ABSTRACT

MASHAT, SARA JAMEEL A. CPU Autoscaling in Cloud Computing Environments. (Under the direction of Ioannis Viniotis).

In this thesis, we address *CPU autoscaling*, a technical problem related to resource provisioning in cloud computing environments.

Pay-as-you-go models are the prevalent financial models in public cloud computing environments due to the cost reduction they offer to cloud users. CPU consumption is a major part of such costs, so using (and paying) only for what a user needs is critical. Given that the CPU load a user generates varies with time and may also be random, there is a need to *adjust (up or down)* the CPU resources the cloud provider offers to the user, in order to match the user need as close as possible. This is the CPU autoscaling problem.

There is a number of technical challenges one needs to address in tackling the problem. A healthy amount of work has been done in both industrial as well as academic settings. In this thesis, we focus on utilizing historical measurements regarding CPU utilization in order to improve system performance. We design three algorithms for processing such measurements and use actual load traces to test them. We use four industry-standard metrics introduced by the SPEC Research Group in 2016 to compare how they fare against each other as well as against the Horizontal Pod Autoscaler, the default (CPU and memory) autoscaling algorithm in kubernetes, a popular resource orchestration tool in cloud computing.

CPU Autoscaling in Cloud Computing Environments

by
Sara Jameel A Mashat

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Master of Science

Computer Engineering

Raleigh, North Carolina
2022

APPROVED BY:

Gregory Byrd

Shih-Chun Lin

Ioannis Papapanagiotou
External Member

Ioannis Viniotis
Chair of Advisory Committee

BIOGRAPHY

Sara Mashat was born in Jeddah, Saudi Arabia in 1993. She received her Bachelor of Science degree in Computer Science from Effat University, Jeddah, Saudi Arabia in 2016. She defended her computer engineering master thesis under Prof. Ioannis Viniotis supervision. During her master at NC state university, Sara worked as a teaching assistant. ...

ACKNOWLEDGEMENTS

First of all, I am sincerely grateful to my supervisors, Prof. Ioannis Viniotis for his continuous guidance and patience during my Master thesis. His guidance carried me through all the stages of writing and implementing my thesis. Working under Prof. Viniotis supervision has been a true honor and I appreciate for everything he has done for me in the past year.

I would also like to thank my committee members, Prof. Greg Byrd, Prof. Shih-Chun Lin, and Dr. Ioannis Papapanagiotou for their comments and feedback. I would like to thank Dr. Sudhendu Kumar for his suggestions and advice that have helped me in accomplishing the thesis.

My sincere thanks go to my family and friends who have been constant sources of love, friendship, hope and strength. Thank you for being there for me without your love and continuous support this research would not have been possible.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER

1

INTRODUCTION

In the past decade and a half, cloud computing has become a popular topic that has gained considerable attention in both academic as well as industrial environments [37]. A very large number of companies, such as Netflix, Dropbox, and Spotify are adopting cloud-based applications due to the scalability, reliability, and cost-effectiveness that cloud computing provides [13] [29] [44]. By connecting to the cloud via the public or private Internet, consumers can access compute, storage and network resources on-demand; cloud resources can be utilized only whenever needed. Therefore, cloud vendors provide pay-as-you-go billing, so consumers are only responsible to pay for the amount of service they use.

1.1 Problem Statement and Our Objectives

In the cloud environment, traffic levels may usually change throughout the day, leading to a dynamic demand of resources. To meet this demand in a cost-effective manner, most cloud systems employ an *auto scaling feature*, by which the number of resources provided is automatically increased or decreased based on the workload. There is one main issue

that must be considered in the design and operation of an auto scaling algorithm. This issue is *responsiveness*: that is, the speed at which the auto scaling algorithm responds to the frequent change of traffic while maintaining other desirable properties such as cost efficiency and quality of service. The algorithm must recognize short time increases as well as decreases of demand; if needed resources are not given in time, *underprovisioning* occurs, and the user's quality of service will deteriorate. If resources are not taken in time when they are no longer needed, *overprovisioning* occurs, and user costs will increase.

So, at a high level, *the problem is to design an auto scaling algorithm that will have small (ideally zero) over- or underprovisioning mismatches*.

We will define later, in Section 3.3, page 29, four specific, industry-standard metrics that quantify these mismatches. We use these metrics in our evaluations in Chapter 4.

In this thesis, we propose three simple algorithms that use past history to decide how to scale up or down the CPU resource. This history includes past resource utilizations as well as previous decisions. We have implemented the algorithms using the Go programming language. Our goal is to evaluate how these algorithms fare against each other with respect to the four metrics.

1.2 Thesis Organization

This thesis is structured as follows. In Chapter 2, we provide the necessary background knowledge and present our literature review. In Chapter 3, we propose the system model, define the selected metrics, and describe the proposed algorithms. In Chapter 4, we use actual data from two realistic datasets in order to measure, evaluate, and analyze the efficiency of our proposed solutions. We also compare them to kubernetes HPA, the most widely used autoscaling algorithm in the industry. In Chapter 5, we summarize the major findings, describe the challenges we faced in this work and provide some recommendations for future work. Finally, we describe the major features of our code in the two Appendices.

CHAPTER

2

BACKGROUND AND LITERATURE REVIEW

In this chapter, we highlight the most important background information needed in this research; more specifically, in Sections 2.1 through 2.5, we briefly outline the concepts of cloud computing, containers, infrastructure management, Kubernetes, and feedback-based algorithms respectively. We conclude this chapter with an overview of literature that is closely related to our problem in Section 2.6.

2.1 Cloud Computing

In the past decade and a half, cloud computing has received great interest in the industrial and academic fields [14]. At its 2019 state of the cloud report that surveyed 786 enterprises, RightScale has reported that 94% of them utilize (private or public) cloud computing [25]. For example, cloud computing has been used as an alternative solution to a traditional data warehouse due to its ability to provide instant access to a wide range of shared computing resources such as networks, servers, storage, applications, and services that are managed

by cloud service providers such as Amazon, Google, and Microsoft.

Consumers using cloud computing pursue numerous cloud benefits such as wide network access, pooling of resources, elasticity, and pay-as-you-go billing. The first of these benefits is that cloud offers its clients convenient, on-demand access to shared computing resources [34] [22]. This means that clients can easily connect to their cloud from any geographical location and through different platforms, like a laptop or phone, using a web browser. In addition, cloud offers provisional services to their clients [33] [15] [34]. In other words, pooling computing resources, physical or virtual resources, is more flexible and efficient since it allows multiple clients to access a resource at the same time. Elasticity is the third benefit of cloud computing as it is able to adjust to clients' varying demands by adding and removing resources such as CPU and memory in real time [28] [27] [36] [49]. Thus, the cloud dynamically calculates the required amount of resources in a current situation, so clients will not worry if the system is under-provisioned or over-provisioned. This feature is also known as scalability. Finally, the last benefit is that cloud charges their clients based on their usage [15]. This means that consumers only pay for the resources that are needed.

There are three[1] different cloud service models that offer unique solutions to meet customers' needs [26] [24] [34]. The following is a brief description of these cloud service models:

1. **Software as a Service (SaaS)**

   In the SaaS model, consumers access a single, cloud-based application through the internet, typically through a browser; the providers manage everything related to the application (e.g., licencing, backups, storage, faults, etc.). This means that clients are not required to install any infrastructure. An example is Overleaf, the latex editor used to write this thesis. Another example is Dropbox, the cloud service for storing and sharing files and data.

2. **Infrastructure as a Service (IaaS)**

   In this model, the provider allocates (virtual) computing, storage and network resources to the client. Clients are responsible for managing them (e.g., install their preferred operating systems and applications, scale the resources, balance the load among resources). Amazon Web Services, Microsoft Azure, and Google Compute Engine are examples of providers who offer IaaS services [47].

[1] Recently, a fourth one, Function as a Service (FaaS) has also gained steam.

3. **Platform as a Service (PaaS)**

   This model is in the middle between IaaS and SaaS. In PaaS, cloud providers manage virtual machines and operating systems and most management tasks (e.g., load balancing). Consumers can deploy applications using specific programming languages, libraries, services, and tools supported by the provider. Google App Engine, and AWS Elastic Beanstalk are examples of PaaS [47].

Figure 2.1 depicts the delineation of management tasks between client and cloud provider under the three models.

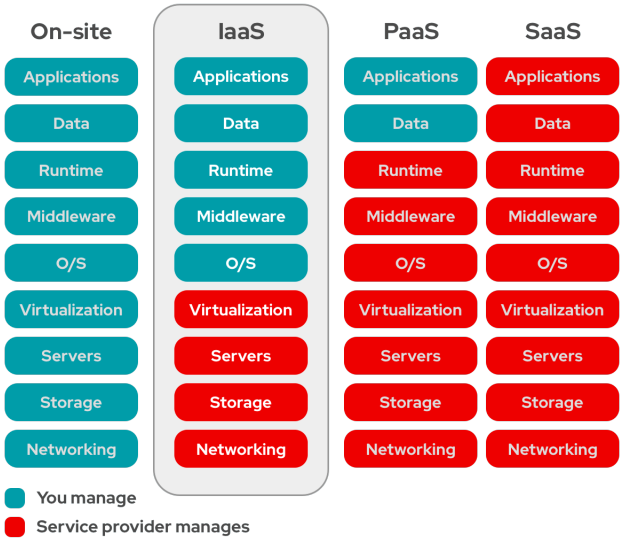


Figure 2.1: How clients and cloud providers share the management burdens.

2.2 Containers

The container technology has emerged with the Linux kernel introduction of control groups (c-group) [31]; its main purpose was to address challenges associated with the preceding technology of virtual machines. A virtual machine (VM) is a virtualization technique that packages a host operating system and desired applications along with their libraries and dependencies [46]. Thus, VMs are a technology created by building images with the operating system, applications and the root file system that is important to deployment.

Moreover, VMs provide strong security as they are completely isolated from the guest OS and other VMs running on the same physical server. However, VMs are heavyweight and not portable [18]; these challenges were addressed by the newer (Docker and Linux) container technology.

More specifically, containers are executable units of software that contain an operating system, application code, runtime environment, system tools, system libraries, binaries, etc.; they are ready to deploy applications [40] [38]. Being lightweight and portable are the main advantages of containers in the cloud. Their first benefit is due to the fact that they share the machine operating system kernel [39]. This means that containers eliminate the need to have a complete operating system instance for each application, so containers have a small code footprint. Therefore, many containers (e.g., Docker and Linux) are portable because they are built around container engines[39]. In other words, containers can run anywhere as they do not need to include an operating system in every instance.

The following is a brief explanation of two popular container implementations, Linux and Docker.

- **Linux Containers**

  A Linux container (LXC) provides an isolated software bundle that is similar to VMs, but LXC is lightweight as they are running their own kernel [45] [35]. This means the kernel in LXC allocates computing resources to provide full access without interference to the processes within the container. LXC is managed by a separate program due to Kernel name-spaces that ensure process isolation.

- **Docker**

  Docker is the leading container platform because it is convenient to use. Docker [12] has an active online community that provides multiple pre-built containers for public use. Docker builds an image that packages user applications by reading the instructions from a Dockerfile. Then, users can pull and push Docker images from the Docker Hub repository to run the containers.

2.3 Infrastructure Management

Broadly speaking, supplying the required resources to external and internal customers through following specific processes and using appropriate tools is attained by cloud infrastructure management. The essential parts of cloud infrastructure include hardware,

network, storage, and virtualization. Examples of hardware used in the cloud are servers, switches, and routers. Storage is responsible for keeping data while network is responsible for connecting users to the cloud through internet connections [9]. Virtualization mainly refers to servers, even though the switches and routers connecting the servers are also virtualized.

Management of cloud infrastructure is particularly paramount to providing optimal and affordable cloud computing solutions. In this thesis, we limit our attention to elements of management of cloud infrastructure that are essential to achieve the following goals [30]:

- **Provisioning and Automation**

  Users can request, configure, and provision assigned resources using tools given by the cloud infrastructure management interface. For example, users are able to initiate new servers, VMs or containers. In addition, these tools can be configured to become automated, and provide auto-provisioning and/or auto-scaling features.

- **Monitoring**

  Monitoring the system's health and performance in the form of real time alerts, notification, periodic analytical reports or utilization measurements is one of the advantages attained from cloud infrastructure management.

- **Resource allocation and Cost optimization**

  Resource allocation and cost optimization are closely related to each other; giving users their needed resources only for an appropriate period of time instead of providing a constant number of resources all the time is a powerful way to reduce cost. This allocation can also be done automatically by enabling auto-scaling features.

2.4 Kubernetes

While containers are essential and provide multiple advantages to the cloud computing domain, managing containers poses a huge challenge especially when containers fail or when they are deployed in very large scale. Therefore, container orchestration became as crucial as containers themselves [23]. Container Orchestration achieves the automation of packaging, scaling, and container recovery, as well as deployment and management [37]. One of the platforms that deliver container orchestration (and arguably the most widely used) is Kubernetes.

Kubernetes, shortened as k8s or kube, is a Google product, based on their Borg and Omega cluster management systems, but it was donated to the Cloud Native Computing Foundation, CNCF [1]. Its popularity has grown enormously and it is now used by various organizations. In addition, managed Kubernetes services are offered by key cloud platform providers such as Amazon, IBM, and Azure. Kubernetes is known for its ability to facilitate running containerized workloads, applications, and other services. Moreover, its architecture allows scalability for distributed systems. Below are pivotal advantages of Kubernetes that motivated its use in this thesis.

1. **API consistency:** Kubernetes deployment for different public and private providers is easier due to its consistent API across all clusters [17].

2. **High Level of Abstraction and Interoperability:** Resources for a specific provider are automatically assigned by Kubernetes as abstract resource types and units for each provider are already integrated in [16]. Moreover, interoperability across various providers is vastly enhanced in Kubernetes compared to other orchestration platforms [43].

2.4.1 Kubernetes Cluster Architecture

A cluster is a result of deploying Kubernetes, and it consists of two main components: the control plane and the worker plane. The control plane manages a set of nodes that run containerized applications and host the application workload components called Pods [6]. In this thesis, we use *replica* and *container* as synonymy to *pods* Figure 2.2 shows the Kubernetes cluster architecture.

2.4.1.1 Control Plane

The control plane, known as master nodes, is the brain of the kubernetes system as its components control the cluster. Each cluster has at least one control plane [37]. Having more than one master nodes increases the availability of the cluster, as if one of the master nodes fails, there is an alternative master node to take control of the cluster. The control plane continuously communicates with the compute machines to detect and respond to a variety of cluster events. This means that the control plane makes a decision to match the current or actual state with the desired state[6].

Four major components comprise the control plane, as drawn in the right side of Figure 2.2.

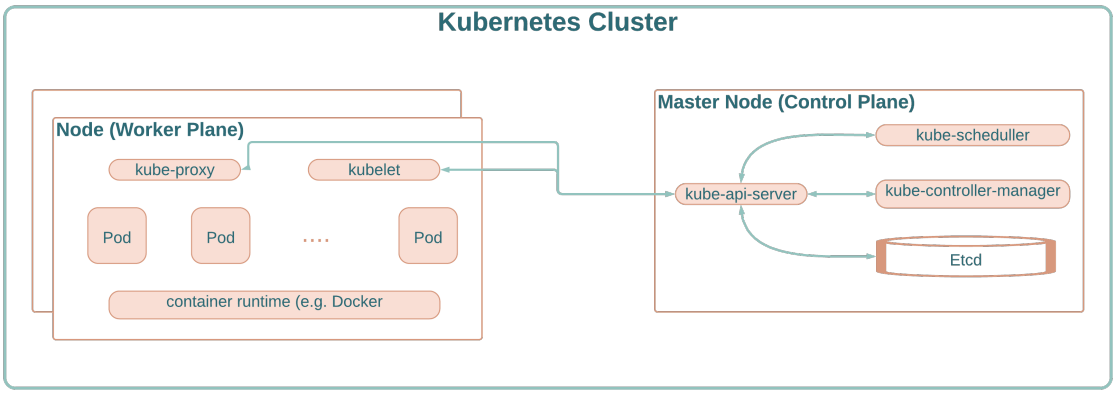


Figure 2.2: Kubernetes Cluster Architecture.

- **kube-controller-manager**

  This component manages Kubernetes by applying control processes such as replication, deployment, job, and services controllers. All these controllers are compiled into one binary controller and apply a single, separate, specific function. For example, the Replication controller [7] monitors nodes and responds in case a node is down or missing by creating a new node; the job controller [5] establishes pods that execute the workflow, and then monitor these pods until the job is completed. The deployment controller [4] updates the pods and replica-set by matching the actual state to a specified, desired state; the service controller [8] generates default accounts and APIs that are used to access the running Pods in the cluster.

- **kube-scheduler**

  Kube-scheduler places new or unassigned workloads (pods) to the most "appropriate" worker node in the cluster by considering predefined constraints and run-time statistics [41]. This means that the kube-scheduler selects a node with the most free resources, and distributes pods across different nodes to maintain resource utilization balance in the worker plane.

- **etcd**

  Etcd is a key-value store database that acts as back storage for all Kubernetes configuration data and information regarding the cluster state.

- **kube-api-server**

The kube-api-server is the front-end of the kubernetes orchestrator. It is communicating with all other components in the control plane and interacts with worker nodes through kubelet and kube-proxy. In other words, kube-api-server is a means to access the internal and external components in the control plane.

2.4.1.2 Worker Plane

Kubernetes system has at least one worker plane that connects to the control plane through kube-api-server to maintain the running pods and report the status of the computing resources [6]. Worker plane elements are also known as (worker) nodes, and contain the required services to run pods.

There are three node components as Figure 2.2 shows.

- **kubelet**

  kubelet is a local agent in every node that ensures that the containers are running a pod as described in PodSpecs, which is the pod specification. The kubelet command line (kubctl) is a tool for the user to interact with the worker plane (e.g., in order to retrieve the cluster state).

- **kube-proxy**

  kube-proxy is the component that's responsible for maintaining the network rules by routing and load-balancing pods in the cluster.

- **Container Runtime**

  The container runtime is a required software agent in all nodes and control plane to run containers in the cluster. There are several container runtimes that Kubernetes supports such as Docker [12], containerd [2], and CRI-O [3]. Docker is arguably the most popular one.

2.4.2 Kubernetes Objects

Kubernetes objects are persistent entities used to display the state of the kubernetes cluster. The following are some of the Kubernetes objects:

2.4.2.1 Pods

Deployable units in Kubernetes vary, and the smallest ones are called Pods. As can be inferred from the name, Pods carry a single one or multiple containers that share the same resources and storage [1]. In other words, Pods are vital in Kubernetes as they provide containers with required computing resources such as CPU, storage and network communication. Kubernetes runs the application workload by placing containers into Pods to run on Worker Nodes. Additionally, an application can run in multiple Pods by having multiple instances of that application working in different Pods. It is important to understand that Pods are *ephemeral*, which means that they are temporary resources that will not restart once destroyed. Hence, when more Pods are required, new ones will be created instead of restarted. Moreover, each pod is assigned a unique IP address, and is linked to a configuration setup, PodSpec, that determines which containers are running on this pod, the method of running these containers as well as assigned resources [1]. This is helpful when an application is trying to request a specific service, running on a specific Pod, so it will look for the Pod's IP address to connect to that service's container. Finally, the creation of Pods can be manual or through a controller.

2.4.2.2 Replica-Sets

Each running pod can have multiple instances, called replicas, of it running. In order to maintain these replicas, Replica-sets are defined [7]. Identifying pods is achieved using a selector field, metadata.ownerReferences, defined in each Replica-set; this also helps Replica-sets to maintain the set of pods it is responsible for. Furthermore, Replica-sets are the specific components that create and delete active pods' replicas, based on the desired number of replicas in addition to the predefined maximum/minimum number of allowed replicas. Each Replica-set contains a pod template based on which new pods are created.

2.4.2.3 Deployment

Deployment is a higher-level controller provided by Kubernetes. Deployment controllers are in charge of maintaining Replica-sets and Pods through defined updates [4]. It is a more recent controller that introduces a few features, one of which is allowing Kubernetes to create new pods whose type is similar to a recently failed Pod under that specific deployment. It also allows declaring new Replica-sets to update to, or return to a previously defined Replica-set.

2.4.2.4 Services

As explained previously, application instances run on Pods; however, Pods are ephemeral, they are created and destroyed constantly. Therefore, a more reliable solution needs to be addressed to allow application instances to communicate with each other. This solution is realized defining Services in a configuration file. Services are an abstract method for discovering pods [8]. For example, to reach a specific Pod, an application will communicate with a Service associated with Pods using selectors and labels, defined by Replica-sets. Services are crucial for applications to speak to each other.

2.4.2.5 Metrics Server

Metrics server plays a significant role in autoscaling algorithms, such as HPA and VPA, as it provides them resource-specific metrics through the Metrics API and Kubernetes API [11]. For example, the Metrics Server collects CPU and memory utilization at a given time for all Worker nodes and Pods (using HTTP). Moreover, the Metrics server has the capability of checking Pods health and metadata both of which can be accessed through an extended API [11].

2.4.3 Autoscaling Types

Applications constantly experience fluctuating workloads, so the autoscale feature in cloud computing is important to maintain availability and performance as utilization increases. Moreover, autoscale helps cloud's clients to pay less as consumers only pay for the resources they use. An autoscaler monitors the application's utilization due to the incoming traffic, then it adjusts the amount of resources to provide the expected Quality of Service (QoS).

An enormous amount of work has been done on autoscaling; several taxonomies have been presented in recent survey papers. Figure 2.3 illustrates the autoscaling taxonomy for web applications provided in a recent survey by [42]. The taxonomy covers the following points:

- **Application Architecture**

  This is the architecture of the web application that the autoscaler is managing. There are three types of architecture: single-tier, multi-tier, and service-oriented architecture. Single-tier architecture is composed of all the application components into a single server or platform. While multi-tier architecture consists of multiple, sequentially-connected software tiers. Thus, application processing, data management functions,

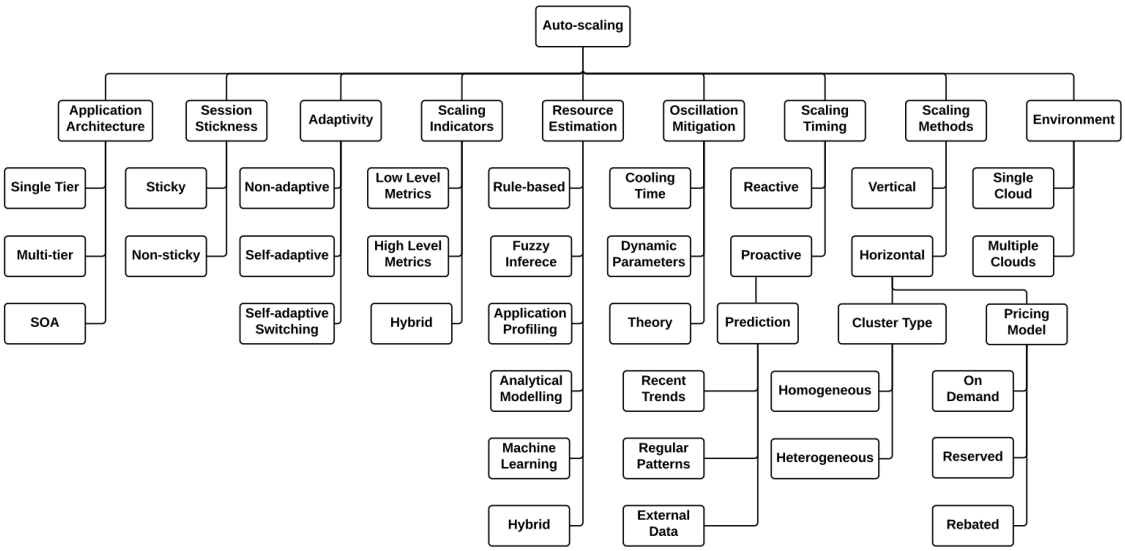


Figure 2.3: The taxonomy for auto-scaling web applications in clouds, [42].

and other components are physically separated. The last architecture type is a service-based architecture that is a collection of independent services which communicate over already defined APIs.

