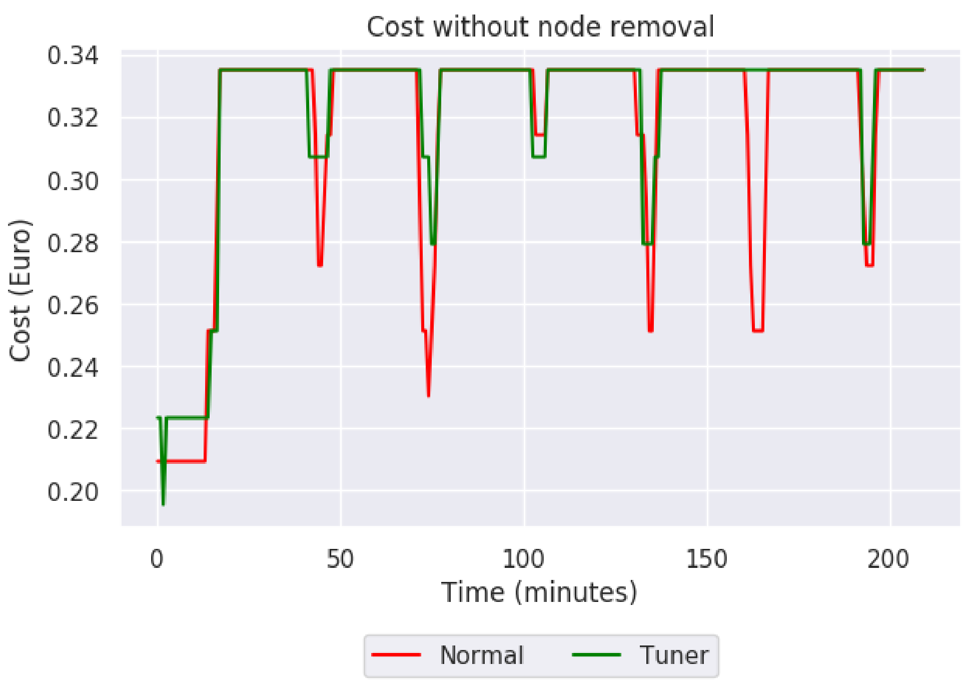

Figure C.12: Cost without node removal pattern 3

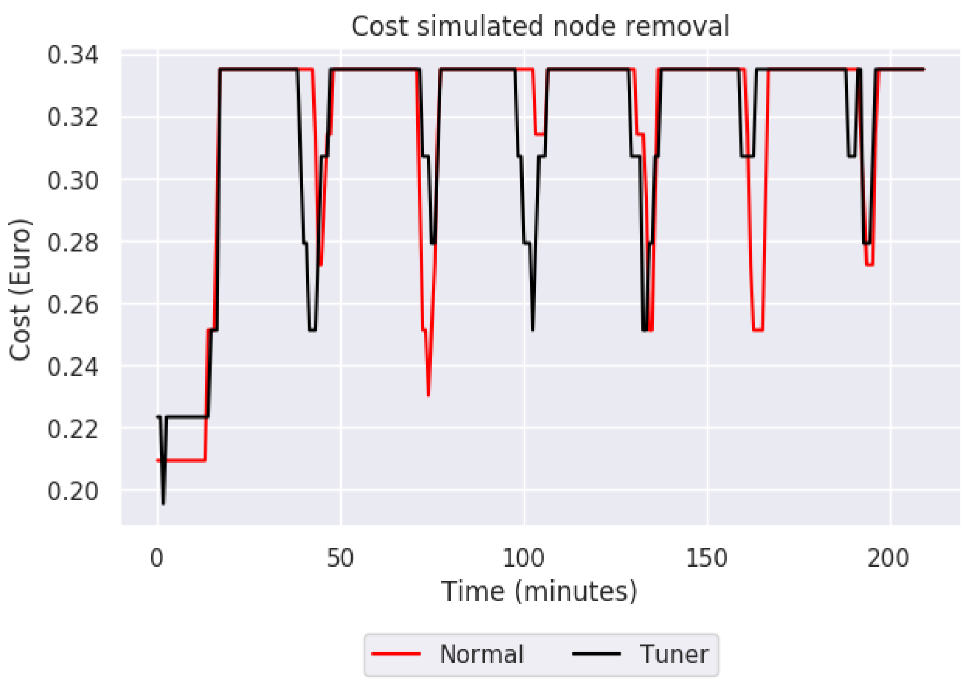

Figure C.13: Cost simulated node removal pattern 3