- **Session Stickiness**

  A session is defined as a sequence of interactions between a client and the application, so the client after each operation waits for the application reply. This feature is to indicate if the auto-scaler supports sticky sessions.

- **Adaptivity**

  Adaptivity refers to how the auto-scaler adjusts to the workload changes; there are three approaches: non-adaptive, self-adaptive, and self-adaptive switching. The non-adaptive approach is predefined by the users, so the scaling decision is applied based on the users' instructions and current status. The self-adaptive approach is able to automatically scale according to real-time events by monitoring the target performance. On the other hand, self-adaptive switching alternates the autoscale control between non-adaptive and self-adaptive controllers based on the current application performance.

- **Scaling Indicators**

This classification represents the metrics that the system will monitor, and then the auto-scaler tools determine the necessary actions. Low-level metrics are the resources collected from the server (physical, container or virtual machine). Examples of low-level metrics are CPU utilization, memory, memory swap, cache miss rate, and network utilization. On the other hand, high-level metrics are monitored in the application layer; examples are request rate, session creation rate, throughput, and average response time. In addition, some autoscalers use hybrid metrics, computed as a combination of high-level and low-level metrics.

- **Resource Estimation**

  Resource estimation is the essence of the auto-scaler as it aims to provide the minimum required computing resources to handle the incoming workload. This is the most difficult part of the auto-scale system as it attempts to supply an accurate resource to satisfy the antagonistic goals of great QoS levels and low costs.

- **Oscillation Mitigation**

  Oscillation Mitigation aims to reduce the frequency of provisioning oscillations. A typical method is the introduction of "cooling time". This is the waiting for a fixed minimum period of time between each scaling decision. In Kubernetes, the default cool time is five minutes when the system scales down and 15 seconds when it scales up.

- **Scaling Timing**

  There are two approaches for scaling timing: reactive and proactive. The reactive is an approach in which the application is scaled based on the current status while the proactive approach analyzes historical data to provide the most accurate scaling decision.

- **Scaling Methods**

  The scaling method determines the scaling type. There are three types: autoscaling vertically, horizontally, and cluster. However, it is possible to used hybrid scaling method as horizontal and vertical scaling can be combined to increase the scaling performance.

- **Environment**

  Cloud environment can be single cloud data center or multiple cloud data centers.

2.4.4 Autoscaling Built-in Kubernetes

Kubernetes supports three different types of autoscaling: Horizontal Pod Autoscaler (HPA), Vertical Pod Autoscaler (VPA), and Cluster Autoscaler (CA). While all autoscalers can exist in a kubernetes environment, the technical details of VPA and CA were not included in the descriptions of this section, as they are not relevant to the work of this thesis.

2.4.4.1 Horizontal Pod Autoscaler

The aim of HPA is to coordinate the decrease or increase of the workload with the number of provided resources by instructing the workload controller (e.g., a deployment), to autoscale [10]. This means that if the workload decreases or the number of demanded resources decreases, HPA will instruct the workload controller to scale down the number of assigned pods leading to the removal of unused replicas/pods. If the workload increases, HPA will add to the number of existing pods. HPA is a Kubernetes API resource as well as a controller in the control plane [10]. HPA commands its workload resource to scale up/down based on metrics such as CPU utilization, or memory utilization, etc. Observed metrics can be either resource metrics or a combination of custom and external metrics. Both the targeted resources and observation metrics are defined in the configuration file.

HPA's algorithm is driven by Equation 2.1. It is important to note that HPA scales down slower than scaling up. This is due to the fact that (by default) scaling up decisions are made every 15 seconds while scaling down decisions are performed every 5 minutes (defined as the cooldown period). This cooldown period is modifiable, and its main purpose is to ensure that the client will not scale up during this cooldown period when scale down is needed. Another important consideration is that the new number of replicas will not be immediately available as scaling up decisions visit multiple control plane components before a new Pod is created.

$$N(k+1) = \left\lceil N(k) * \frac{m(k)}{M} \right\rceil \tag{2.1}$$

In Equation 2.1, $\lceil . \rceil$ is the ceiling function and

- M is the desired value of the metric
- $m(k)$ is the value of the metric during the kth interval of measurements
- $N(k)$ is the number of containers in use during the kth interval of measurements.

- *N(k + 1)* is the scaled number of containers to be in use during the $k+1$st interval of measurements.

2.4.4.2 Vertical Pod Autoscaler

VPA automatically changes the amount of resources provided for each Pod [48]. In other words, VPA autoscales CPU and memory assigned to a specific pod instead of scaling the number of assigned pods as HPA does. Initially, VPA assigns resources to pods based on historical information. After that, autoscaling of resources is achieved by restarting the targeted pod and assigning the appropriate amount of resources to that pod, thereby interrupting applications running on that pod. For example, if a pod has 1 GB of memory, but the client needs 2 GB of memory, VPA will restart that pod and assign it 2 GB of memory. In addition, unlike HPA, in which autoscaling can be conducted based on custom/external metrics, autoscaling in VPA is only determined by CPU and memory usage. The advantage of VPA is reducing the amount of excess resources that is due to static resource assignment. However, the algorithm (not presented here) is more complex and requires disrupting running applications.

2.4.4.3 Cluster Autoscaler

The third type of kubernetes scaler is CA. As the name suggests, CA autoscales the number of worker nodes instead of scaling resources or pods, thereby adjusting the cluster size [32]. Unused nodes will be removed from the cluster, and new nodes will be added when there is an inadequate number of computing resources in the cluster. For example, if a worker node's pod utilization is low, CA can schedule the pods to other worker nodes leading to a decrease in the number of existing worker nodes. However, scaling using CA only occurs when it detects the presence of unscheduled pods, which could be the result of HPA and VPA autoscalers or the deployment of a new application in the cluster.

2.5 Feedback-based Algorithms

The abstract model, shown in figure 2.4, captures the feedback characteristic of autonomic systems. The model is also known as MAPE (Monitor - Analyze - Plan - Execute) loop that is used in kubernetes to continuously monitor the system to analyze and response to the current state.

The abstract feedback model consists of four components:

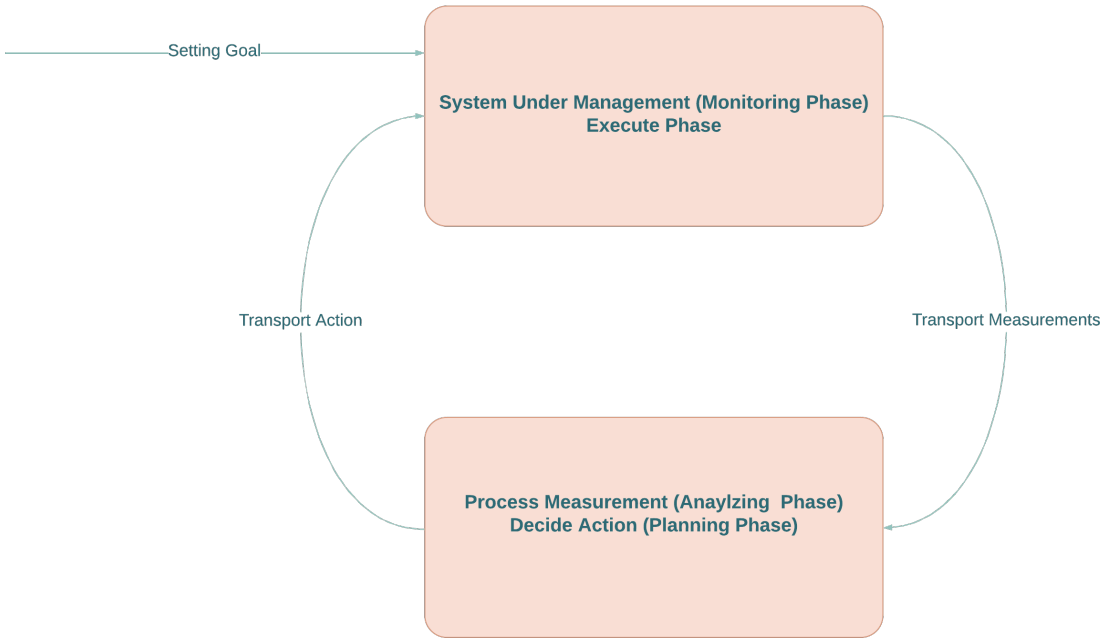


Figure 2.4: The Abstract Feedback (MAPE) Model.

Setting a Goal: The first step is setting the objectives of the system that are acquired by cloud users. This step requires answering three questions in order to formalize what the management objectives are.

- *What are the goals?* This will dictate how computational resources are monitored.
- *Who sets the goals?* Usually, this would be cloud clients.
- *When are the goals set?* Usually before the system starts operation.

  Example: before deploying the system, a cloud user specifies the target CPU utilization to be 50%. This means that the goal of the system is to ensure that the CPU utilization is kept under or equal to 50%, as long as possible, throughout the operation of the system.

System Under Management (SUM): This is the set of resources (e.g., pods) that is controlled. The current status of the SUM (with respect to the stated goal) is monitored. This step requires answering three questions in order to define the details of the monitoring phase:

- *Who monitors?* Could be cloud monitoring tools, software, humans, container orchestration.
- *What is monitored?* Depending on the goal as it could be CPU, memory, network resources.
- *How long should we monitor?* For cloud computing, continuous monitoring may be required, but sometimes the user may prefer to "sleep for a specific time" and re-monitor the computational resources (e.g., monitor only over busy periods).

Processing The Measurements: This step involves analyzing the collected, monitored information in order to determine which actions should the controller take. In general, the analysis involves some comparison of the actual status to its desired state. Note that the measurements may have to be transported to the processing location, since the SUM will not, in general, be colocated with the SUM.

- *What is needed for processing?* It depends on the autoscaling algorithms, but usually target utilization and current status that includes CPU usage and the current number of replica.

Taking Actions: This is the execution step that applies the action decided by the previous step.

- *Who decides on what actions are taken?* Based on the goal and the reactive or proactive scaling type. It could be the cloud infrastructure or container orchestration such as kubernetes.
- *Who is affected by the actions?* Usually, the action affects the monitored system (e..g, by reducing the CPU utilization) and impacts the amount that the users pay to the cloud providers.

In Figure 2.5, we depict how the generic MAPE loop is applied in kubernetes.

2.6 Related Work

This section provides an overview and discussion of some of the existing research for Kubernetes autoscalers, specifically Horizontal Pod Autoscaling (HPA), as this type of scaling is the focus of this work.

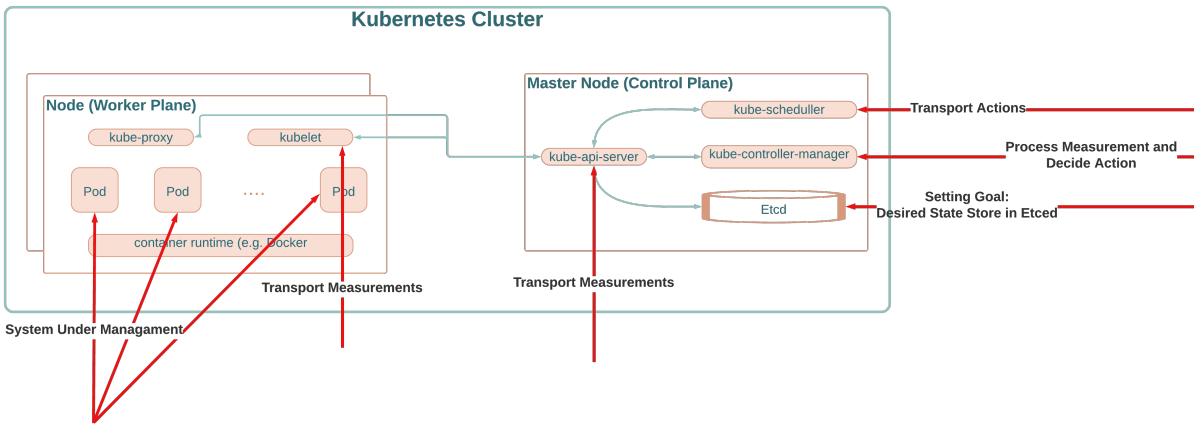


Figure 2.5: The MAPE loop in Kubernetes.

2.6.1 Platform for Dynamic Cloud Resource Provisioning

In the paper [21], a generic platform for dynamic resource provisioning in Kubernetes is introduced by three essential features for operating a distributed system on top of an orchestration platform, e.g. Kubernetes.

- **Comprehensive monitoring:** The platform takes the system computational resources and QoS metrics into account, which provides better provisioning decisions. Therefore, the monitoring stack is based on four steps of dynamic resource provisioning as follows:

  1. *Heapster* for collecting low-level system metrics
  2. *Apache JMeter* for generating application load and measuring response time
  3. *InfluxDB* for time-series database for storing metrics
  4. *Grafana* as a visualization tool

- **Deployment flexibility:** The separation of tasks into the four-piece stack helps the users to easily replace an existing algorithm without affecting other modules.

- **Automatic operation:** This feature automatically applies the resource provisioning operation.

The *Resource Scheduler Module* and *Pod Scaler Module* was described that modify the replica number based on CPU usage with static thresholds. The application cannot directly set these thresholds as the cluster operator is needed to set these thresholds.

2.6.2 KHPA-A

The focus of [19] and [20] were on CPU- intensive utilization. The authors investigated the relation between relative and absolute CPU-intensive usage in micro-services autoscaling. Kubernetes Horizontal Pod Autoscaling algorithm calculates the relative CPU utilization to provide the desired number of containers that maintain the desired threshold for CPU utilization. Relative utilization is the average CPU usage reported as a percentage that is exposed through the control groups (cgroups). Moreover, the authors found that relative usage cannot determine the required amount of resources needed as it underestimates the required capacity. Therefore, they introduced the KHPA-A algorithm that is similar to the default HPA, but utilizes absolute CPU metrics for CPU-intensive workloads by taking into account two correlation coefficients as shown in equation 2.2.

$$U_{absolute} = b + a \times U_{relative} \tag{2.2}$$

In Equation 2.2,

- b and a are the correlation coefficient
- $U_{relative}$ is the average of CPU utilization value during an interval of measurements.

The paper compared the performance of default HPA and KHPA-A under four workloads, sysbench and stress-ngd. KPHA-A's pod scaling number is 10% higher compared to HPA. Yet, it has been found that the performance of KPHA-A is better than regular HPA for different use cases with single or multi-containers as KPHA-A responded faster than HPA by 50% to application workload in single container use cases. In addition, the response time of HPA was triple that of KPHA-A in multi-containers workloads.

2.6.3 Microscaler

Microscaler [50] is a horizontal autoscale algorithm that identifies the need for scaling services to meet the service level agreement (SLA). *Microscaler* combines an online learning approach with a heuristic approach to provide an optimal scaling cost and maintain QoS. In the paper, the authors establish a service power criterion that determines the necessity for scaling when the system is under-or over-provisioned. Service power is the ratio between P50 and P90 as shown in equation 2.3.

$$Service\ Power = \frac{P50}{P90} \tag{2.3}$$

In Equation 2.3,

- *P50* is the average latency of the slowest 50% during the last 30 seconds.
- *P90*is the average latency of the slowest 10% during the last 30 seconds.

Therefore, if the result of the service power is above or equal to one, the system QoS level is managed as the system can handle the incoming requests. However, if the service power is below one, the QoS is low. This means that *Microscaler* only scales when the SLA is violated even if there is high CPU utilization or increased response time. Furthermore, the results presented in the paper are vague as the difference between *Microscaler* and other autoscaling approaches, such as *Amazon Emulated auto-scaler*, are not significant.

CHAPTER

3

PROBLEM STATEMENT AND PROPOSED SOLUTION

In this chapter we state the problem and propose three algorithms for its solution. More specifically, in Section 3.1, we provide a clear explanation of the problem including the research questions and objectives. In Section 3.2, we describe the system model. In Section 3.3, we list four specific metrics for the evaluation of the three algorithms we propose in Section 3.4.

3.1 Problem Statement

Autoscaling is a feature that the cloud utilizes to dynamically adjust the number of computational resources based on the system load. The workload may change dramatically throughout the day, so an autoscaling algorithm increases or decreases the number of resources to (hopefully) match the needs of the changing traffic. The main problem with the autoscaling algorithm is not adding or removing resources, but rather the speed of the algorithm's response to meet traffic changes to maintain scalability, reliability, and

cost-effectiveness. For example, in many cases, there are short bursts of load that can affect Quality of Service (QoS). They can trick the autoscaling algorithm into over scaling. It takes time to provision and create a new replica of a resource, and the short-burst may not exist by that time. This issue results in sub-optimal cost efficiency: customers may spend money on resources that are not used.

3.1.1 Research Questions

This thesis attempts to answer the following three specific questions:

- **RQ1: suppose we use CPU utilization measurements. Can the incorporation of previous scaling decisions and measurements into the scaling algorithm meet the required performance and reduce over or under provisions?**

  The algorithm should identify the required measurements (e.g., how many past measurements) and past decisions to be involved. We have opted to investigate three possible implementations.

- **RQ2: compare the three algorithms against each other with respect to under- and over-provisioning. Moreover, compare with Kubernetes' HPA.**

  Evaluating the proposed feedback algorithm is a critical step to verify its effectiveness. Therefore, we have selected two realistic datasets with different loading patterns for this evaluation. We describe them in detail in Section 4.1, page 40.

- **RQ3: how to monitor the incoming workload?**

  THe monitoring phase is the primary step to establish the feedback loop in autoscaling. In this thesis, we will consider only a low-level metric, specifically CPU utilization. We do this for two reasons. The datasets we employ for utilization do not provide information about connections; moreover, CPU utilization can be observed through built-in monitoring tools in Kubernetes (connections would need more advanced monitoring tools).

3.1.2 Research Approach

We propose the following approach for answering the research questions of the previous section.

- **(1) Develop multiple feedback-based algorithms that take into account past decisions and measurements (aims to answer RQ1)**

  The proposed feedback algorithms aim to be self-adaptive to CPU usage. The intuition (to be checked) and hope is that by incorporating past measurements as well as past decisions will "improve performance". The challenge we faced was the use of "clean" CPU measurements, as we explain later.

- **(2) Reduce oscillatory behavior for both up and down decisions (aims to answer RQ2)**

  An alternative would have been to focus only on reducing only up or only down "mismatches". We have chosen to evaluate the behavior of our algorithms using real datasets.

- **(3) Use the built-in kubelet monitoring for CPU utilization (aims to answer RQ3)**

  The kubelet monitoring tool inside each pod checks (a running) pod continuously and supplies an (inaccurate) load measurement to the metrics server. The inaccuracy comes from a variety of reasons. For example, even if we set a measurement period of 1 minute, kubelet may report a measurement over 50 seconds or 67 seconds. With, say, 10 pods running, we may get 10 measurements taken over 10 different intervals. Even though the kubernetes documentation says that the average utilization reported for the 10 pods is weighted, the implementation does not employ a weighting formula[1]. The rationale behind this is that the CPU utilization is not calculated accurately in the first place; moreover, the intuitive tendency of HPA is to "correct" such inaccuracies with the next decision. So the tradeoff made in kubernetes was in favor of simplicity over accuracy.

3.2 System Model

In this section, we explain the proposed system design in depth.

[1] It took us a good amount of time searching the code to discover this; that was painful but valuable experience...

3.2.1 System Design Components

There are three main system components, as illustrated in Figure 3.1, that bring the measurements from a pod in the worker plane to the control algorithm in the master plane through the metrics server.

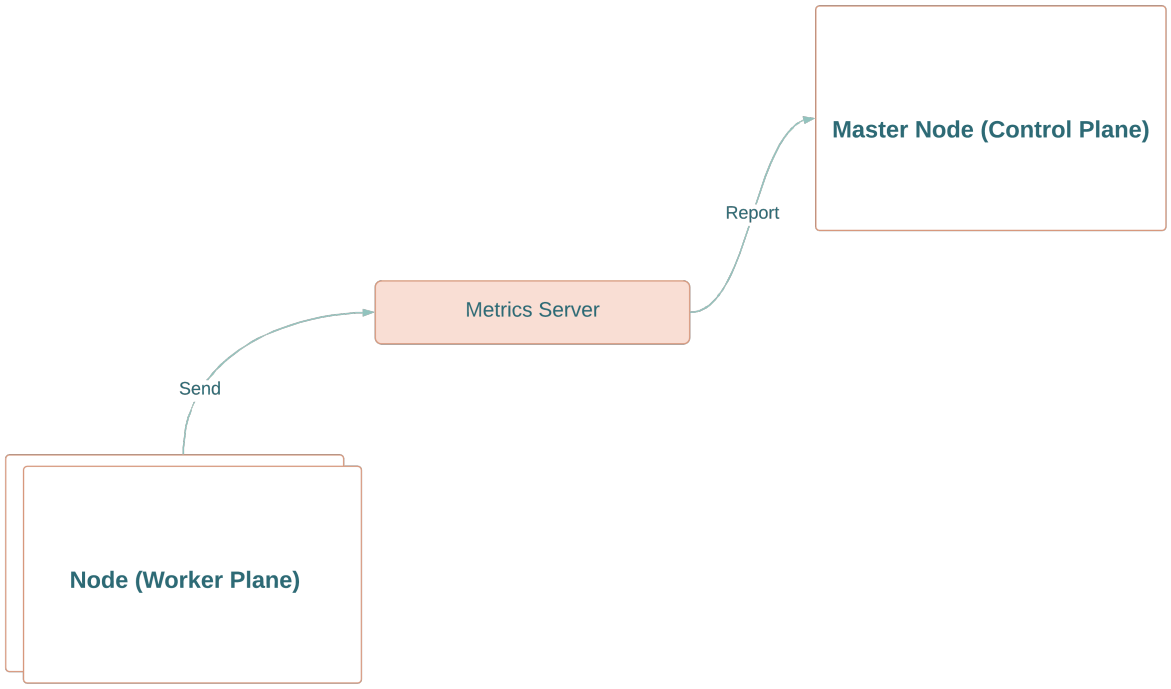


Figure 3.1: The main system design components in a Kubernetes cluster.

3.2.1.1 K8s Control Plane

The control plane in kubernetes is responsible for making decisions such as scheduling and managing active (running, error-free) pods. In this thesis, we focus on the two sub-components, kube-controller-manager and Kube-api-server.

3.2.1.2 K8s Worker Plane

The worker plane in Kubernetes hosts pods and manages them following the control plane's instructions. In this thesis, the components of interest are only the kubelet, pods, and

Docker runtime.

3.2.1.3 Metrics Server

The metrics server component collects pod measurements from kubelet through the kube-api-server. This collection follows a push model, as a result of which only the most recent measurement is stored. It is the responsibility of whoever wants to process the measurement to "pull it in time". This architecture created unforeseen issues for us, regarding coding for history-based algorithms, that are not present in algorithms that do not utilize history.

More specifically, suppose that you have a control algorithm that uses two consecutive measurements M_1, M_2 in making its decision. It is the responsibility of the algorithm to pull them from the metrics server. In an ideal environment in which the algorithm, the kubelets and the metrics server are fully synchronized, pulling is periodic and there is no problem. However, there is randomness in when the kubelets report and when the algorithm pulls. If we do not want to miss a measurement, teh algorithm should pull with a frequency higher than the reporting frequency of the kubelets.

This discrepancy in the frequency creates duplicate measurements, that must be discarded in the processing of a history-based algorithm[2].

3.2.2 MAPE Loop

The proposed system model is based on the Monitor-Analyze-Plan-Execute (MAPE) loop; we have seen it in Section 2.5, page 16. MAPE loop is considered to be a self-adaptive feedback loop in software systems. Figure 3.2 shows the four phases that are part of the loop, specialized for a kubernetes-controlled system (and using a goal of 50% CPU utilization as an example). Each stage has its own role and purpose in achieving the system's goal. The various phases are described as follows:

3.2.2.1 Monitor Phase

Metrics server continuously stores information from the worker and control planes that fully characterizes the state of the deployment at a given time. There are two different types of raw measurement data. The first is pod-related information reported by the kubelet(s); examples are metrics timestamp, window size, and CPU utilization. The second is data obtained from the kube-api-server that contains information about all running nodes;

[2]It was not easy to make this realization at first.

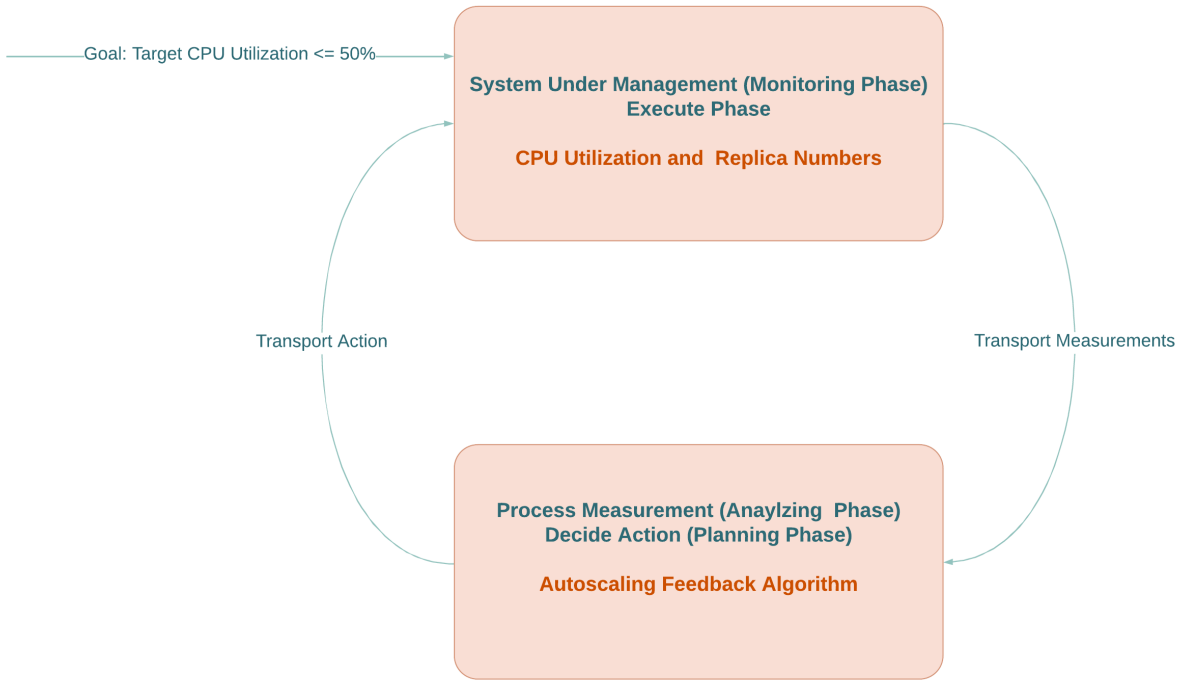


Figure 3.2: Kubernetes HPA as a MAPE Loop implementation.

examples are the number of current replicas, previous decisions or measurements, and the number of desired replicas. Users can configure parameters such as reporting duration and frequency of decisions.

3.2.2.2 Analyze Phase

In this phase, the collected information is analyzed in the autoscaler (e.g., one of the three algorithms we propose later in this chapter) to determine if any change in the number of running pods (replicas) is required. Since the proposed autoscaler is proactive, it will predict future CPU usage by utilizing historical data (i.e. previous scaling decision or measurements). We will discuss details in Section 3.4.

3.2.2.3 Plan Phase

The Analysis phase evaluates the new current state, so the planning phase calculates the total number of replicas required to achieve the goal, and decides whether scale up or scale down is required.

3.2.2.4 Execute Phase

In this phase, the new desired state, determined in the planning phase, will be implemented. kube-api-server is responsible for executing the desired change by passing the new information to a kubelet through the metrics server. Kubelet updates the running pods by scaling up or down based on the new instruction. Note that the implementation of this decision may take some (random) time; this time is not controllable.

3.2.3 System Model Flow

There are eleven sequential steps that the system goes through in order to implement a scaling decision. Figure 3.3 depicts them.

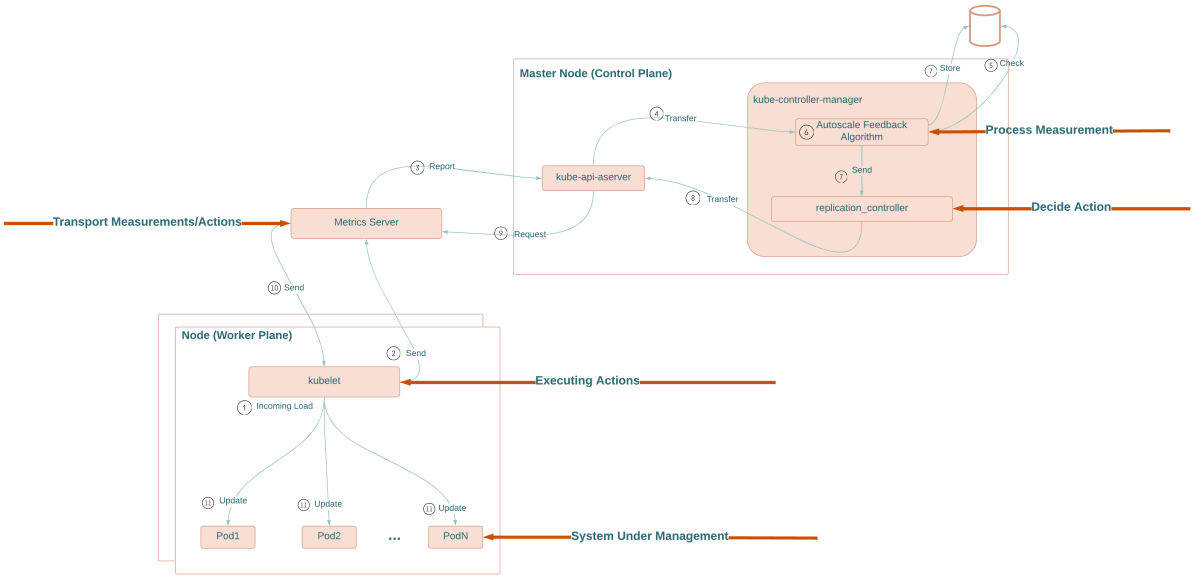


Figure 3.3: Flow of steps, from monitoring to a decision taking effect.

1. Each Kubelet in Kubernetes worker plane gathers the current CPU utilization from the running pods. This measurement is (slightly?) inaccurate and not synchronized across all pods.

2. Each Kubelet sends its measurement to the metrics server. There is unavoidable variation in the time the measurements are received.

3. Metrics server stores the measurements sent by all the pods. Metrics server reports the current utilization to kube-api-server in kubernetes control plane.

4. Kube-api-server transfers the pods' measurements and the current replica based on the deployment status to kube-controller-manager as inputs for the autoscale feedback algorithm.

5. Auto-scale feedback algorithm "checks" (e.g., cleans up) the previous data based on the autoscaling algorithm.

6. At this time, the auto-scale feedback algorithm calculates the desired state based on collected inputs.

7. The feedback algorithm sends the calculated desired state to the replication-controller to check the policy and stores the data in its historical database.

8. The Replication-controller transfers the desired state to kube-api-server (to communicate with the worker plane so all nodes can update their status).

9. Kube-api-server sends the new instructions to metrics server.

10. Metrics server transfers the new instructions to kubelet(s).

11. Each Kubelet manages and updates its pod based on the new instructions.

3.3 Metrics Selection

The following four are the criteria that will be considered for evaluating the proposed algorithm**s**:

1. Reduce/Minimize the duration of time during which a user is running on low resources. The Under-Provisioning Time, shown in Equation 3.3, is the quantitative measure used in this criterion.

2. Reduce/Minimize the duration of time during which a user is using more resources than needed. The Over-Provisioning Time, shown in Equation 3.4, is the quantitative measure used in this criterion.

3. Allocate the minimum amount of resources needed to achieve a stated CPU utilization goal. This can be captured by the Under-Provisioning Accuracy, shown in Equation 3.1 and the Over-Provisioning Accuracy, shown in Equation 3.2 metrics.

The ideal, optimal value for all these metrics is zero. The computations in the equations below have been defined by the Research Group of the Standard Performance Evaluation Corporation (SPEC) [27].

- **Provisioning Accuracy Metrics**

  Provisioning accuracy metrics calculate θ_U, the under-provisioning accuracy that defines the number of missing resources that are required, and θ_O, the over-provisioning accuracy that defines the number of unused resources during the time interval T. The metrics are expressed as percentages.

$$\theta_U[\%] = \frac{1}{T}.\sum_{k=0}^{K-1} \frac{\max(N(k+1)-N(k),0)}{N(k+1)} \Delta T_k \times 100 \tag{3.1}$$

$$\theta_O[\%] = \frac{1}{T}.\sum_{k=0}^{K-1} \frac{\max(N(k)-N(k+1),0)}{N(k+1)} \Delta T_k \times 100 \tag{3.2}$$

- **Provisioning Timeshare Metrics** Provisioning timeshare metrics (τ_U) and (τ_O) measure the total duration of time when the system is under-provisioned and over-provisioned, respectively, during the experiment time. These metrics capture whether the deviation between request and provide measurements is big or small. The metrics are expressed as percentages.

$$\tau_U[\%] = \frac{1}{T}.\sum_{k=0}^{K-1} \max(sgn(N(k+1)-N(k),0)\Delta T_k \times 100 \tag{3.3}$$

$$\tau_O[\%] = \frac{1}{T}.\sum_{k=0}^{K-1} \max(sgn(N(k)-N(k+1),0)\Delta T_k \times 100 \tag{3.4}$$

In Equations 3.1-3.4:

- T is the total experiment duration. It is split into K non-overlapping subintervals of duration ΔT_k. In our experiments, we took all of them to be of equal length.

- *N(k)* is the number of running replicas (containers) during the *k*th interval of measurements.
- *N(k+1)* is the requested number of replicas to be in use during the *k+1*st interval of measurements.
- *sgn* is the signum function.

Note that for any subinterval, only one of the terms $\max(N(k+1)-N(k),0), \max(N(k)-N(k+1),0)$ will be nonzero. In the ideal scenario of exact provisioning, both should be zero. Smaller values of the metrics are better.

3.4 Proposed Algorithms

In this section, we propose three autoscaling algorithms that consider history. All algorithms follow the concept of Horizontal Pod Autoscaler (HPA), described in Section 2.4.4.1.

3.4.1 Algorithm 1: One-step history

Algorithm 1 is our starting point; it is a simple scaling algorithm that takes into account only one, the most recent decision when it calculates the desired number of replicas for the next time interval. The goal of Algorithm 1 is to reduce high fluctuation by taking into account one previous decision D_{k-1}, to scale up or down.

3.4.1.1 Algorithm Description

As previously discussed in Sections 3.2.2 and 3.2.3, the kubelet in the worker plane measures the CPU utilization and sends it through the metrics server and the kube-api-server to Algorithm 1. The kube-api-server passes all the inputs mentioned in Algorithm 1, in order to calculate n_k, the desired number of replicas at the current period, and of course the scaling decision D_k.

For simplicity, the initial values needed are set as follows: $n_{k-1} = 1$, $D_{k-1} =$ UP. A typical value for U in our experiments is 50%.

Before calculating the desired number of replicas, the algorithm should calculate U(k), the average CPU utilization during the current period $[T_{k-1}, T_k)$ (as a percentage). The

Algorithm 1 One-step history

Input:

I: Goal U; ▷ The desired threshold for CPU utilization.

I: Utilization $U(k)$; ▷ CPU utilization during the current period $[T_{k-1}, T_k)$.

I: Decision instant T_k; ▷ The current decision instant.

I: Decision D_{k-1}; ▷ The Up/Down decision made at the previous instant T_{k-1}.

I: n_{k-1}; ▷ The number of containers requested at the previous instant T_{k-1}.

Output:

O: Decision D_k; ▷ The decision made at the current instant T_k.

O: n_k; ▷ The desired number of containers at the current period.

Description:

At T_k, input $U(k)$
if $U(k) > U$ **then** ▷ Goal not met, scale up
    if $D_{k-1} == Up$ **then** ▷ Scaled up before
        $n_k = \lceil n_{k-1} \cdot \frac{U(k)}{U} \rceil$ ▷ scale up more vigorously
        $D_k = Up$
    else ▷ Scaled down before
        $n_k = n_{k-1} + 1$ ▷ scale up lightly
        $D_k = Up$
    end if
else ▷ Goal is met, scale down
    if $D_{k-1} == Up$ **then** ▷ Scaled up before
        $n_k = n_{k-1} - 1$ ▷ scale down lightly
        $D_k = Down$
    else ▷ Scaled down before
        $n_k = \lceil n_{k-1} \cdot \frac{U(k)}{U} \rceil$ ▷ scale down more vigorously
        $D_k = Down$
    end if
end if

average is over all pods that were active during the current period. This is done by using the following equation:

$$U(k) = \frac{\sum_n U_n \times baseMilicore}{requestValue \times numPods} \times 100 \tag{3.5}$$

In Equation 3.5,

- the summation is over all active pods n during this period;
- U_n is the CPU utilization of pod n during in time interval $[T_{k-1}, T_k)$; note that the utilization is reported in the standard kubernetes unit of cores, not percentage;
- *baseMilicore* converts the CPU utilization to milicore by multiply by 1000.
- *requestValue* is the minimum amount of resources that containers need.
- *numPods* is total number of pods that were collected during the period $[T_{k-1}, T_k)$.

After calculating the total CPU utilization $U(k)$, Algorithm 1 compares it with the target utilization U yielding the four cases as Figure 3.4 illustrates.

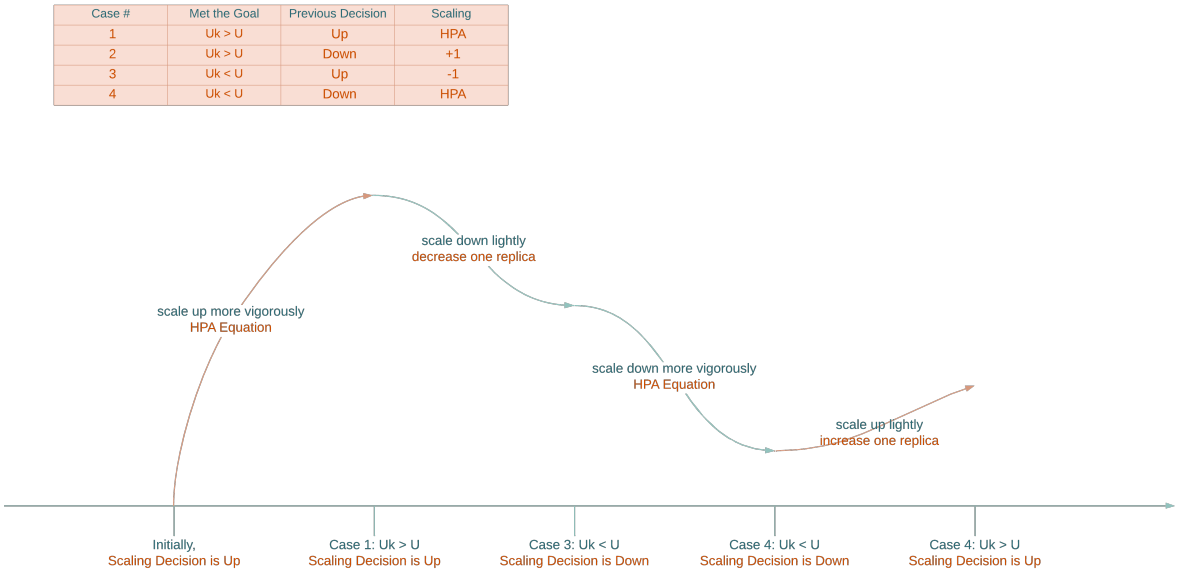


Figure 3.4: Algorithm 1 Explanation.

If the goal utilization is not met as $U(k)$ is greater than U, the algorithm must scale up. The algorithm checks the previous decision D_{k-1} that has been stored inside the database as shown in step 5 in Figure 3.3. If the previous decision was UP, the autoscaling algorithm will scale up more aggressively, by using the HPA Equation 2.1. The reason for scaling up vigorously is that we predict that the workload is increasing in the next interval, so we increase the resource to maintain adequate performance. However, if the previous decision was to scale down, the algorithm lightly scales up by only one replica.

On the other hand, the algorithm must scale down if $U(k)$ is less than U. The scaling algorithm scales down lightly or aggressively based on the last decision. If the previous decision was to scale up, the algorithm lightly scales down by decreasing the number of replicas by one as there is a high chance that in the next time interval the workload is decreasing. However, if the previous decision was to scale down, the algorithm scales down by a bigger amount using the HPA equation.

3.4.2 Algorithm 2: Rolling Averages

Algorithm 2 is our next suggestion. It does not utilize the previous decision; instead, it uses a rolling average of previous measurements. Recall that the rolling average of a quantity X is a metric that utilizes the past L measurements $M_{i-L}, M_{i-L+1}, \ldots, M_{i-1}$, in order to calculate the average at time i. The intuition (and hope) then is that this averaging (acting as a low-pass filter) will smooth out short-term fluctuations in the workload.

At every decision instant, Algorithm 2 uses this average (see lines 7 and 8 in the algorithm pseudocode).

3.4.2.1 Algorithm Description

As an example, suppose that the last five measurements are used in the calculation; the required initial values are set as follows: $n_{k-1} = 1$, $M[0..4] = 0,0,0,0,0$. Measurements are taken every one minute.

The monitoring phase will collect the total CPU utilization and number of pods. In other words, the measurements that the algorithm stores are the average utilization during the previous five minutes. Fore example, at minute 7, the algorithm utilizes the latest five minute (M3, M4, M5, M6, and M7) values. However, at the first four decision instants, the algorithm should consider rolling 1, 2, 3, and 4 measurements only.

Then, the analysis phase calculates the rolling average using equation 3.6.

Algorithm 2 Rolling Averages

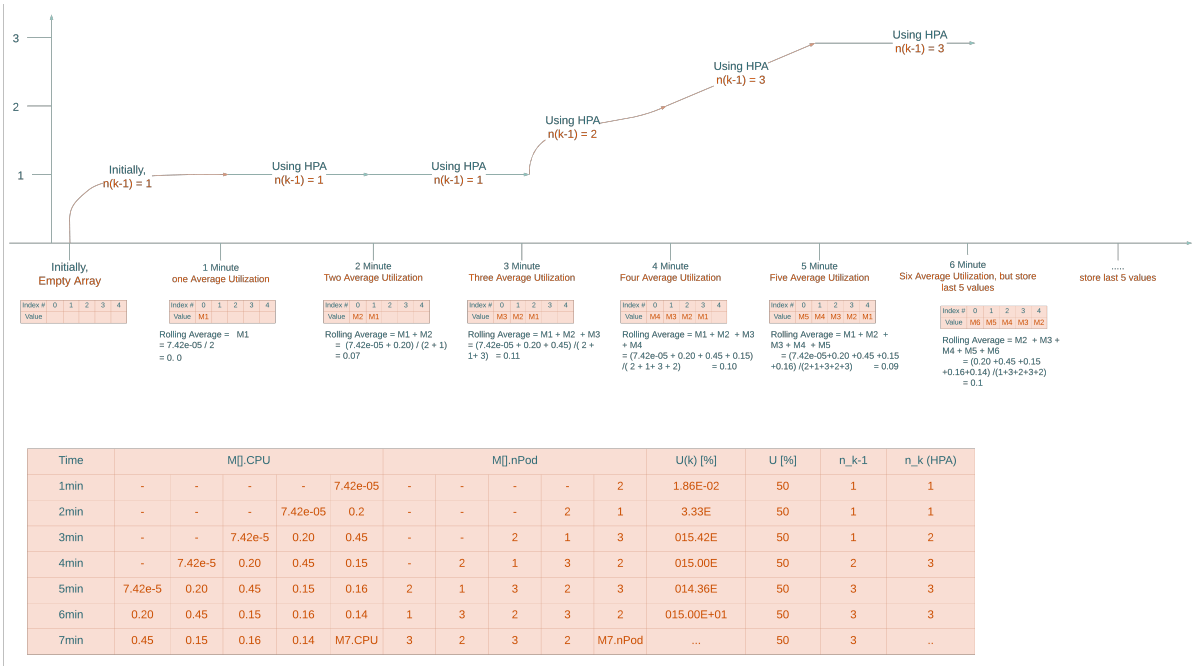
Input:

I: Goal U; ▷ The desired threshold for CPU utilization.
I: Utilization $U(k)$; ▷ CPU utilization during the current period $[T_{k-1}, T_k)$.
I: Decision instant T_k; ▷ The current decision instant.
I: n_{k-1}; ▷ The number of containers requested at the previous instant T_{k-1}.
I: $M[0..M_c].CPU$; ▷ An array store five measurements of CPU utilization.
I: $M[0..M_c].nPods$; ▷ An array store five measurements of number of pods.
I: U_t; ▷ The total average utilization during the previous five measurements.
I: P_t; ▷ The total number of pods during the previous five measurements.
I: M_c; ▷ Number of measurement we take into account for scaling.
I: **Output:**
O: n_k; ▷ The desired number of containers at the current period.