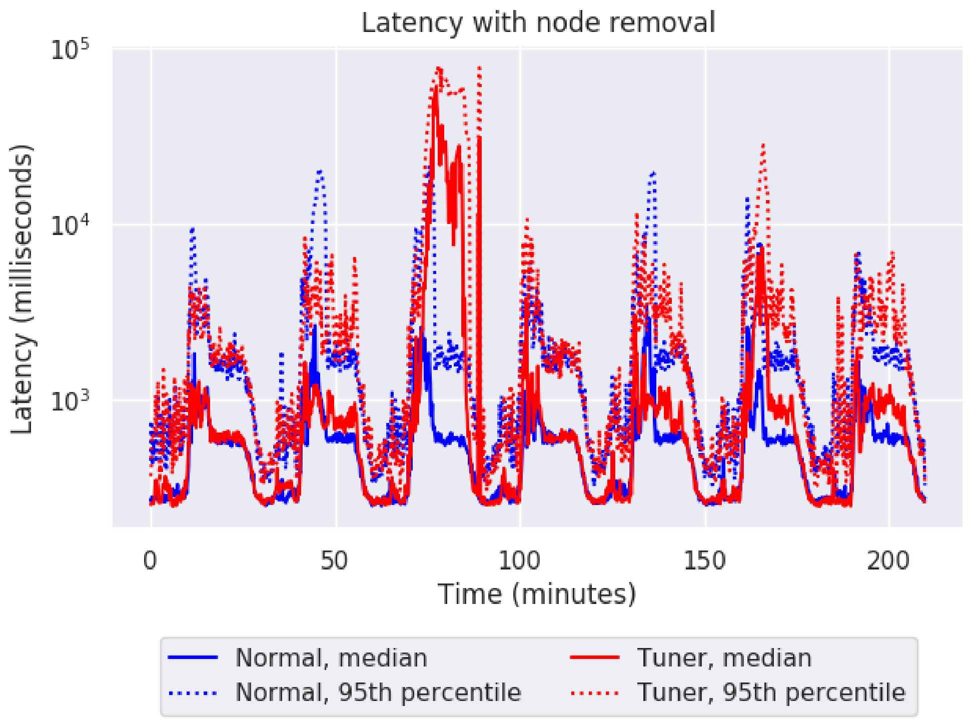

Figure C.14: Latency with node removal pattern 3

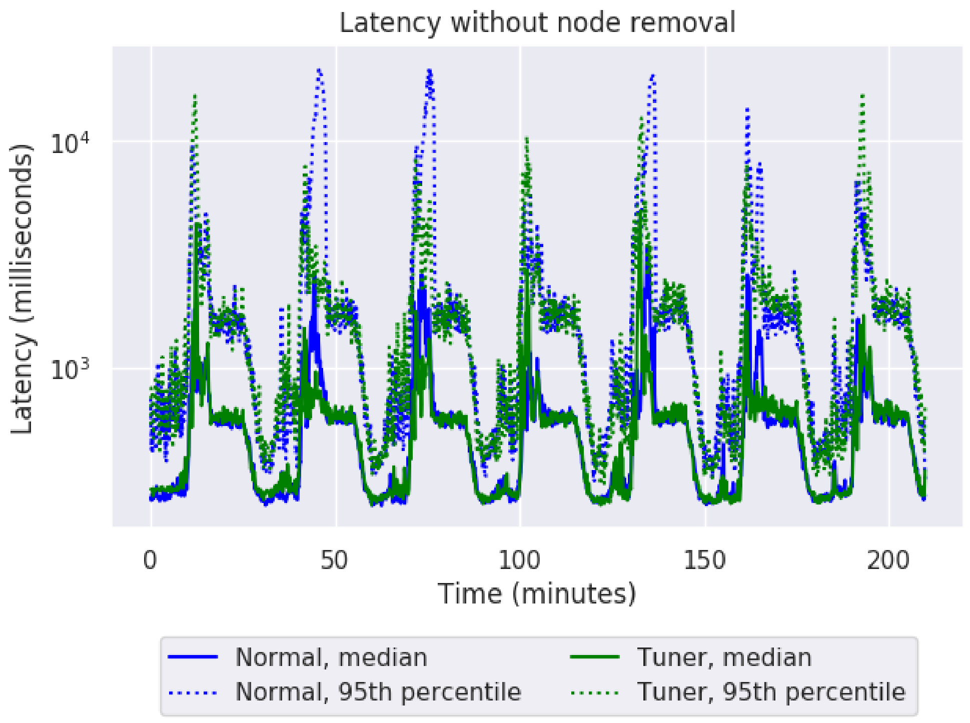

Figure C.15: Latency without node removal pattern 3