Description:

1: At T_k, input $U(k)$
2: $M[k] = U(k)$
3: **for** i : 0 to M_c **do**
4: $U_t = U_t + M[i].CPU$
5: $P_t = P_t + M[i].nPods$
6: **end for**
7: $U(k) = \frac{U_t}{P_t}$
8: $n_k = \lceil n_{k-1} \cdot \frac{U(k)}{U} \rceil$

$$5-minute\ Rolling\ Average = \frac{\sum_{i=k-5}^{k} U_i}{\sum_{i=k-5}^{k} P_i} \tag{3.6}$$



Time	M[].CPU					M[].nPod					U(k) [%]	U [%]	n_k-1	n_k (HPA)
1min	-	-	-	-	7.42e-05	-	-	-	-	2	1.86E-02	50	1	1
2min	-	-	-	7.42e-05	0.2	-	-	-	2	1	3.33E	50	1	1
3min	-	-	7.42e-5	0.20	0.45	-	-	2	1	3	015.42E	50	1	2
4min	-	7.42e-5	0.20	0.45	0.15	-	2	1	3	2	015.00E	50	2	3
5min	7.42e-5	0.20	0.45	0.15	0.16	2	1	3	2	3	014.36E	50	3	3
6min	0.20	0.45	0.15	0.16	0.14	1	3	2	3	2	015.00E+01	50	3	3
7min	0.45	0.15	0.16	0.14	M7.CPU	3	2	3	2	M7.nPod	...	50	3	..

Figure 3.5: Algorithm 2: 5-minutes Rolling Averages Explanation

Figure 3.5 shows that the algorithm stores the average utilization in an array every one minute. For instance, the total CPU U_t is 7.422e-05 cores and the total number of pods P_t is 2 in once minute. Therefore, the average utilization during this time interval is $\frac{7.422e^{-05}}{2} = 0.00003711$. While at four minutes, there will be four measurements as follows:

- for minute 1, $U_t = 7.422e^{-05}$ and $P_t = 2$.
- for minute 2, $U_t = 0.203781431$ and $P_t = 1$.
- for minute 3, $U_t = 0.451052744$ and $P_t = 3$.
- for minute 4, $U_t = 0.14972116600000002$ and $P_t = 2$.

In this case, the rolling average is the sum of the previous four CPU utilization measurements divided by the total number of pods during the first four minutes.

Once the analysis phase calculates the rolling average, the planning phase uses it as the value for $U(k)$ in line 7 of the pseudocode, when it calculates n_k, the desired number of replicas for the current period T_k, in line 8.

3.4.3 Algorithm 3: Moving Window Averages

This algorithm is a slight variation of the previous one; instead of rolling, it uses moving window averages. Since an average is taken, we expect to have a low-pass effect as well. The difference with the previous algorithm is that with moving averages, we "forget" the history before the current window. The motivation for studying this algorithm is curiosity.

We'll use again an example of a 5-minute window, with measurements taken every one minute. Then the moving window average reduces five individual, one-minute measurements into one. The algorithm takes a decision every 5 minutes. Any workload fluctuations with the five-minute interval will be absorbed.

The idea is described as Algorithm 3.

3.4.3.1 Algorithm Description

The proposed system gathers five one-minute measurements that will be utilized to calculate a 5-minute window average and will be considered as input for the HPA equation. This algorithm needs at least 5 consecutive minutes of measurements to calculate the window average. For example, let's assume that the following measurements were collected every minute.

- for minute 1, $U_t = 7.422e^{-05}$ cores and $P_t = 2$.
- for minute 2, $U_t = 0.203781431$ cores and $P_t = 1$.
- for minute 3, $U_t = 0.451052744$ cores and $P_t = 3$.
- for minute 4, $U_t = 0.14972116600000002$ cores and $P_t = 2$.
- for minute 5, $U_t = 0.159225618$ cores and $P_t = 3$.

The 5-minute moving window average equation is similar to the rolling average formula in Equation 3.6. However, instead of rolling the values by taking into account the latest 5 minutes measurements, the 5-minutes window average will consider 5 minutes measurements every 5 minutes.

Algorithm 3 Moving Window Averages

Input:

I: Goal U; ▷ The desired threshold for CPU utilization.
I: Utilization $U(k)$; ▷ CPU utilization during the current period $[T_{k-1}, T_k)$.
I: Decision instant T_k; ▷ The current decision instant.
I: n_{k-1}; ▷ The number of containers requested at the previous instant T_{k-1}.
I: $M[0..M_c]$; ▷ An array store five measurements.
I: U_t; ▷ The total average utilization during the previous five measurements.
I: M_c; ▷ Number of measurement we take into account for scaling.

Output:

O: n_k; ▷ The desired number of containers at the current period.

Description:

At T_k, input $U(k)$
if $M_c = 5$ **then** ▷ Autoscaling algorithm applies every 5 minutes
    $M[k] = U(k)$
    for i : 0 to M_c **do**
        $U_t = U_t + M[i].CPU$
        $P_t = P_t + M[i].nPods$
    end for
    $U(k)[\%] = \frac{U_t}{M_c}$
    $n_k = \lceil n_{k-1} \cdot \frac{U(k)}{U} \rceil$
else ▷ Collecting and storing CPU utilization
    $M[k] = U(k)$
end if

$$\text{5-minute window average} = \frac{\sum_{i=k-5}^{k} total\ CPU\ usage}{\sum_{i=k-5}^{k} total\ pods\ number}$$

$$= \frac{7.422e^{-05}+0.203781431+0.451052744+0.14972116600000002+0.159225618}{2+1+3+2+3}$$

$$= \frac{0.963855179}{11}$$

$$= 0.08762319809$$

After calculating the 5-minute average, the system passes the information as an input to the autoscaling algorithm, HPA. The desired number of containers at the current period T_k considers the 5-minute window average utilization as $U(k)$, n_{k-1}, and U. The decision instants are minutes $T_k = 5$, 10, 15, 20, etc.

CHAPTER

4

EVALUATION AND RESULTS

In this chapter, we evaluate the proposed algorithms presented in section 3.4. First, in Sections 4.1 and 4.2, we describe the workload and experimental setups that were used to evaluate the performance of the algorithms. Then, in Section 4.3, we first discuss details of experiments in order to explain the steps and challenges we faced in implementing the proposed algorithms; we then present our evaluations.

4.1 Workload

In this section, we consider two different real-time workloads, namely the traces from the FIFA World Cup 98 web servers and a National Aeronautics and Space Administration (NASA) web server.

4.1.1 FIFA World Cup 98 Web Servers

This dataset contains 1,352,804,107 requests, logged from 30 April 1998 to 26 July 1998, almost three months worth of data. It has a high degree of variation in its values and has peaks that are difficult to predict. This dataset shows every day in a separate row with two

elements, creation time and number of bytes the server has responded with. For example, `5/1/1998 22:00 | 352` .

In our experiments we selected two subsets with only six hours worth of data in each. Each experiment of ours runs in real, not simulated time, so using the entire dataset would be unrealistic. We scaled the number of bytes by a factor of 100,000. This was done in order to create proper utilization levels at our CPUs. The first subset was collected from the number of requests on day *May 1st, 1998* as shown in Figure 4.2. The other subset was randomly collected from the entire FIFA dataset as illustrated in Figure 4.3.

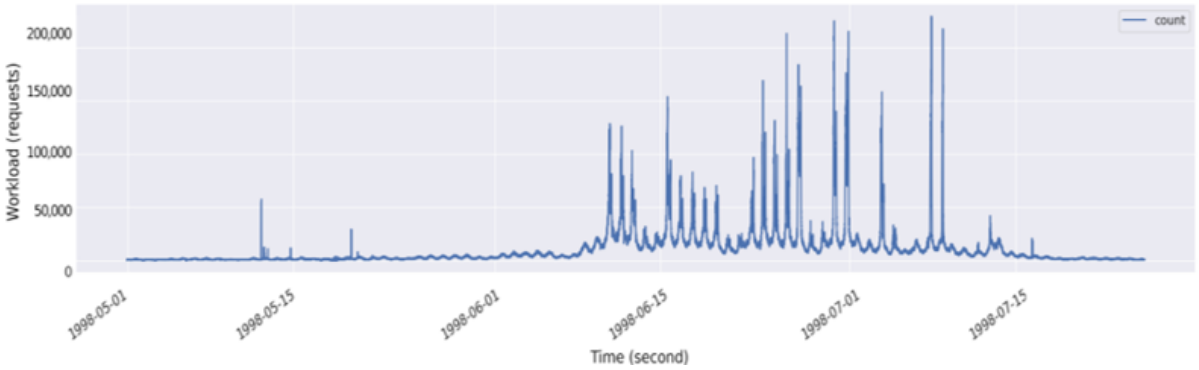


Figure 4.1: FIFA World-Cup Dataset, three months.

4.1.1.1 NASA web server

NASA Web server logs are publicly available; the one we used contains collected HTTP requests sent to the NASA Kennedy Space Center Web server in Florida. There are two months worth of datapoints, from 1 July 1995 to 31 August 1995. However, there were no logs during the period 1 August 1995 (11:59:59 PM) until 3 August 1995 (04:36:13) due to a server lock down. There were 1,891,714 and 1,569,898 datapoints in the July and August periods, respectively. In total, the two logs contain 3,461,612 HTTP Requests. Moreover, the NASA-HTTP workload includes three different types of HTTP requests: GET, POST, and HEAD with eight status codes (e.g. 200, 302, 304, 400, 403, 404, 500, and 501).

There is no specific pattern, but each HTTP request is logged as one row with six columns as follows:

`199.72.81.55 - - [01/Jul/1995:00:00:01 -0400] "GET /history/apollo/ HTTP/1.0" 200 6245`

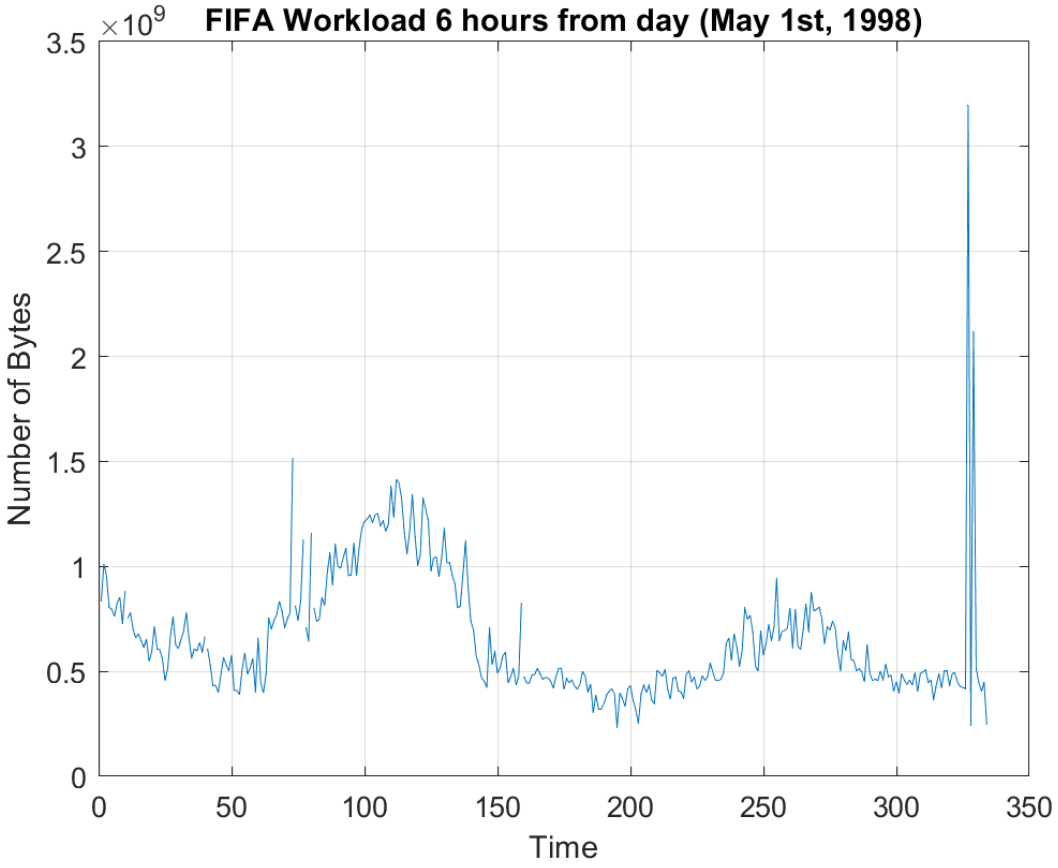


Figure 4.2: Subset of six hours of FIFA World-Cup Dataset on day May 1st, 1998.

- IP address or host name who is the requested client; e.g., `199.72.81.55`.
- Timestamp indicating when processing the request started and its format "Day/Month/Year:Hour:Minute:Second -Timezone"; e.g., `[01/Jul/1995:00:00:01 -0400]`.
- HTTP request type that includes path and protocol version;. e.g., `"GET /history/apollo / HTTP/1.0"`.
- HTTP status code; e.g., `200`.
- The size of reply content to the client in bytes; e.g., `6245`.

The original dataset has been pre-processed by aggregating all rows that occur in the same minute into one cumulative number, so NASA dataset shows the total workload for each minute. Similar to FIFA dataset, we are considering two subset of six hours from July 1st, 1995 as illustrated in Figures 4.5 and 4.6, and multiply the number of bytes by 100.

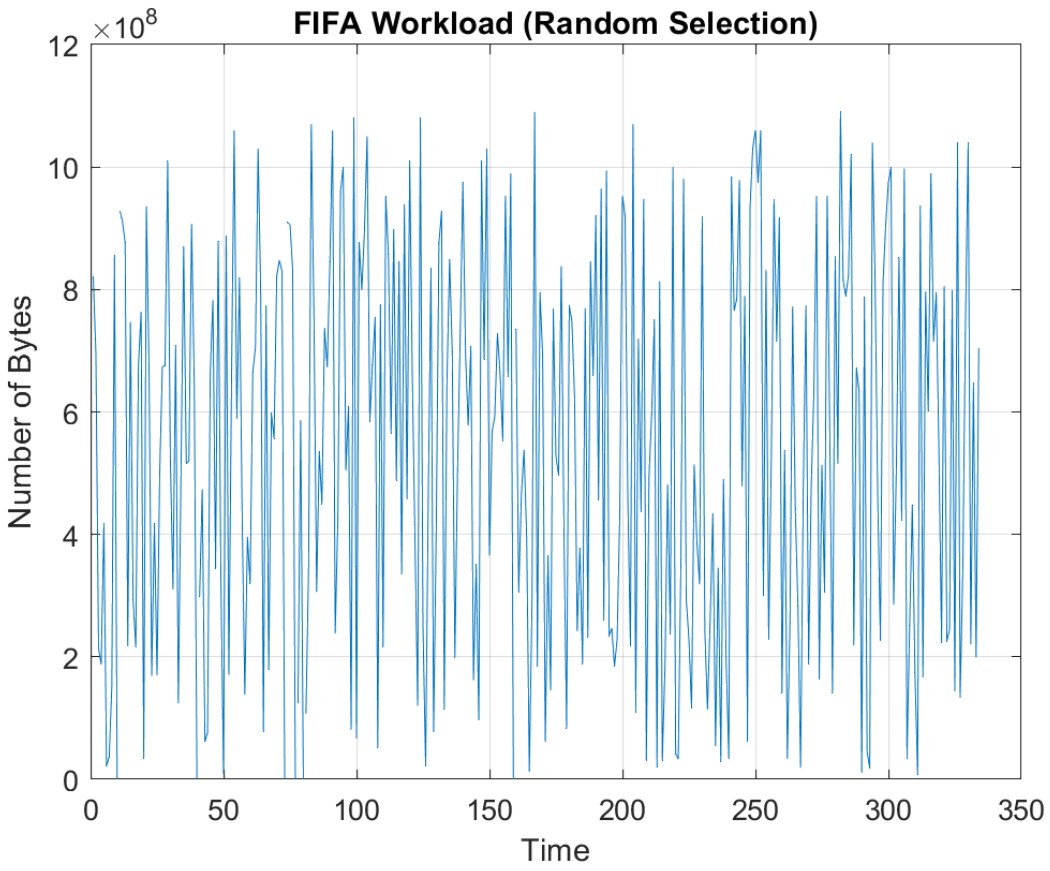


Figure 4.3: Subset of six hours of FIFA World-Cup dataset (random selection).

HTTP Request Types	GET	POST	HEAD
Number of Requests	3,453,473	222	7,917

4.1.2 Load Generator Setup

One of thesis's objective is to evaluate the proposed auto-scaling algorithm performance when the system faces real-time workload fluctuations. To generate a load inside the kubernetes cluster, we need to follow three steps.

Step 1: Reading The Number of Requests The aim of using load generator image is to setup the php-apache development environment related to PHP file based on web apps. The first step to generate a load inside kubernetes is creating two shell scripts to read the records data from NASA-Dataset.csv and FIFA-Dataset.csv. In the below script, we used the read command to read a line from the CSV file and sleep six seconds to read the next

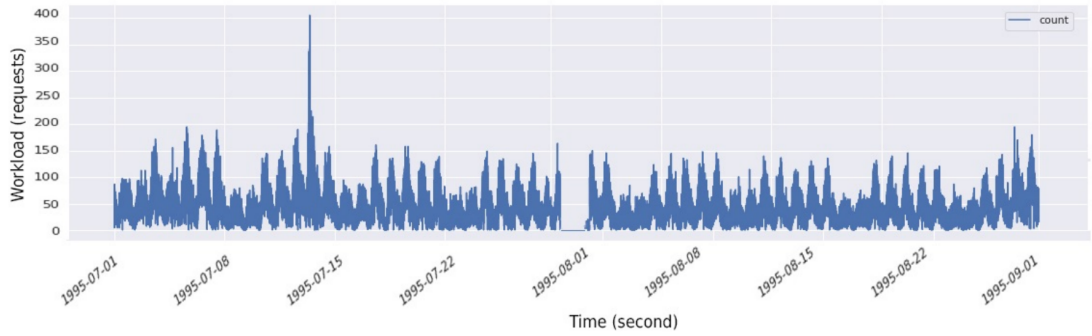


Figure 4.4: NASA-HTTP Workload, two months.

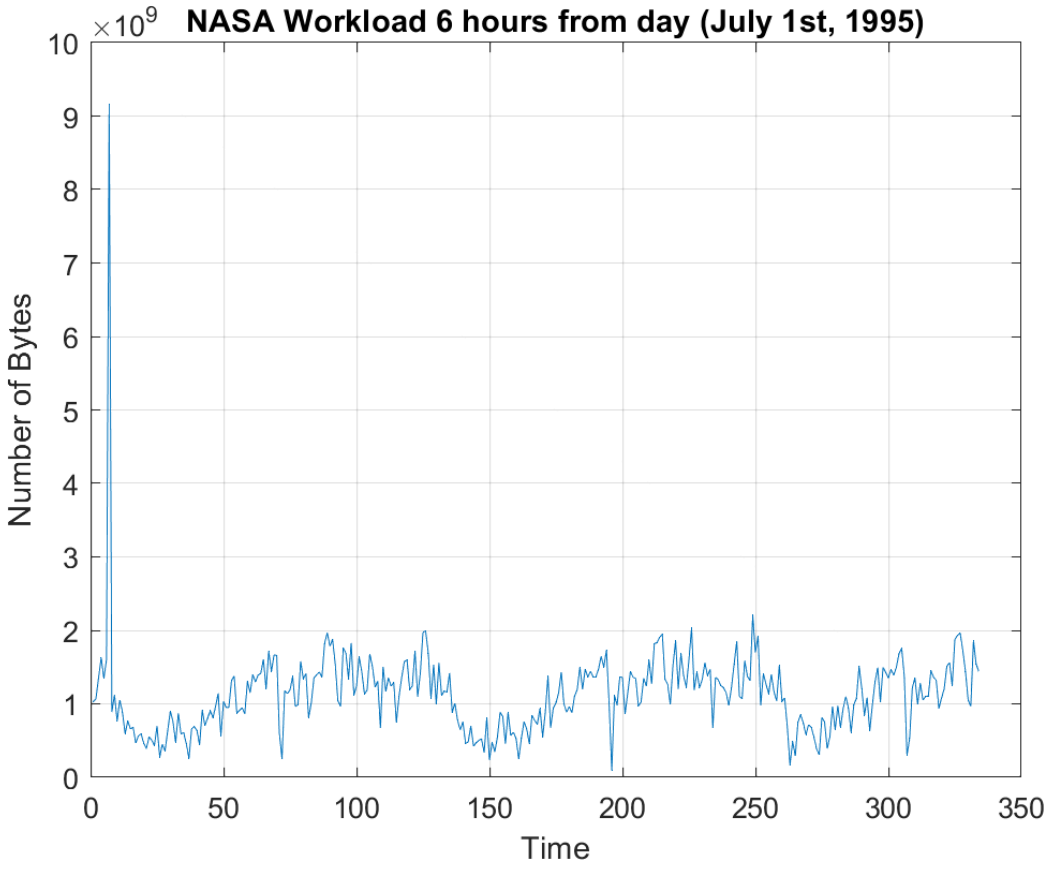


Figure 4.5: First subset of six hours of NASA-HTTP Workload.

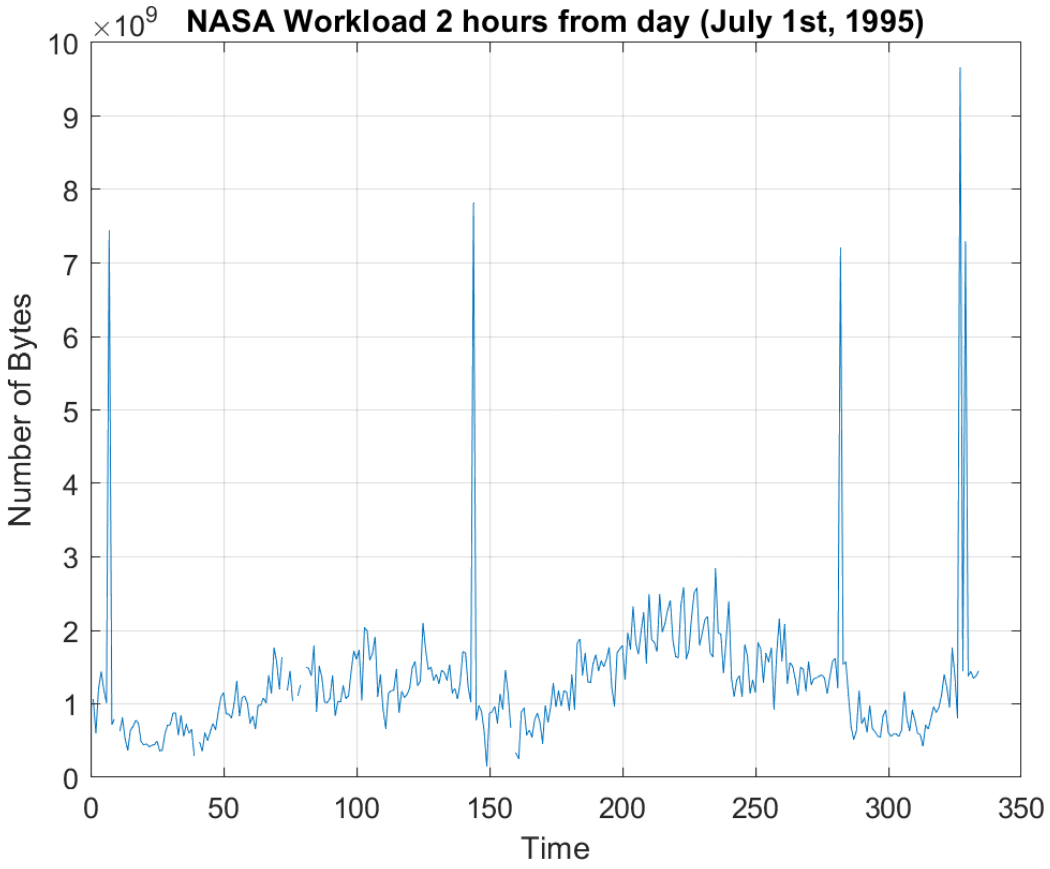


Figure 4.6: Second subset of repeating 2 hours of NASA-HTTP Workload.

line.

```
#! /bin/sh
while IFS=, read -r reply; do
#echo "$reply"
reply=$(echo $reply | sed 's/,,,//')
echo "$reply"
wget -q -O- http://php-apache/index.php?bytes=$reply
sleep 6
done < NASA_Dataset.csv
```

Listing 4.1: Shell script that reads the number of requests from the datasets.

Step 2: Computing Load This step defines a way to apply load in the kubernetes cluster to generate some CPU intensive computations. The first line `ini-set('max-execution-time', 600)` sets the maximum time a script can run before it is terminated. The reason for adding this line is to prevent tying up the containers in the server. Then, the code gets the number of bytes, call it B, from the previous step, and runs a loop for calculating square roots B times.

```
<?php
ini_set('max_execution_time', 600);
$x = 0.0001;
$arg = $_GET['bytes'];
$bytes = (int) $arg;
for ($i = 0; $i <= $bytes; $i++) {
  $x += sqrt($x);
}
echo $bytes;
?>
```

Listing 4.2: Shell script that creates load in a container.

Step 3: Importing Load Generator Image Dockerfile is the final step to insert a set of instructions to create a docker image inside kubernetes. The content of Dockerfile is shown below. The first line in the Dockerfile is `FROM 1234nag/loadgenerator:latest`; it pulls an image from an existing docker image. Then, the next four lines are used to add the datasets (NASA and FIFA, from their CSV files) and the shell scripts that read the number of requests from the CSV files. The following four lines are running a permission to read the added files, which are `NASA-Dataset.csv, FIFA-Dataset.csv, NASA-Dataset.sh, and FIFA-Dataset.sh`.

```
FROM 1234nag/loadgenerator:latest
```

```
ADD ./NASA_Dataset.csv /
ADD ./FIFA_Dataset.csv /
ADD ./NASA_Dataset.sh /
ADD ./FIFA_Dataset.sh /
RUN chmod +r /NASA_Dataset.csv
RUN chmod +r /NASA_Dataset.sh
RUN chmod +r /FIFA_Dataset.csv
RUN chmod +r /FIFA_Dataset.sh
```

Listing 4.3: Dockerfile

Now, all the instructions have been set, so it is time to build the docker image by using `-t switch` to set the tag of `1234nag/loadgenerator2:latest`.

```
docker build -t 1234nag/loadgenerator2:latest .
docker tag loadgenerator:latest 1234nag/loadgenerator:latest
sudo docker push 1234nag/loadgenerator:latest
```

Listing 4.4: Dockerfile

4.2 Experiment Setup

The experiment setup, whose details are shown in Table 4.1, consists of a kubernetes cluster configuration and a workload generator. In this thesis, we perform four experiments, each repeated five times by using a different autoscaling algorithm to evaluate their performance in terms of provisioning accuracy and timeshare. Each algorithm has a specific scaling period as described previously in Section 3.4. For example, Algorithm 3: moving window averages calculates the required number of pods every five minutes. In addition, we set the starting number of replica to one, and the autoscaling algorithm scale sup or down as response to the workload changes. We also set the target utilization threshold to 50% and the replica number between 1 and 10 as illustrated in Table 4.2.

All the experiments run on kubernetes `version V1.21.1` on Ubuntu `version 18.04.6 LTS` that contains four CPU cores. Moreover, the metrics server scrapes CPU utilization from kubelet every 50 seconds, by setting the `metric-resolution` to 50. For Algorithms 1, 2, and 3, we collect information every ten seconds and store them in a database.

Table 4.1: Details of the experimental setup.

Experiments	Algorithm	Scaling Time	Workload
E1	One-step history	2min	FIFA World Cup 98 (6 hours from day: May, 1st 1998)
	Rolling Average	1min	
	Moving Window Average	5min	
	HPA with Cooling Down	1min	
	HPA without Cooling Down	1min	
E2	One-step history	2min	NASA-HTTP Workload (6 hours from day: July, 1st 1995)
	Rolling Average	1min	
	Moving Window Average	5min	
	HPA with Cooling Down	1min	
	HPA without Cooling Down	1min	
E3	One-step history	2min	FIFA World Cup 98 (6 hours randomly)
	Rolling Average	1min	
	Moving Window Average	5min	
	HPA with Cooling Down	1min	
	HPA without Cooling Down	1min	
E4	One-step history	2min	NASA-HTTP Workload (almost 2 hours and repeat them for 6 hours)
	Rolling Average	1min	
	Moving Window Average	5min	
	HPA with Cooling Down	1min	
	HPA without Cooling Down	1min	

Table 4.2: Pod Configuration.

Scaling Metrics	Target Threshold	Min Replica	Max Replica	CPU Request	CPU Limits	Metric Resolution
Low level metric (CPU)	50%	1	10	200	500	50 second

4.2.1 Calculating Average CPU Utilization

At the time a decision to scale is made, the algorithm will calculate the total CPU utilization in millicores and the number of pods to use until the next decision. In this section, we explains the necessary steps for storing measurements and the challenges we faced and solved.

The first step (see listing 4.5) to storing the pod measurement in an array is importing Kubernetes packages that provide authentication to work properly with Kubernetes.

```
import (
  metav1 "k8s.io/apimachinery/pkg/apis/meta/v1"
  "k8s.io/client-go/kubernetes"
  "k8s.io/client-go/tools/clientcmd"
  "k8s.io/metrics/pkg/apis/metrics/v1beta1"
  metricsclientset "k8s.io/metrics/pkg/client/clientset/versioned"
)
```

Listing 4.5: Kubernetes Packages

Client Configuration Then, we need to configure the client with kubernetes cluster by using the script in listing **??**. It takes two arguments, *masterURL* and *kubeconfigPath*. The master URL argument is the control plane URL to reach the kube-api-server while kubeconfigPath is the location of the kubernetes configuration file that contains cluster, users, namespaces, and authentication information.

```
kubeconfig := flag.String("kubeconfig", "/home/ece792/.kube/config"
    , "location of kubeconfig")
config, err := clientcmd.BuildConfigFromFlags("", *kubeconfig)
    if err != nil {
      panic(err)
    }
```

API Client Instantiating Now we need to instantiate a client set using the configuration we have loaded. A client set contains API groups, and the client set exported in k8s.io/client-go/kubernetes contains all the officially supported groups for the chosen package version. This means that the latest version, as of writing, includes all the API groups available in Kubernetes v1.20.

For the metrics API however, we need to use an additional client set which is defined in k8s.io/metrics/pkg/client/clientset/versioned. The NewForConfigOrDie is a helper func-

tion used to declare panic if an error occurs. The NewForConfig can be used instead if you want to handle the error manually.

```
kube_cs, err := kubernetes.NewForConfig(config)
    if err != nil {
    panic(err)
  }
metric_cs, err := metricsclientset.NewForConfig(config)
  if err != nil {
    panic(err)
  }
```

Listing 4.6: API Client Instantiating

Querying The final step is querying using the client sets through *metrics API*. The code in listing 4.7 is used to list all pods in the default namespaces. The list has two parameters. `context.Background()` is a place to save a context that will be discussed later.

`metav1.ListOptionsLabelSelector:"run=php-apache"` returns only the objects that match a supplied label.

```
podMetrics, err := metric_cs.MetricsV1beta1().PodMetricses("default
   ").List(context.Background(), (metav1.ListOptions{LabelSelector:
    "run=php-apache"}))
    if err != nil {
    fmt.Println("Error:", err)
    return
    }
```

Listing 4.7: Querying Using Client Set

Collecting Pod Measurements Then, the query returns pod information (namely, pod name, window size, timestamp, creation timestamp, and CPU utilization) based on the length of live container (see listing **??**). The pod information is stored in two-dimensional array called `measurements[2000][10]` that uses a slice of structs named `measurmentPod`, which includes six fields.

```
type measurementPod struct {
  PodName           string
  CPU               float64
  WindowSize        float64
  TimeStamp         int64
```

```
  CreationTimestamp int64
  isAccountedFor    bool
}
var measurements [2000][10]measurementPod
var measurementIndex int32 = 0
var containerIndex int32 = 0

for _, podMetric := range podMetrics.Items {
    podContainers := podMetric.Containers
  for _, container := range podContainers {
    measurements[measurementIndex][containerIndex].PodName =
   podMetric.ObjectMeta.Name
        measurements[measurementIndex][containerIndex].WindowSize =
    podMetric.Window.Duration.Seconds()
        measurements[measurementIndex][containerIndex].TimeStamp =
   podMetric.Timestamp.Time.Unix()
        measurements[measurementIndex][containerIndex].
   CreationTimestamp = podMetric.CreationTimestamp.Time.Unix()
        measurements[measurementIndex][containerIndex].CPU =
   container.Usage.Cpu().ToDec().AsApproximateFloat64()
        measurements[measurementIndex][containerIndex].
   isAccountedFor = isAccountedFor
        containerIndex += 1
    }
    measurementIndex += 1
}
```

Duplicate Measurements In this thesis, the program pools pod information every ten seconds from the metric server. Since kubelet reports new measurements to the metrics server every 50 seconds, there will be duplicates as shown in Figure 4.7. In one minute, there is only two fresh unique measurements that we should consider while measuring the average utilization. However, the metrics server is sending the same measurement (duplicate) until the kubelet updates it. Therefore, we compare the current pod name and timestamp with the previous ones to eliminate the presence of duplicates. If current and previous pod name and timestamp have the same values, the program considers the current measurements as duplicate, and discards the previous measurements. However, if current and previous pod name and timestamp are not matching, the program considers the measurements as fresh measurements. Realizing the presence of duplicates and eliminating them was one of the biggest challenges we faced.

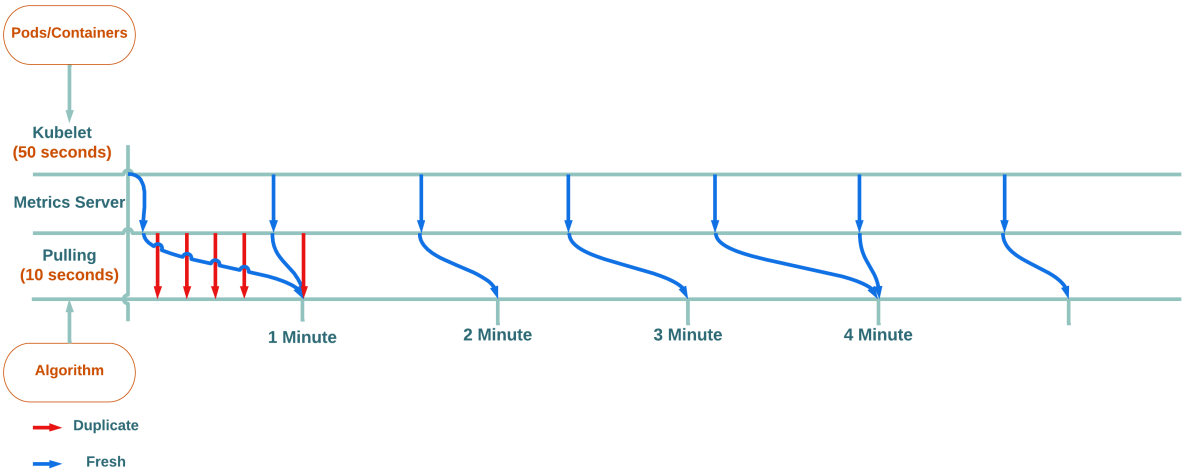


Figure 4.7: Duplicate Measurements.

```
func FindPreviousIndexDuplicateContainer(m [][10]measurementPod,
    index int32, name string, timestamp int64) int32 {
  for i := 0; i < int(maxReplica); i++ {
    mustCheckDuplicateMeasurement := name == m[index-1][i].PodName
    && int64(m[index-1][i].TimeStamp) == timestamp
    if mustCheckDuplicateMeasurement {
      return int32(i)
    }
  }
  return -1
}
```

Listing 4.8: Duplicate Measurements

Setting CPU Request and CPU Limit parameters There are two types of resource configuration, request and limits, that can be set on the deployment file. Request value is the minimum amount of resources that containers need while the limits is the max amount of resources that the container is allowed to consume. In this thesis, we set the CPU request to 200 millicore and limits to 500 millicore.

Average CPU Utilization After all the previous steps, we can calculate the average usage by using the formula in Equation 3.5, page 33. Based on the autoscaling algorithm, the scaling decision instant is different. When the scaling occurs, the program checks fresh

measurements to calculate the total CPU usage and total number of pods at the current period and then, applies the current metrics formula 3.5.

```
for j := pMeasurementindex; j < measurementIndex+1; j++ {
  for i := 0; i < int(maxReplica); i++ {
    mustTakeMeasurementIntoAccount := m[j][i].PodName != "dummy" &&
    !m[j][i].isAccountedFor
    if mustTakeMeasurementIntoAccount {
      totalCpu += m[j][i].CPU
      m[j][i].isAccountedFor = true
      numPods = numPods + 1
    }
  }
}
currMetrics = totalCpu * baseMilicore * 100.0 / (requestValue *
   float64(numPods))
```

Listing 4.9: Average CPU Utilization

4.2.2 Calculating the number of Desired Replicas

Depending on the scaling algorithm, the inputs needed to calculate the desired number of containers at the current period will be different. However, the main inputs are CPU utilization during the current period $[T_{k-1}, T_k)$, the desired threshold for CPU utilization, and the number of containers requested at the previous instant T_{k-1}.

CPU Utilization In the previous subsection 4.2.1, we discussed the steps to calculate the CPU utilization expressed as a percentage.

Target CPU Utilization In this thesis, we set the CPU utilization goal to 50%.

Current Replica php-apache deployment watches the status of the live replicas and assigns it to be the number of replica count. There are three types of status: progressing, complete, and fail to progress. Progressing status means that a new replica set is created or updated due to scaling up or down. While, complete status indicates that the required replicas is available and the replica set has changed. However, for some reason like insufficient quota, readiness probe failures, image pull errors, etc, the deployment may fail to update the replica set.

```
replicaCount = phpDeployment.Status.Replicas
```

Listing 4.10: Current Replica

Poll Replica Since the system might get stuck trying to deploy the new replica set, we had to create a function called poll replica. Poll replica checks the the difference between the number of containers requested at the previous decision instant T_{k-1} and the current number of containers to ensure that the number of replicas is updated in the current interval. In case the current replica is not updated, we should wait until the deployment updated before we scale up or down based on the requested number.

```
func PollReplicas(desiredReplicaCount int32, currReplicaCount int32
    , kube_cs *kubernetes.Clientset) {
  fmt.Println("THIS WILL SHIFT THE TIME LINE...")
  for {
    fmt.Println("Current Replica = ", currReplicaCount, "Desired
   Replica", desiredReplicaCount)
    time.Sleep(1 * time.Second)
    phpDeployment, err := kube_cs.AppsV1().Deployments("default").
   GetScale(context.TODO(), "php-apache", metav1.GetOptions{})
    if err != nil {
      fmt.Println("Error:", err)
      return
    }
    currReplicaCount = phpDeployment.Status.Replicas
    if desiredReplicaCount == currReplicaCount {
      fmt.Println("System in Sync")
      return
    }
  }
}
```

Listing 4.11: Poll Replica Function

Desired Replica Now, all the required inputs are available to apply the scaling algorithm that has been explained in Section 3.4. The entire code for implementing Algorithms 1, 2, and 3 is presented in Appendix B.

```
//Calculate the desired number of replicas using Algorithm 1: One-
   step history.
//Previous decision is the additional input.
```

```
replicaCount, preDecision = scalingAlgorithm(replicaCount,
    measurements[:][:], measurementIndex, pMeasurementindex,
    preDecision)

//Calculate the desired number of replicas using Algorithm 2 and
    Algorithm 3.
//No additional input.
replicaCount = scalingAlgorithm(replicaCount, measurements[:][:],
    measurementIndex, pMeasurementindex)
```

Listing 4.12: Calculating Desired Replica

Updating Current Replica Once the desired replica function provides the new number of replicas to meet the target CPU utilization, we need to update the deployment by specifying the number of pods should run through `.Spec.Replicas`. Then, the replica set will add or remove its pods.

```
phpDeployment.Spec.Replicas = replicaCount
kube_cs.AppsV1().Deployments("default").UpdateScale(context.TODO(),
    "php-apache", phpDeployment, metav1.UpdateOptions{})
```

Listing 4.13: Updating The Number of Replica During The Current Period

4.3 Results

4.3.1 The Top-Level Evaluation Questions

4.3.1.1 The Baseline Scenario

In Listing 4.14, we show a yaml file that creates a HPA for an existing Deployment called php-apache (see listing A.4) that autoscales the pods to maintain the target CPU utilization of 50%. The HPA scales up and down the replica counts between a minimum of 1 and maximum of 10. In this thesis, we run two different experiments using Kubernetes autoscaling algorithm, HPA. The first experiment was the default configuration where the cooldown period was set to 5 minutes. In the other experiment, we reconfigure the yaml file by adding three lines as shown in listing 4.15 in the end to set the cooling down to zero.

The following command is utilized to configure the php-apache deployment with CPU to create the HPA inside kubernetes

`$kubectl create -f <file-name>.yaml`

The following command is used to view the current status of the HPA

```
$kubectl get hpa
```

```
apiVersion: autoscaling/v2beta2
kind: HorizontalPodAutoscaler
metadata:
  name: cpu-autoscale
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: php-apache
  minReplicas: 1
  maxReplicas: 10
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageValue: 50m
```

Listing 4.14: Kubernetes Horizontal Pod Autoscaler (HPA) for CPU Utilization with cooldown

```
    behavior:
      scaleDown:
        stabilizationWindowSeconds: 0
```

Listing 4.15: Kubernetes Horizontal Pod Autoscaler (HPA) for CPU Utilization with cooldown zero

4.3.1.2 The Evaluation Questions

The central questions we attempt to answer in our evaluations are the following:

- How does an algorithm perform with regard to the four evaluation metrics?
- How is the number of replicas increasing or decreasing?
- What is the overhead associated with a given algorithm?

4.3.1.3 The "Winners"

Before we present detailed results, we summarize in Table 4.3 the algorithms which performed the best for a given metric. We provide a detailed explanation after we present the results, in Section 4.3.2.

Table 4.3: The best-performing algorithm among the three defined ones, per metric considered.

Experiments	(θ_U)	(θ_O)	(τ_U)	(τ_O)
E1	Moving Window	Moving Window	Moving Window	Moving Window
E2	Moving Window	Moving Window	Moving Window	Moving Window
E3	One-step history and Rolling Average	One-step history	Rolling Average	Moving Window
E4	Moving Window	Moving Window	Moving Window	Moving Window

4.3.1.4 Autoscaling Performance Metrics

To evaluate the autoscaling algorithms, we use the metrics described in Section 3.3. The aggregated metrics for all the four experiments are presented in Table 4.4. Lower values for the percentages of the under- and over-provisioning accuracy and timeshare indicate a better autoscaling algorithm. For easy of presentation, we present the best values for each metric as bold and orange shadow.

First, we compare the three proposed algorithm (One-step history, rolling average, and moving window average) together, and then we will compare them with kuberentes' HPA with cooldown equal to 5 minutes and zero.

Proposed Autoscaling Algorithms: By looking at *E1, E2,* and *E4* results, we can see that Algorithm 3, that utilizes a moving window average, has outperformed the other two proposed algorithms in terms of under-provisioning and over-provisioning accuracy and timeshare. Yet, in *E3,* the performance of moving window average did not yield the best results due to the randomness, random alternation, of the workload. As a matter of fact, the least percentages obtained in *E3* for the four evaluation criteria were not achieved by the same algorithm. To elaborate, rolling average's and One-step history resulted in similar percentage, 8%, in under-provisioning accuracy. However, the one-step history algorithm scores a lower per-

centage in over-provisioning accuracy while rolling average's results for under-provisioning timeshare is superior. Meanwhile, the moving window average over-provisioning timeshare accomplished the minimum percentage of 30%.

Proposed Autoscaling Algorithms Comparison with HPA: HPA algorithm with a cooling down period of five minutes produces low percentage of under-provisioning and over-provisioning accuracy metrics as 1% (*using FIFA May/1*), 1% (*using NASA July/1*), 2% (*using FIFA random*), and 1% (*using NASA Repeat*), so it provides less resources from the demand needs compared to the other algorithms.

In addition, the over-provisioning accuracy percentage for HPA algorithm with cooling down was also the minimal in all the experiments except *E1 (using FIFA May/1)* as it was 1%, but the HPA algorithm without cooling down was 0%.

Above all, determining which algorithm performs the worst is not a fair comparison because the workload plays a significant role in the performance of all algorithms. For example, the under-provisioning accuracy for Moving Window Average algorithm was 10% (*using FIFA random*). However, 3%, 5%, and 5% were the percentages of under-provisioning accuracy for the moving window average algorithm using *using FIFA May/1*), (*using NASA July/1*), and 1% (*using NASA Repeat*). This means that there is no guarantee that the percentage of provisioning accuracy will be the same while using different workloads.

Furthermore, HPA algorithm with cooling down period equal to five minutes performs better as it produce the smallest values in both the under- and over-provisioning timeshare. The only exception was that HPA algorithm without cooling down was 1% in the over-provisioning timeshare (*using FIFA May/1*).

Table 4.4: Autoscaling performance metrics for all the four experiments.

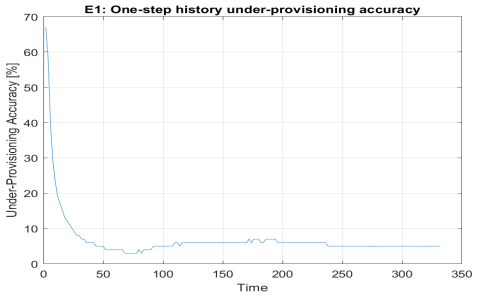
Experiments	Algorithm	(θ_U)	(θ_O)	(τ_U)	(τ_O)
E1	One-step history	5%	6%	20%	20%
	Rolling Average	12%	19%	34%	38%
	Moving Window Average	**3%**	**3%**	**11%**	**9%**
	HPA with Cooling Down	**1%**	1%	**3%**	3%
	HPA without Cooling Down	94%	**0%**	99%	**1%**
E2	One-step history	8%	10%	36%	34%
	Rolling Average	12%	20%	35%	40%
	Moving Window Average	**5%**	**5%**	**21%**	**17%**
	HPA with Cooling Down	**1%**	**1%**	**4%**	**3%**
	HPA without Cooling Down	9%	25%	23%	23%
E3	One-step history	**8%**	**10%**	33%	33%
	Rolling Average	**8%**	12%	**26**	36%
	Moving Window Average	10%	12%	35%	**30%**
	HPA with Cooling Down	**2%**	**3%**	**8%**	**8%**
	HPA without Cooling Down	4%	7%	9%	9%
E4	One-step history	8%	10%	36%	37%
	Rolling Average	11%	15%	35%	39%
	Moving Window Average	**5%**	**5%**	**20%**	**17%**
	HPA with Cooling Down	**1%**	**1%**	**3%**	**2%**
	HPA without Cooling Down	8%	21%	22%	78%

4.3.1.5 Overhead comparisons

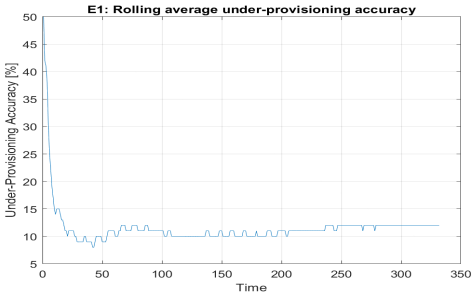
In this section, we discuss briefly the overheads related to the three algorithms and HPA. Overall, we do not see any big differences in any one of the three types of overhead.

CPU overhead: this is the time it takes to execute the autoscaling algorithm. HPA (with or without cooldown) runs its Equation 2.1, page 15, using a single measurement. Algorithm 1 has similar runtime, since it utilizes one measurement and only one previous decision. Algorithm 2 needs to perform $2M$ more additions (see lines 4 and 5), where M is the window size. And so does Algorithm 3. Compared to the overall time it takes to run the rest of the algorithms, the extra times are negligible.

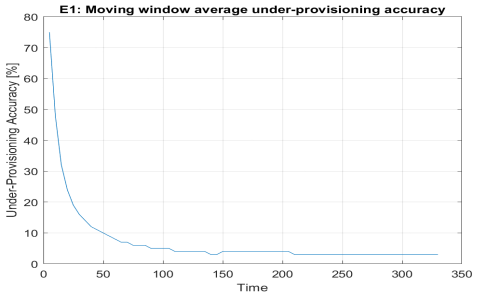
Memory overhead: Algorithm 1 needs to store one previous decision; this is the extra memory needed when compared to HPA. Algorithms 2 and 3 need some extra memory for



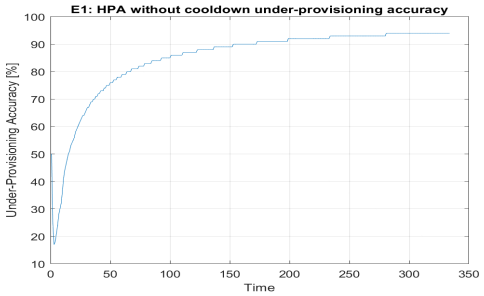
(a) One-step history Algorithm Under-Provisioning Accuracy



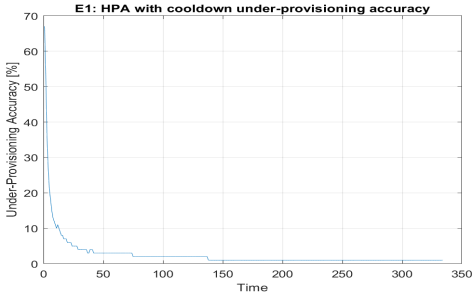
(b) Rolling Average Algorithm Under-Provisioning Accuracy



(c) Moving Window Average Algorithm Under-Provisioning Accuracy

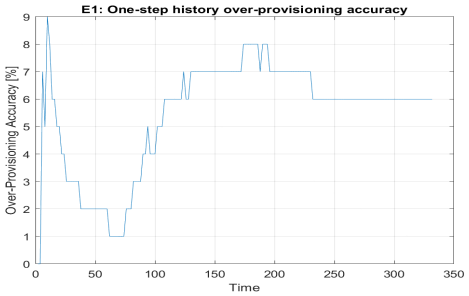


(d) HPA with cooling down Algorithm Under-Provisioning Accuracy

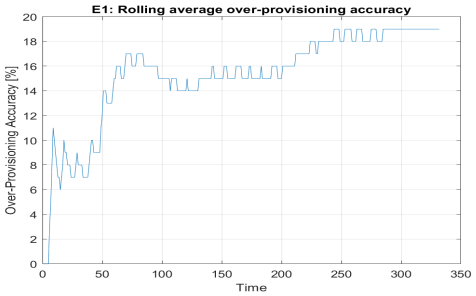


(e) HPA without cooling down Algorithm Under-Provisioning Accuracy

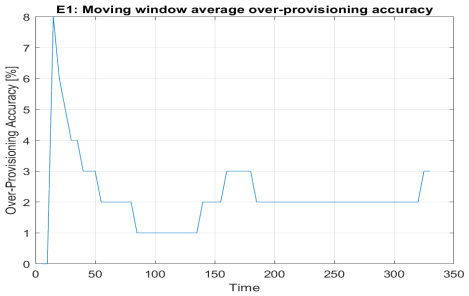
Figure 4.8: Comparison of The Under-Provisioning Accuracy Percentage for All Algorithms (E1: FIFA Dataset



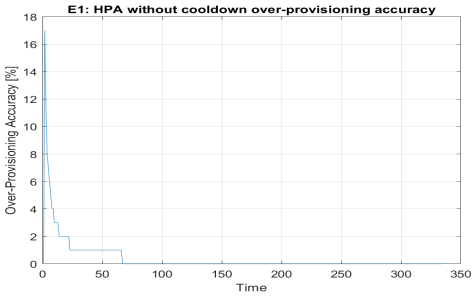
(a) One-step history Algorithm Over-Provisioning Accuracy



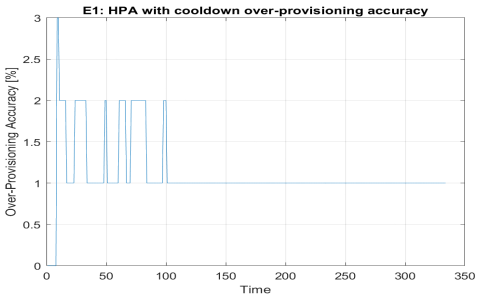
(b) Rolling Average Algorithm Over-Provisioning Accuracy



(c) Window Algorithm Over-Provisioning Accuracy

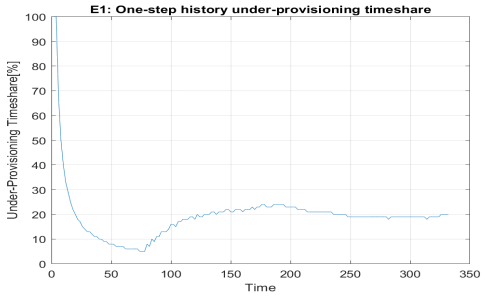


(d) HPA with cooling down Algorithm Over-Provisioning Accuracy

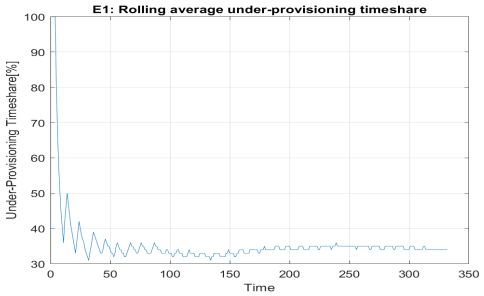


(e) HPA without cooling down Algorithm Over-Provisioning Accuracy

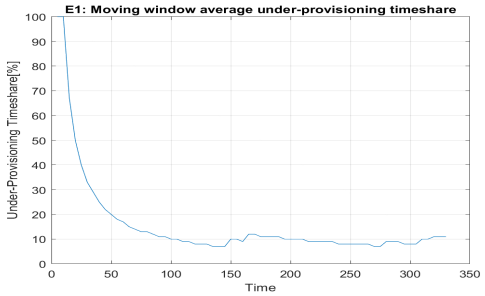
Figure 4.9: Comparison of The Over-Provisioning Accuracy Percentage for All Algorithms (E1: FIFA Dataset



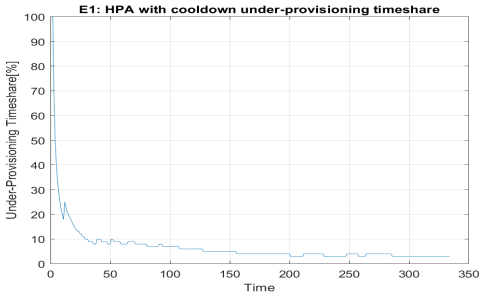
(a) One-step history Algorithm Under-Provisioning Timeshare



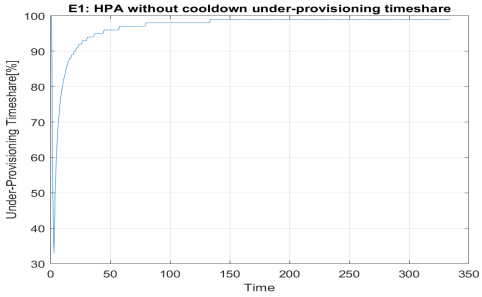
(b) Rolling Average Algorithm Under-Provisioning Timeshare



(c) Moving Window Average Algorithm Under-Provisioning Timeshare

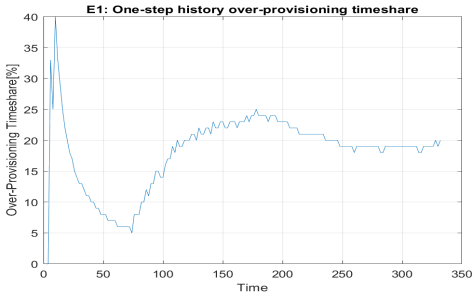


(d) HPA with cooling down Algorithm Under-Provisioning Timeshare

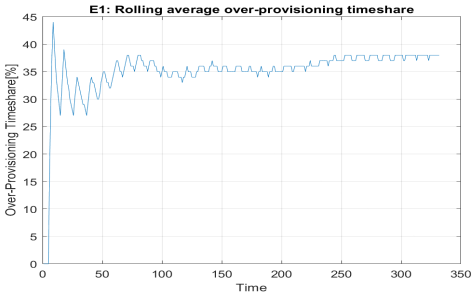


(e) HPA without cooling down Algorithm Under-Provisioning Timeshare

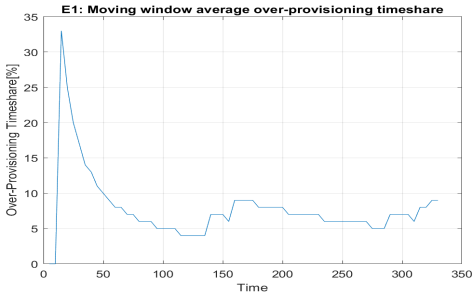
Figure 4.10: Comparison of The Under-Provisioning Timeshare Percentage for All Algorithms (E1: FIFA Dataset



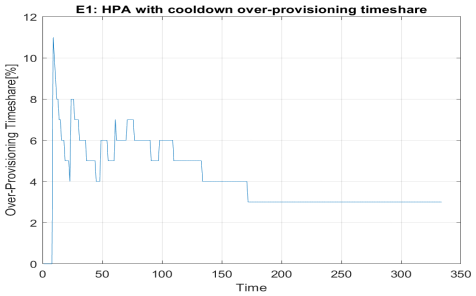
(a) One-step history Algorithm Over-Provisioning Timeshare



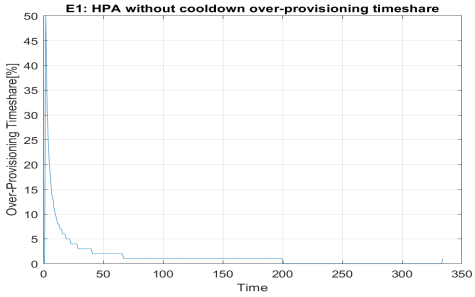
(b) Rolling Average Algorithm Over-Provisioning Timeshare



(c) Moving Window Average Algorithm Over-Provisioning Timeshare

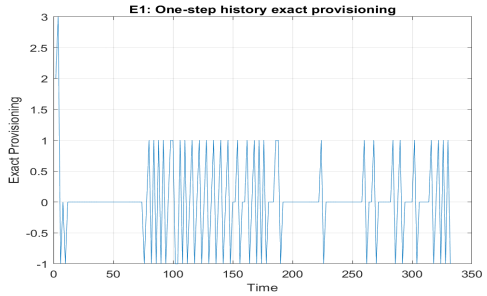


(d) HPA with cooling down Algorithm Over-Provisioning Timeshare

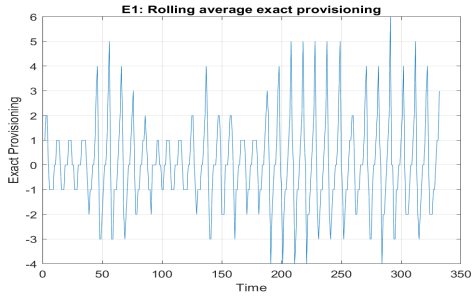


(e) HPA withoout cooling down Algorithm Over-Provisioning Timeshare

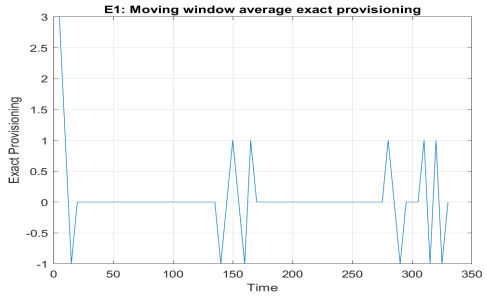
Figure 4.11: Comparison of The Over-Provisioning Timeshare Percentage for All Algorithms (E1: FIFA Dataset



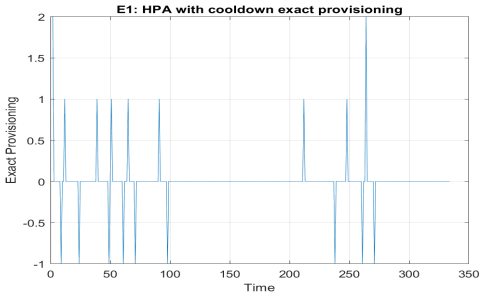
(a) One-step history Algorithm Exact Provisioning



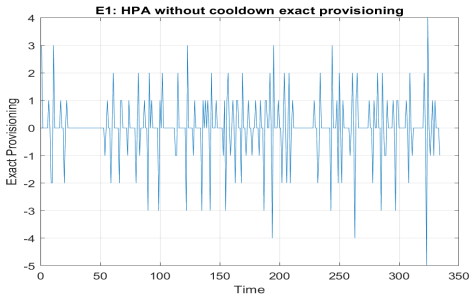
(b) Rolling Average Algorithm Exact Provisioning



(c) Moving Window Average Algorithm Exact Provisioning

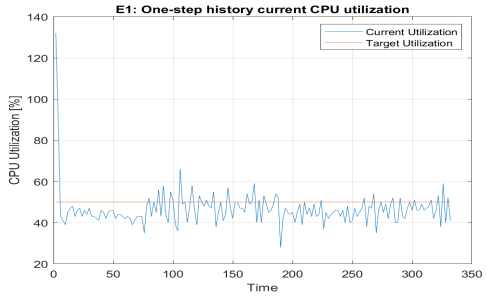


(d) HPA with cooling down Algorithm Exact Provisioning

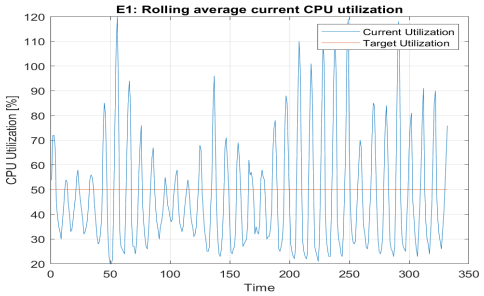


(e) HPA without cooling down Algorithm Exact Provisioning

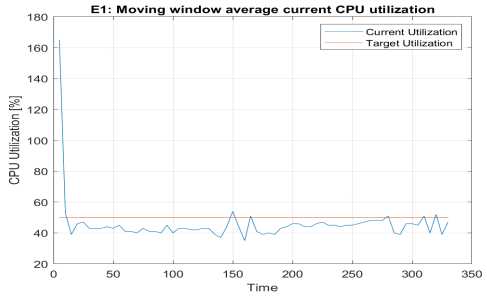
Figure 4.12: Comparison of The Exact Provisioning Accuracy for All Algorithms (E1: FIFA Dataset



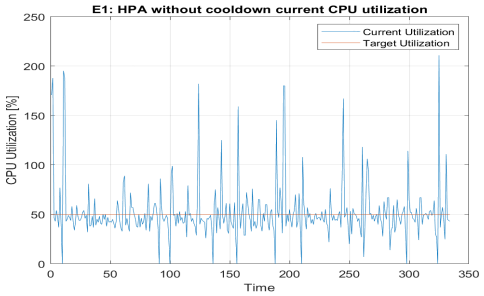
(a) One-step history Algorithm Current Metrics



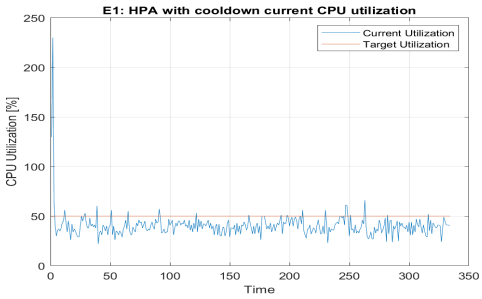
(b) Rolling Average Algorithm Current Metrics



(c) Moving Window Average Algorithm Current Metrics

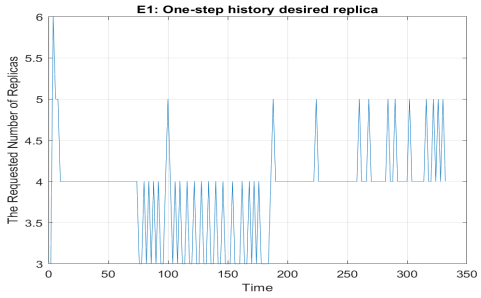


(d) HPA with cooling down Algorithm Current Metrics

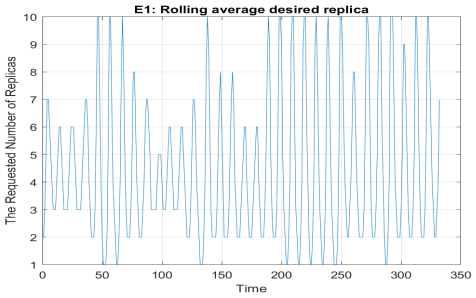


(e) HPA without cooling down Algorithm Current Metrics

Figure 4.13: Comparison of The CPU Utilization Percentage for All Algorithms (E1: FIFA Dataset

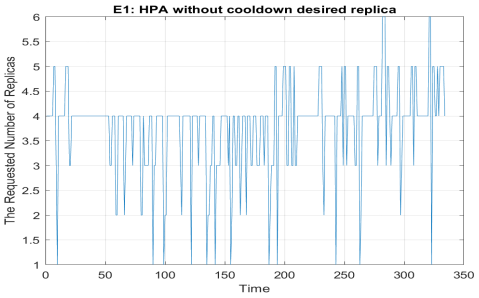


(a) One-step history Algorithm Desired Replica

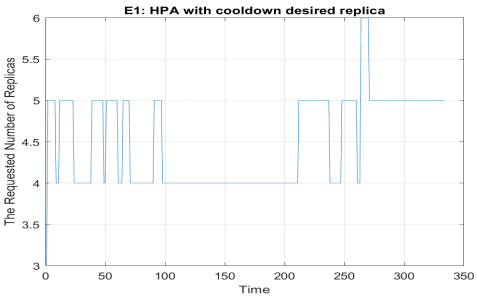


(b) Rolling Average Algorithm Desired Replica

(c) Moving Window Average Algorithm Desired Replica



(d) HPA with cooling down Algorithm Desired Replica



(e) HPA without cooling down Algorithm Desired Replica

Figure 4.14: Comparison of The Desired Replica e for All Algorithms (E1: FIFA Dataset

the M measurements. We also consider this overhead negligible.

Communication overhead: The overhead it takes to pull the measurements from the metrics server is the same for all algorithms. Algorithm 1 pulls the previous decision as well, but this overhead is negligible.

4.3.2 Overall Evaluation

CPU Utilization: Figures C.20 show two lines, the blue line represents the variation in CPU utilization and the orange line illustrates the target CPU threshold when we test our five algorithms using four different real dataset.

Since the initial pod count is 1 replica, the minimum number of pods specified in the configuration file, the system will notice a sudden jump in the CPU utilization percentage when the load starts running. Once the autoscaling algorithm respond to the incoming load by scaling the pod count, we can see that the system is able to handle the load and the CPU usage is less than the target CPU threshold (50%).

By comparing the five autoscaling algorithms together, we found that all of them not providing the same CPU utilization. For example, the first utilization detects were 132%, 54%, 165%, 130%, and 171% for One-step history, rolling average, window average, HPA with cooling down, and HPA without cooling down while running the same workload (FIFA Day May/1). This is due to two reasons as follow:

- **Detection and scaling Period:** This period is when the autoscaling algorithm detect that the CPU utilization is beyond the target threshold. As shown in table 4.1, each algorithm has its own scaling time that can be 1 minute, 2 minutes, or 5 minutes.
- **Delay Period:** Depending on when the autoscaling algorithm pools for metrics from the metrics server, there could be a delay period of 50-65seconds for HPA algorithm, or 50-60 seconds for others three algorithms.
    1. *Pooling Period:* One-step history, Rolling Average, and Moving Window Average pool for metrics every 10 seconds while HPA, whether with or without cooling down, pools for metrics every 15 seconds.
    2. *Metrics Server Period:* It is the time that metrics server polls for aggregate metrics every 50 seconds.

This means that there is a time between when the autoscaling detecting that the target CPU threshold was breached and when additional replica(s) was up and running.

The Number of Requested Pods: The replicas recommended by the five autoscaling algorithms with target CPU utilization set at 50% are illustrated in Figure C.21. In these figures, we evaluate the cost efficiency as the cloud's clients are charges based on usage (*pay-as-you-go*). As the load increases, the replica count increases between 1 to 10 replicas, to maintain the CPU utilization below 50%.

Table 4.5 shows the peak number of replicas for each autoscaling algorithm in the four different experiments. We found that *Moving Window Average* algorithm provides less number of replicas compared to other four algorithms. This means that *Moving Window Average* algorithm is not wasted resources and increase the cost efficiency while the efficiency of *rolling average* algorithm declines due to increase the number of replicas.

Table 4.5: The Number of Requested Replica for Each Autoscaling Algorithms using diffident Workload.

Experiments	One-step history Algorithm	Rolling Average Algorithm	Window Average Algorithm	HPA with Cooling Down	HPA without Cooling Down
E1	6 replicas	10 replicas	**5 replicas**	6 replicas	6 replicas
E2	10 replicas	10 replicas	**6 replicas**	10 replicas	9 replicas
E3	10 replicas	10 replicas	10 replicas	10 replicas	10 replicas
E4	10 replicas	10 replicas	**8 replicas**	9 replicas	10 replicas

Moving Window Average algorithm minimizes wrong provisioning accuracy and timeshare metrics. In contrast to other proposed algorithms, Moving Window Average allows us to proactively respond to future CPU usage as it calculates the required number of pods every five minutes. This offers more time to collect the utilization to eliminates unnecessary scaling decision as the scaling decision happen less frequently compared to other algorithms. One-step history and Moving Window Average algorithms manage to lower their percentage in all the four provisioning metrics below the percentage of rolling average algorithm.

Rolling average algorithm performs poorly. While comparing the three Proposed algorithms, we notice that most of the time *rolling average* has the worst values of (θ_U), (θ_O), (τ_U), and (τ_O) due to highly inaccurate scaling decision. This means that the scaling decision calculates the number of pods required wrongly. Therefore, the provisioning accuracy and timeshare of rolling average algorithm perform the worst.

Autoscaling algorithm with longer provisioning times have better performance. While implementing *One-step history algorithm*, we set the scaling decision for 1 minute, 2 minutes, 3 minutes, 4 minutes, and 5 minutes. We notice that provisioning for long time is better as shown in table 4.6. Detecting and scaling every 1 minute performs worst, and while increasing the scaling period, the better percentage of (θ_U), (θ_O), (τ_U), and (τ_O). However, if we set the scaling period too long, the autoscaling algorithm will not be responsive to workload changes. This is the reason why the value of (θ_U), (θ_O), (τ_U), and (τ_O) for 5 minutes were going back to almost the same percentage for 1 minute.

Table 4.6: The performance of One-step history algorithm with different scaling Time.

One-step history Algorithm with Different Scaling Time	(θ_U)	(θ_O)	(τ_U)	(τ_O)
1min	10%	15%	36%	36%
2min	5%	6%	20%	20%
3min	7%	9%	29%	28%
4min	7%	8%	28%	25%
5min	9%	11%	35%	33%

Moreover, *rolling average* algorithm is more easily affected by new incoming load. For example, at time = 54, the current CPU utilization, current replica, and desired replica were 78%, 1, and 2, respectively. Then in the next interval (time=55), the utilization exceeded 108%.

Autoscaling algorithm behave different while running different workload. As illustrate in table 4.4, the Moving Window Average algorithm performs the best in *E1*, *E2*, and *E4*. However, Moving Window Average algorithm performs worst in *E3* as the $(\theta_U) = 10\%$, $(\theta_O) = 12\%$, and $(\tau_U) = 35\%$.

Moving Window Average algorithm has the smoother scaling pattern and the smallest number of replicas. Moving Window Average algorithm successfully decreased the waste of unused or unnecessary resources by providing less replica counts. In addition, it was able to not frequently scale, which produces less fluctuation. However, the maximum number of replicas in the rolling average algorithm was 10 replicas. This means that using rolling average algorithm will cost more money and more fluctuation, as the scaling time every 1 minute.

Proposed Autoscaling Algorithms Comparison with HPA: A comparison of all five algorithms is illustrated in table 4.4. We found that HPA algorithm with cooling down = 5 minutes achieves better performance on all metrics except (θ_O) and (τ_O) in *E1*. This means that HPA algorithm with cooling down tend to not under- and over-provisioning most of the time as they occur for a fraction of time as compared to other algorithms. The reason for this result is the cooling time period. The cooling down in HPA is set to five minutes while the other four algorithms were set to zero cooling down. Thus, the scale of requested replicas keep thrashing more in the four autoscaling algorithms compared to HPA.

CHAPTER

5

CONCLUSION AND FUTURE WORK

5.1 Conclusion

In cloud computing environments, reducing costs is one of the main business objectives and a key component of the "pay-as-you-go" financial model. In this thesis, we have investigated *service autoscaling*, one of the technical tools that technical designers use to control costs. More specifically, we have defined three scaling algorithms that utilize, in different ways, "history" in their scaling up or down decision making. We have evaluated such algorithms with actual traffic traces. In our evaluation, we have used four industry-defined metrics.

In summary, we have found that, overall and for most metrics, the moving window average algorithm performed better than the other two algorithms. When compared to HPA with cooling feature, the algorithms "underperformed".

We faced two main challenges in this work. The first was the need to code the algorithms in the *go language*. We had no prior experience in this language; learning it on our own was rewarding but took considerable time. The second challenge was related to the process of obtaining the utilization metrics from the kubernetes metric server. There was an inherent mismatch between the (lower) frequency of containers reporting such metrics and the

(higher) frequency of our algorithms pulling such metrics. The imbalance created duplicate measurements that we had to ensure were used only once; this required some code tweaking that also took considerable time.

5.2 Future Work

The work in this dissertation can be extended in three main directions, namely business, technical and academic.

(Business) New SLAs. The SLA we analyzed in this work had considered CPU-related metrics. The work can be extended to other business SLAs; for example, average response times, percentile utilizations or percentile response times. It would be interesting to develop pay as you go models based on such SLAs.

(Technical) Different scale up/down policies. We have considered three different ways to utilize history of past decisions. The technical work can be extended by considering: (a) similar algorithms but feedback on different metrics, (b) different algorithms altogether, and, (c) reduce "noisy neighbor" problems. An example of the first approach would be utilizing response times. An example of the second would be algorithms based on AI/ML ideas. Finally, the third problem deals with how to allocate a limited number of containers among competing users (neighbors).

(Technical) Develop rules of thumb. A deeper study around environment, configuration, and traffic parameters is needed for establishing rules of thumb that could be used in suggesting "best practices".

(Academic) Theoretical proofs. All three algorithms utilize "limited history"; appropriate Markovian models can be formulated based on assumptions regarding the traffic and service models. It is interesting then to attempt theoretical proofs that such algorithms (as well as kubernetes' HPA) can indeed satisfy the desired (CPU, memory, etc.) objectives.

REFERENCES

[1] Production-grade container orchestration. URL: https://kubernetes.io/.

[2] Containerd - an industry-standard container runtime with an emphasis on simplicity, robustness and portability, 2021. URL: https://containerd.io/.

[3] cri-o, Dec 2021. URL: https://cri-o.io/.

[4] Deployments, Sep 2021. URL: https://kubernetes.io/docs/concepts/workloads/controllers/deployment/.

[5] Jobs, Nov 2021. URL: https://kubernetes.io/docs/concepts/workloads/controllers/jobs-run-to-completion/.

[6] Kubernetes components, Oct 2021. URL: https://kubernetes.io/docs/concepts/overview/components/.

[7] Replication controller, Oct 2021. URL: https://kubernetes.io/docs/concepts/workloads/controllers/replicationcontroller/.

[8] Services, Nov 2021. URL: https://kubernetes.io/docs/concepts/services-networking/service/.

[9] Cloud infrastructure management: Cloud management services, Jan 2022. URL: https://www.appdynamics.com/learn/cloud-infrastructure-management#~how-cloud-infrastructure-works.

[10] Horizontal pod autoscaling, Mar 2022. URL: https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/.

[11] Resource metrics pipeline, Feb 2022. URL: https://kubernetes.io/docs/tasks/debug-application-cluster/resource-metrics-pipeline/#metrics-server.

[12] Empowering app development for developers, n.d. URL: https://www.docker.com/.

[13] Martin Adane. Cloud computing adoption: Strategies for sub-saharan africa smes for enhancing competitiveness. *African Journal of Science, Technology, Innovation and Development*, 10(2):197–207, 2018.

[14] Yahya Al-Dhuraibi, Fawaz Paraiso, Nabil Djarallah, and Philippe Merle. Elasticity in cloud computing: State of the art and research challenges. *IEEE Transactions on Services Computing*, 11(2):430–447, 2018. doi:10.1109/TSC.2017.2711009.

[15] Michael Armbrust, Armando Fox, Rean Griffith, Anthony Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A view of cloud computing. *Commun. ACM*, 53:50–58, 04 2010. doi:10.1145/1721654.1721672.

[16] Eric A. Brewer. Kubernetes and the path to cloud native. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, SoCC '15, page 167, New York, NY, USA, 2015. Association for Computing Machinery. URL: https://doi.org/10.1145/2806777.2809955, doi:10.1145/2806777.2809955.

[17] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. Borg, omega, and kubernetes: Lessons learned from three container-management systems over a decade. *Queue*, 14(1):70âĂŞ93, jan 2016. URL: https://doi-org.prox.lib.ncsu.edu/10.1145/2898442.2898444, doi:10.1145/2898442.2898444.

[18] J. P. Buzen and U. O. Gagliardi. The evolution of virtual machine architecture. In *Proceedings of the June 4-8, 1973, National Computer Conference and Exposition*, AFIPS '73, page 291âĂŞ299, New York, NY, USA, 1973. Association for Computing Machinery. URL: https://doi.org/10.1145/1499586.1499667, doi:10.1145/1499586.1499667.

[19] Emiliano Casalicchio. A study on performance measures for auto-scaling cpu-intensive containerized applications. *Cluster Computing*, 22, 09 2019. doi:10.1007/s10586-018-02890-1.

[20] Emiliano Casalicchio and Vanessa Perciballi. Auto-scaling of containers: The impact of relative and absolute metrics. 09 2017. doi:10.1109/FAS-W.2017.149.

[21] Chia-Chen Chang, Shun-Ren Yang, En-Hau Yeh, Phone Lin, and Jeu-Yih Jeng. A kubernetes-based monitoring platform for dynamic cloud resource provisioning. In *GLOBECOM 2017 - 2017 IEEE Global Communications Conference*, pages 1–6, 2017. doi:10.1109/GLOCOM.2017.8254046.

[22] Alfredo Cuzzocrea, Ladjel Bellatreche, and Il-Yeol Song. Data warehousing and olap over big data: Current challenges and future research directions. In *Proceedings of the Sixteenth International Workshop on Data Warehousing and OLAP*, DOLAP '13, page 67âĂŞ70, New York, NY, USA, 2013. Association for Computing Machinery. URL: https://doi.org/10.1145/2513190.2517828, doi:10.1145/2513190.2517828.

[23] Datadog. Datadog, Oct 2021. URL: https://www.datadoghq.com/container-report/.

[24] Tharam Dillon, Chen Wu, and Elizabeth Chang. Cloud computing: Issues and challenges. In *2010 24th IEEE International Conference on Advanced Information Networking and Applications*, pages 27–33, 2010. doi:10.1109/AINA.2010.187.

[25] Flexera. Rightscale 2019 state of the cloud report, Jan 2019. URL: https://resources.flexera.com/web/media/documents/rightscale-2019-state-of-the-cloud-report-from-flexera.pdf.

[26] Chunye Gong, Jie Liu, Qiang Zhang, Haitao Chen, and Zhenghu Gong. The characteristics of cloud computing. *2010 39th International Conference on Parallel Processing Workshops*, pages 275–279, 2010.

[27] Nikolas Herbst, Rouven Krebs, Giorgos Oikonomou, George Kousiouris, Athanasia Evangelinou, Alexandru Iosup, and Samuel Kounev. Ready for rain? a view from spec research on the future of cloud metrics. Technical Report SPEC-RG-2016-01, SPEC Research Group — Cloud Working Group, Standard Performance Evaluation Corporation (SPEC), 2016. URL: https://research.spec.org/fileadmin/user_upload/documents/rg_cloud/endorsed_publications/SPEC-RG-2016-01_CloudMetrics.pdf.

[28] Nikolas Roman Herbst, Samuel Kounev, and Ralf Reussner. Elasticity in cloud computing: What it is, and what it is not. In *10th International Conference on Autonomic Computing (ICAC 13)*, pages 23–27, San Jose, CA, June 2013. USENIX Association. URL: https://www.usenix.org/conference/icac13/technical-sessions/presentation/herbst.

[29] Djilali Idoughi, Karima Ait Abdelouhab, and Christophe Kolski. Towards a microservices development approach for the crisis management field in developing countries. *2017 4th International Conference on Information and Communication Technologies for Disaster Management (ICT-DM)*, pages 1–6, 2017.

[30] A. Anasuya Innocent. Cloud infrastructure service management - a review. *IJCSI International Journal of Computer Science Issues 1694-0814*, 9:287–292, 05 2012.

[31] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. Kvm: the linux virtual machine monitor. In *In Proceedings of the 2007 Ottawa Linux Symposium (OLSâĂŹ-07*, 2007.

[32] Kubernetes. Autoscaler/cluster-autoscaler at master Âů kubernetes/autoscaler. URL: https://github.com/kubernetes/autoscaler/tree/master/cluster-autoscaler.

[33] Sean Marston, Zhi Li, Subhajyoti Bandyopadhyay, Julie Zhang, and Anand Ghalsasi. Cloud computing âĂŤ the business perspective. *Decision Support Systems*, 51:176–189, 04 2011. doi:10.2139/ssrn.1413545.

[34] Peter Mell and Timothy Grance. The nist definition of cloud computing, Sep 2011. URL: https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-145.pdf.

[35] Shripad Nadgowda, Sahil Suneja, and Ali Kanso. Comparing scaling methods for linux containers. pages 266–272. IEEE, 2017.

[36] Athanasios Naskos, Anastasios Gounaris, and Spyros Sioutas. *Cloud Elasticity: A Survey*, pages 151–167. Algorithmic Aspects of Cloud Computing. Springer International Publishing, Cham, 2016.

[37] Thanh-Tung Nguyen, Yu-Jin Yeom, Taehong Kim, Dae-Heon Park, and Sehan Kim. Horizontal pod autoscaling in kubernetes for elastic container orchestration. *Sensors*, 20(16), 2020. URL: https://www.mdpi.com/1424-8220/20/16/4621, doi:10.3390/s20164621.

[38] Trung Nguyen Tri, Eui-nam Huh, Jae Park, Md Hossain, Md. Delowar Hossain, Seung-Jin Lee, Jin Jang, Seo Jo, Luan Huynh, and Khanh Tran. Performance analysis of data parallelism technique in machine learning for human activity recognition using lstm. pages 387–391, 12 2019. doi:10.1109/CloudCom.2019.00066.

[39] Claus Pahl. Containerisation and the paas cloud. *IEEE Cloud Computing*, 2:24–31, 06 2015. doi:10.1109/MCC.2015.51.

[40] Claus Pahl, Antonio Brogi, Jacopo Soldani, and Pooyan Jamshidi. Cloud container technologies: A state-of-the-art review. *IEEE Transactions on Cloud Computing*, 7(3):677–692, 2019. doi:10.1109/TCC.2017.2702586.

[41] Dana Petcu. Multi-cloud: expectations and current approaches. In *Proceedings of the 2013 international workshop on Multi-cloud applications and federated clouds*, pages 1–6, 2013.

[42] Chenhao Qu, Rodrigo Calheiros, and Rajkumar Buyya. Auto-scaling web applications in clouds: A taxonomy and survey. *ACM Computing Surveys*, 51, 09 2016. doi:10.1145/3148149.

[43] B. Rochwerger, D. Breitgand, E. Levy, A. Galis, K. Nagin, I. M. Llorente, R. Montero, Y. Wolfsthal, E. Elmroth, J. Caceres, M. Ben-Yehuda, W. Emmerich, and F. Galan. The reservoir model and architecture for open federated cloud computing. *IBM Journal of Research and Development*, 53(4):4:1–4:11, 2009. doi:10.1147/JRD.2009.5429058.

[44] Yaman Roumani and Joseph K Nwankpa. An empirical study on predicting cloud incidents. *International journal of information management*, 47:131–139, 2019.

[45] Andrew Silver. Software simplified. *Nature (London)*, 546(7656):173–174, 2017.

[46] J.E. Smith and Ravi Nair. The architecture of virtual machines. *Computer*, 38(5):32–38, 2005. doi:10.1109/MC.2005.173.

[47] Barrie Sosinsky. *Cloud Computing Bible*. Wiley Publishing, 1st edition, 2011.

[48] Thomas Wang, Simone Ferlin, and Marco Chiesa. Predicting cpu usage for proactive autoscaling. In *Proceedings of the 1st Workshop on Machine Learning and Systems*, EuroMLSys '21, page 31âĂŞ38, New York, NY, USA, 2021. Association for Computing Machinery. URL: https://doi.org/10.1145/3437984.3458831, doi:10.1145/3437984.3458831.

[49] Zhuoqun Yang, Zhi Li, Zhi Jin, and Yunchuan Chen. A systematic literature review of requirements modeling and analysis for self-adaptive systems. In Camille Salinesi and Inge van de Weerd, editors, *Requirements Engineering: Foundation for Software Quality*, pages 55–71, Cham, 2014. Springer International Publishing.

[50] Guangba Yu, Pengfei Chen, and Zibin Zheng. Microscaler: Cost-effective scaling for microservice applications in the cloud with an online learning approach. *IEEE Transactions on Cloud Computing*, pages 1–1, 2020. doi:10.1109/TCC.2020.2985352.

[sort, numbers]natbib

APPENDICES

APPENDIX

A

PHP-APACHE SERVER AND LOAD GENERATOR SETUP

A.1 Docker Image

```
FROM 1234 nag / loadgenerator : latest
ADD ./ NASA_DAY.csv /
ADD ./ FIFA_DAY.csv /
ADD ./ NASA_DAY.sh /
ADD ./ FIFA_DAY.sh /
ADD ./ NASA_Repeat.csv /
ADD ./ RandomFIFA.csv /
ADD ./ NASA_Repeat.sh /
ADD ./ RandomFIFA.sh /
RUN chmod + r / NASA_DAY.csv
RUN chmod + r / NASA_DAY.sh
RUN chmod + r / FIFA_DAY.csv
RUN chmod + r / FIFA_DAY.sh
RUN chmod + r / NASA_Repeat.csv
RUN chmod + r / NASA_Repeat.sh
```

```
RUN chmod +r /RandomFIFA.csv
RUN chmod +r /RandomFIFA.sh
```

Listing A.1: Dockerfile

A.2 Load Generator

```
#! /bin/sh
while IFS=, read -r reply; do
#echo "$reply"
reply=$(echo $reply | sed 's/,,,//')
echo "$reply"
wget -q -O- http://php-apache/index.php?bytes=$reply
sleep 6
done < \textbf{FIFA_DAY}.csv
```

Listing A.2: FIFA Dataset Script

```
<?php
    //Expand the maximum execution time to 600, so it can take more
    load.
    //Defualt was 300.
    ini_set('max_execution_time', 600);
    $x = 0.0001;
    //define a variable that can take integer value (number of
   bytes).
    $arg = $_GET['bytes'];
    $bytes = (int) $arg;
    //a loop to calculate the squre root of x. The loop's lenght
   will equal to number of bytes.
    for ($i = 0; $i <= $bytes; $i++) {
        $x += sqrt($x);
    }
    //print bytes number everytime you access this code.
    echo $bytes;
?>
```

Listing A.3: One bytes index.php

Run the load

```
# Run this in a separate terminal
# so that the load generation continues and you can carry on with
   the rest of the steps
```

```
kubectl run -i --tty load-generator --rm --image=busybox --restart=
   Never -- /bin/sh -c "while sleep 0.01; do wget -q -O- http://php
   -apache; done"
```

A.3 Deployment

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: php-apache
spec:
  selector:
    matchLabels:
      run: php-apache
  replicas: 4
  template:
    metadata:
      labels:
        run: php-apache
    spec:
      containers:
      - name: php-apache
        image: saramashat/php-apache:latest
        imagePullPolicy: Always
        ports:
        - containerPort: 80
        resources:
          limits:
            cpu: 500m
          requests:
            cpu: 200m
---
apiVersion: v1
kind: Service
metadata:
  name: php-apache
  labels:
    run: php-apache
spec:
  ports:
  - port: 80
```

```
  selector:
    run: php-apache
```

Listing A.4: php-apache.yaml

A.4 Metrics Server

```
apiVersion: v1
kind: ServiceAccount
metadata:
  labels:
    k8s-app: metrics-server
  name: metrics-server
  namespace: kube-system
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  labels:
    k8s-app: metrics-server
    rbac.authorization.k8s.io/aggregate-to-admin: "true"
    rbac.authorization.k8s.io/aggregate-to-edit: "true"
    rbac.authorization.k8s.io/aggregate-to-view: "true"
  name: system:aggregated-metrics-reader
rules:
- apiGroups:
  - metrics.k8s.io
  resources:
  - pods
  - nodes
  verbs:
  - get
  - list
  - watch
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  labels:
    k8s-app: metrics-server
  name: system:metrics-server
rules:
```

```
- apiGroups:
  - ""
  resources:
  - nodes/metrics
  verbs:
  - get
- apiGroups:
  - ""
  resources:
  - pods
  - nodes
  verbs:
  - get
  - list
  - watch
---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  labels:
    k8s-app: metrics-server
  name: metrics-server-auth-reader
  namespace: kube-system
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: extension-apiserver-authentication-reader
subjects:
- kind: ServiceAccount
  name: metrics-server
  namespace: kube-system
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  labels:
    k8s-app: metrics-server
  name: metrics-server:system:auth-delegator
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: system:auth-delegator
```

```
subjects:
- kind: ServiceAccount
  name: metrics-server
  namespace: kube-system
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  labels:
    k8s-app: metrics-server
  name: system:metrics-server
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: system:metrics-server
subjects:
- kind: ServiceAccount
  name: metrics-server
  namespace: kube-system
---
apiVersion: v1
kind: Service
metadata:
  labels:
    k8s-app: metrics-server
  name: metrics-server
  namespace: kube-system
spec:
  ports:
  - name: https
    port: 443
    protocol: TCP
    targetPort: https
  selector:
    k8s-app: metrics-server
---
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    k8s-app: metrics-server
  name: metrics-server
```

```
  namespace: kube-system
spec:
  selector:
    matchLabels:
      k8s-app: metrics-server
  strategy:
    rollingUpdate:
      maxUnavailable: 0
  template:
    metadata:
      labels:
        k8s-app: metrics-server
    spec:
      containers:
      - args:
        - --cert-dir=/tmp
        - --secure-port=4443
        - --kubelet-preferred-address-types=InternalIP,ExternalIP,
   Hostname
        - --kubelet-use-node-status-port
        - --metric-resolution=50s
        image: k8s.gcr.io/metrics-server/metrics-server:v0.6.1
        command:
          - /metrics-server
          - --kubelet-insecure-tls
        imagePullPolicy: Always
        livenessProbe:
          failureThreshold: 3
          httpGet:
            path: /livez
            port: https
            scheme: HTTPS
          periodSeconds: 10
        name: metrics-server
        ports:
        - containerPort: 4443
          name: https
          protocol: TCP
        readinessProbe:
          failureThreshold: 3
          httpGet:
            path: /readyz
```

```
            port: https
            scheme: HTTPS
          initialDelaySeconds: 20
          periodSeconds: 10
        resources:
          requests:
            cpu: 100m
            memory: 200Mi
        securityContext:
          allowPrivilegeEscalation: false
          readOnlyRootFilesystem: true
          runAsNonRoot: true
          runAsUser: 1000
        volumeMounts:
        - mountPath: /tmp
          name: tmp-dir
      nodeSelector:
        kubernetes.io/os: linux
      priorityClassName: system-cluster-critical
      serviceAccountName: metrics-server
      volumes:
      - emptyDir: {}
        name: tmp-dir
---
apiVersion: apiregistration.k8s.io/v1
kind: APIService
metadata:
  labels:
    k8s-app: metrics-server
  name: v1beta1.metrics.k8s.io
spec:
  group: metrics.k8s.io
  groupPriorityMinimum: 100
  insecureSkipTLSVerify: true
  service:
    name: metrics-server
    namespace: kube-system
  version: v1beta1
  versionPriority: 100
```

Listing A.5: metricsServer.yaml

APPENDIX

B

PODMETRICS GOLANG IMPLEMENTATION

B.1 Current Metrics Function

B.1.1 Algorithm 1: One-Step History

```
//currentMetrics function calculates current metrics by taking:
//Inputs: CPU Utlization in Nanocore, Number of Running Pods, and
   Request Value.
//Output: Current Metrics in precentage.
func currentMetrics(m [][10]measurementPod, measurementIndex int32,
    pMeasurementindex int32) float64 {
  var totalCpu float64 = 0
  var currMetrics float64 = 0
  var numPods int32 = 0

  //Verify if the last measurement is accounted for in scaling
   algorithm.
  // If not, take that into account as well.
```

```
  //The loop start with the last measuremt is accounted and ends at current measurements,
  //so it will take six measurements for 1 min.
  //prvious measurement index initaly is 0
  //measurement index is the current measurement.
  for j := pMeasurementindex; j < measurementIndex+1; j++ {
    //Another loop to check inside each measurements for the number of running containers
    for i := 0; i < int(maxReplica); i++ {
      //before we calculate the total CPU and number of containers.
      //Check the Pod Name if is not dummy (this means we have a CPU value).
      //Also check if the measurement is accounted (false).
      //True --> we take it into account.
      //False --> we didn't take, so we have to consider it.
      mustTakeMeasurementIntoAccount := m[j][i].PodName != "dummy" && !m[j][i].isAccountedFor
      if mustTakeMeasurementIntoAccount {
        //Calculating the total CPU in cores and containers number.
        //Then, we set the measurement to True to not retake into account.
        totalCpu += m[j][i].CPU
        m[j][i].isAccountedFor = true
        numPods = numPods + 1
      }
    }
  }

  //Current Metrics in Precentage = (Average CPU Utlization in Milicore) / Request Value * 100 = Total CPU (core) / (Number of pods * request Value)
  //fmt.Println("Total CPU: ", totalCpu, "\nTotal number of pods: ", numPods)
  currMetrics = totalCpu * baseMilicore * 100.0 / (requestValue * float64(numPods))
  //Printing and returning the current metrics in precentage.
  fmt.Println("Current Metrics = ", currMetrics)
  return currMetrics
}
```

Listing B.1: Current Metrics

B.1.2 Algorithm 2: Rolling Average

```
//currentMetrics function calculates current metrics by taking:
//Inputs: CPU Utlization in Nanocore, Number of Running Pods, and Request Value.
//Output: Current Metrics in precentage.
func currentMetrics(m [][10]measurementPod, measurementIndex int32, pMeasurementindex int32) float64 {
  var totalCpu float64 = 0
  var currMetrics float64 = 0
  var numPods int32 = 0
  var overallCPU float64 = 0 //total CPU value considered for the current metrics calculation
  var totalPods int32 = 0    //total number of pods considered for the current metrics calculation

  //Verify if the last measurement is accounted for in scaling algorithm.
  // If not, take that into account as well.
  //The loop start with the last measuremt is accounted and ends at current measurements
  //so it will take six measurements for 1 min.
  //previous measurement index initially is 0
  //measurement index is the current measurement
  for j := pMeasurementindex; j < measurementIndex+1; j++ {
    //Another loop to check inside each measurements for the number of running containers
    for i := 0; i < int(maxReplica); i++ {
      //before we calculate the total CPU and number of containers.
      //Check the Pod Name if is not dummy (this means we have a CPU value).
      //Also check if the measurement is accounted (false).
      //True --> we take it into account.
      //False --> we didn't take, so we have to consider it.
      mustTakeMeasurementIntoAccount := m[j][i].PodName != "dummy" && !m[j][i].isAccountedFor
      if mustTakeMeasurementIntoAccount {
        //Calculating the total CPU in cores and containers number.
        //Then, we set the measurement to True to not retake into account.
        totalCpu += m[j][i].CPU
        m[j][i].isAccountedFor = true
```

```
31          numPods = numPods + 1
32        }
33      }
34    }
35
36    //The totalCPU value and the numPods from the above calculation
       is stored in the totalCPUIndex array
37    //We will get the totalCPU and numPods for every 1 min
38    totalCPUIndex[index].CPU = totalCpu
39    totalCPUIndex[index].nPods = numPods
40    //fmt.Println("totalCPUIndex: ", totalCPUIndex)
41
42    //k-> indexValue to 0 //k indicates the current decision
43    // if k= 2, it is decremented to 1 and 0 to consider all three
       decision values
44    //counter-> 5 to 1 // we are considering latest 5 minutes scaling
        decision
45    for k, counter := index, 5; k >= 0 && counter > 0; k, counter = k
       -1, counter-1 {
46      //fmt.Println("counter: ", counter)
47      //fmt.Println("gettingAddedCPu", totalCPUIndex[k].CPU)
48      overallCPU += totalCPUIndex[k].CPU
49      //fmt.Println("overallCPU: ", overallCPU)
50      //fmt.Println("gettingAddedPods", totalCPUIndex[k].nPods)
51      totalPods += totalCPUIndex[k].nPods
52      //fmt.Println("totalPods: ", totalPods)
53    }
54    //fmt.Println("overallCPU: ", overallCPU)
55    //fmt.Println("totalPods: ", totalPods)
56    index += 1 //incrementing index to store next scaling decision
       values
57
58    //overallCPU is the summation of CPU values of latest 5 minute
       scaling decisions
59    //Current Metrics in Precentage = (Average CPU Utlization in
       Milicore) / Request Value * 100 = Total CPU (core) / (request
       Value * Number of pods )
60    currMetrics = overallCPU * baseMilicore * 100.0 / (requestValue *
        float64(totalPods))
61    //Printing and returning the current metrics in precentage
62    return currMetrics
```

```
}
```

Listing B.2: Current Metrics Function that used for Rolling Average

B.1.3 Algorithm 3: Moving Window Average

```
//currentMetrics function calculates current metrics by taking:
//Inputs: CPU Utlization in Nanocore, Number of Running Pods,
   Average Utilization Array
//Output: Average Utlization Array for Every 1 min
func currentMetrics(m [][10]measurementPod, measurementIndex int32,
    pMeasurementindex int32, totalUtilizationArray []float64,
   totalNumPodsArray []int32, index int32) {
  var totalCpu float64 = 0
  //var currMetrics float64 = 0
  var numPods int32 = 0

  //Verify if the last measurement is accounted for in scaling
   algorithm.
  // If not, take that into account as well.
  //The loop start with the last measuremt is accounted and ends at
    current measurements,
  //so it will take six measurements for 1 min.
  //prvious measurement index initaly is 0
  //measurement index is the current measurement.
  for j := pMeasurementindex; j < measurementIndex+1; j++ {
    //Another loop to check inside each measurements for the number
    of running containers
    for i := 0; i < int(maxReplica); i++ {
      //before we calculate the total CPU and number of containers.
      //Check the Pod Name if is not dummy (this means we have a
   CPU value).
      //Also check if the measurement is accounted (false).
      //True --> we take it into account.
      //False --> we didn't take, so we have to consider it.
      mustTakeMeasurementIntoAccount := m[j][i].PodName != "dummy"
   && !m[j][i].isAccountedFor
      if mustTakeMeasurementIntoAccount {
        //Calculating the total CPU in cores and containers number.
        //Then, we set the measurement to True to not retake into
   account.
        totalCpu += m[j][i].CPU
        m[j][i].isAccountedFor = true
```

```
29          numPods = numPods + 1
30        }
31      }
32    }
33    //fmt.Println("Total CPU  [CurrentMetrics Function] ", totalCpu)
34    //fmt.Println(" Total number of pods [CurrentMetrucs Function] ",
        numPods)
35    //Calculate the average utlization every 1 min
36    totalUtilizationArray[index] = totalCpu
37    //Store the average utlization in an array to be used to
       calculate the current metrics in precentage.
38    totalNumPodsArray[index] = numPods
39    //fmt.Println("totalUtilizationArray [CurrentMetrics Function] ",
        totalUtilizationArray[index], "totalCpu", totalCpu)
40    //fmt.Println("totalNumPodsArray [CurrentMetrucs Function] ",
       totalNumPodsArray[index], "numPods", numPods)
41  }
```

Listing B.3: Current Metrics Function that used for Moving Window Average

B.2 Scaling Function

B.2.1 Algorithm 1: One-Step History

```
1  //scalingAlgorithm function calculates the Desired Replicas based
       on Algorithm 1.
2  //Input : Current Metrics, Desired Metrics, Current Relicas, and
       Previous Decision.
3  //Output: Desired Replicas and updated Previous Decision.
4  func scalingAlgorithm(replicaCount int32, m [][10]measurementPod,
       measurementIndex int32, pMeasurementindex int32, preDecision
       string) (int32, string) {
5    var desiredReplicas int32 = 0
6
7    //recal current metrics function to provide current metrics for
       the current scaling period
8    var currMetrics = currentMetrics(m[:][:], measurementIndex,
       pMeasurementindex)
9
10   //If the Current Metrics is greater than Desired Metrics,
11   //this means that we have to scale up to meet our target metrics.
12   mustScaleUP := currMetrics > desiredMetrics
```

```
13  mustScaleDown := currMetrics <= desiredMetrics
14
15  if mustScaleUP {
16    //if previous decision was UP and Current Metrics is greater
    than Desired Metrics, calculate the HPA equation
17    if preDecision == "UP" {
18      //HPA equation: Ceiling (Current Replicas * Current Metrics /
    Desired Metrics).
19      desiredReplicas = int32(math.Ceil(float64(replicaCount) *
    currMetrics / desiredMetrics))
20      //Check policy (minReplica and maxReplica)
21      //After we calculate the desired replicas, we should check
    the autoscaling policy.
22      //also we should store the prvious decision whether is up or
    down.
23      //recall decision check function.
24      desiredReplicas = getDesiredNumReplica(desiredReplicas,
    replicaCount)
25      preDecision = "UP"
26      return desiredReplicas, preDecision
27
28      //if previous decision was Down and current metrics is
    greater than desired metrics, increase by 1
29    } else if preDecision == "DOWN" {
30      desiredReplicas = replicaCount + 1
31      //Check policy and get the previous decsion
32      desiredReplicas = getDesiredNumReplica(desiredReplicas,
    replicaCount)
33      preDecision = "UP"
34      return desiredReplicas, preDecision
35    }
36    //If the Current Metrics is less than Desired Metrics, this
    means that we have to scale down.
37  } else if mustScaleDown {
38    //if previous decision was UP and current metrics is less than
    desired metrics, decrease the current replica  by 1
39    if preDecision == "UP" {
40      desiredReplicas = replicaCount - 1
41      //Check policy and get the previous decsion
42      desiredReplicas = getDesiredNumReplica(desiredReplicas,
    replicaCount)
43      preDecision = "DOWN"
```

```
44        return desiredReplicas, preDecision
45
46        //if previous decision was DOWN and current metrics is less
      than desired metrics, Calculate the HPA
47      } else if preDecision == "DOWN" {
48        //HPA equation: Ceiling (Current Replicas * Current Metrics /
       Desired Metrics).
49        desiredReplicas = int32(math.Ceil(float64(replicaCount) *
      currMetrics / desiredMetrics))
50        //Check policy and get the previous decsion
51        desiredReplicas = getDesiredNumReplica(desiredReplicas,
      replicaCount)
52        preDecision = "DOWN"
53        return desiredReplicas, preDecision
54      }
55    }
56    //Everytime we returing the desired Replica and the new decision
57    return desiredReplicas, preDecision
58  }
```

Listing B.4: Scaling Algorithm 1

B.2.2 Scaling Using HPA Formula

Rolling average and moving window average are using the HPA formula.

```
1  //scalingAlgorithm function calculates the Desired Replicas based
      on Algorithm 1.
2  //Input : Current Metrics, Desired Metrics, Current Relicas, and
      Previous Decision.
3  //Output: Desired Replicas and updated Previous Decision.
4  func scalingAlgorithm(replicaCount int32, m [][10]measurementPod,
      measurementIndex int32, pMeasurementindex int32) int32 {
5    var desiredReplicas int32 = 0
6
7    //recal current metrics function to provide current metrics for
      the current scaling period
8    var currMetrics = currentMetrics(m[:][:], measurementIndex,
      pMeasurementindex)
9
10   //HPA equation: Ceiling (Current Replicas * Current Metrics /
      Desired Metrics).
```

```
11    desiredReplicas = int32(math.Ceil(float64(replicaCount) *
       currMetrics / desiredMetrics))
12    //Check policy (minReplica and maxReplica)
13    //After we calculate the desired replicas, we should check the
       autoscaling policy.
14    //also we should store the prvious decision whether is up or down
       .
15    //recall decision check function
16    desiredReplicas = getDesiredNumReplica(desiredReplicas,
       replicaCount)
17    fmt.Println("Desired Replica = ", desiredReplicas, " Current
       Metrics = ", currMetrics, " Current Replica = ", replicaCount)
18    return desiredReplicas
19 }
```

Listing B.5: Scaling Algorithm 1

B.3 Scaling Policy Function

```
1  //Check policy and get the previous decsion
2  //After we calculate the desired replicas, we should check the
       autoscaling policy.
3  //Desired Replica cannot be greater than the Max Replica.
4  //Desired Replica cannot be less than the Min Replica.
5  func getDesiredNumReplica(desiredReplica int32, replicaCount int32)
       int32 {
6    if desiredReplica > maxReplica {
7      desiredReplica = maxReplica
8    } else if desiredReplica < minReplica {
9      desiredReplica = minReplica
10   }
11   return desiredReplica
12 }
```

Listing B.6: Check Scaling Policy

B.4 Poll Replica Function

```
1  //This function is called to check for the difference between
       desired and current replica count
2  //The current replica should be updated in the next time interval.
```

```
//The number of containers should be installed and running.
//if the current replica is not updated, we should wait until the deployment updated by scale up or down based on the desired replica.
func PollReplicas(desiredReplicaCount int32, currReplicaCount int32, kube_cs *kubernetes.Clientset) {
  fmt.Println("THIS WILL SHIFT THE TIME LINE...")
  for {
    fmt.Println("Current Replica = ", currReplicaCount, "Desired Replica", desiredReplicaCount)
    time.Sleep(1 * time.Second)
    //Updating the current replica in the deployment.
    phpDeployment, err := kube_cs.AppsV1().Deployments("default").GetScale(context.TODO(), "php-apache", metav1.GetOptions{})
    if err != nil {
      fmt.Println("Error:", err)
      return
    }
    //current replica count is updated by calling the phpdeployment and checked for the synchronacy
    currReplicaCount = phpDeployment.Status.Replicas
    if desiredReplicaCount == currReplicaCount {
      fmt.Println("System in Sync")
      return
    }
  }
}
```

Listing B.7: Poll Replica

B.5 Find Duplicate Measurement Function

```
//This function is used to indicate if there is any duplicate value
//Input: measurement index, current pod name, current metrics time stamp, previous measurement.
//Output: the container number in the current measurement that has the same pod name and same timestamp.
func FindPreviousIndexDuplicateContainer(m [][10]measurementPod, index int32, name string, timestamp int64) int32 {
  //check each container in a measurement
  for i := 0; i < int(maxReplica); i++ {
    //if the pod name is equal to prvious name AND if the timestamp
```

```
    is equal to the previous measurement
    //return the number of container for the next measurement int32(i).
    mustCheckDuplicateMeasurement := name == m[index-1][i].PodName && int64(m[index-1][i].TimeStamp) == timestamp
    if mustCheckDuplicateMeasurement {
      return int32(i)
    }
  }
  //if the values does not match with the previous measurement, return -1
  return -1
}
```

Listing B.8: Duplicate Measurement

B.6 Updating Measurement Function

```
//This function is used to indicate if there is any duplicate value
//Input: measurement index, current pod name, current metrics time stamp, previous measurement.
//Output: the container number in the current measurement that has the same pod name and same timestamp.
//This function is to get the pods' information form kubelet through the metrics server
func updateMetricsInArray(m [][10]measurementPod, measurementIndex int32, containerIndex int32,
  podMetric v1beta1.PodMetrics, container v1beta1.ContainerMetrics, isAccountedFor bool) {
  //These value we don't have any control
  //metricTimestamp is timestamp of kubelet (we consider metrics timestamp to be 50 sec.)
  //CreationTimestamp is the polling time (here is set to be 10 second)
  m[measurementIndex][containerIndex].PodName = podMetric.ObjectMeta.Name
  m[measurementIndex][containerIndex].WindowSize = podMetric.Window.Duration.Seconds()
  m[measurementIndex][containerIndex].TimeStamp = podMetric.Timestamp.Time.Unix()
  m[measurementIndex][containerIndex].CreationTimestamp = podMetric.CreationTimestamp.Time.Unix()
```

```
14    m[measurementIndex][containerIndex].CPU = container.Usage.Cpu().
       ToDec().AsApproximateFloat64()
15    //We are controlling this value by checking if the provided
       measurment has been taken into account or not
16    //True: we already take it into account
17    //False: we didn't take it yet. (considering it is not accounted
       for calculation of CPU)
18    m[measurementIndex][containerIndex].isAccountedFor =
       isAccountedFor
19  }
```

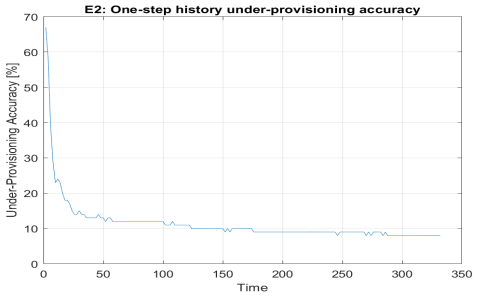
Listing B.9: Updating Measurement

APPENDIX

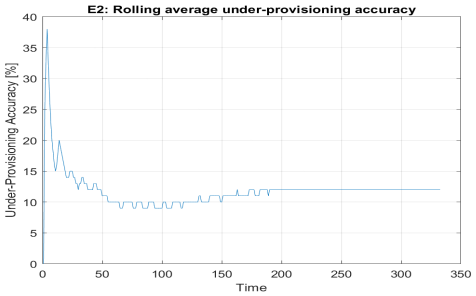
C

EXPERIMENTS RESULTS

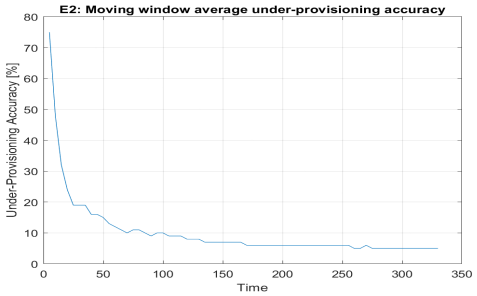
This appendix includes all the other three experiments figures that are similar to *E1* in section 4.3.



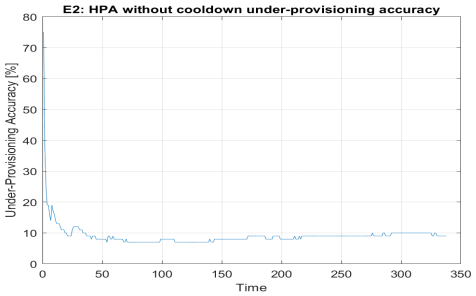
(a) One-step history Algorithm Under-Provisioning Accuracy



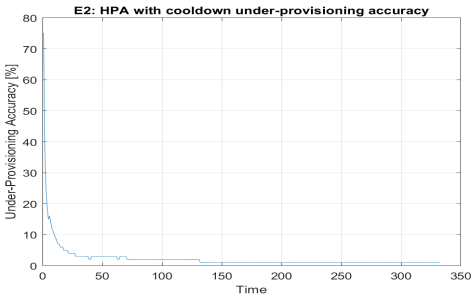
(b) Rolling Average Algorithm Under-Provisioning Accuracy



(c) Moving Window Average Algorithm Under-Provisioning Accuracy

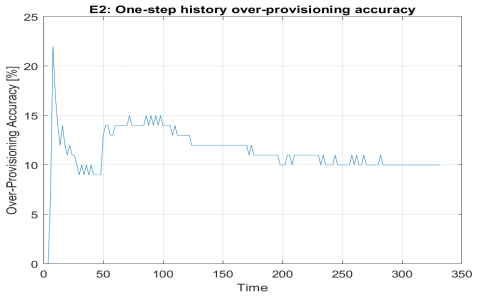


(d) HPA with cooling down Algorithm Under-Provisioning Accuracy

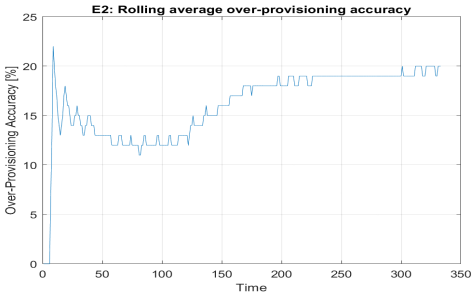


(e) HPA without cooling down Algorithm Under-Provisioning Accuracy

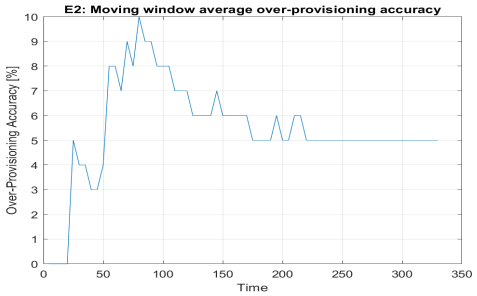
Figure C.1: Comparison of The Under-Provisioning Accuracy Percentage for All Algorithms (E2: NASA Dataset)



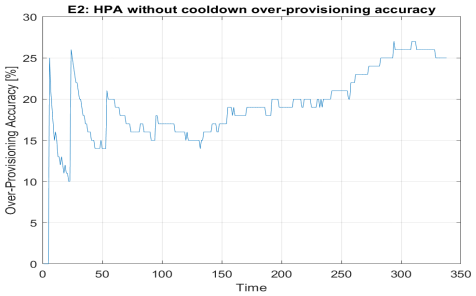
(a) One-step history Algorithm Over-Provisioning Accuracy



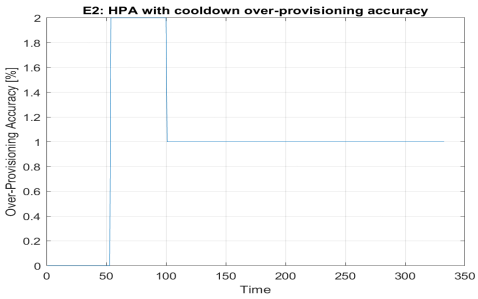
(b) Rolling Average Algorithm Over-Provisioning Accuracy



(c) Window Algorithm Over-Provisioning Accuracy

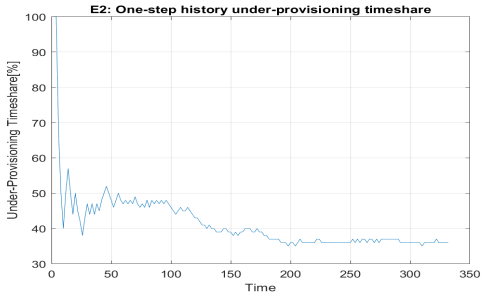


(d) HPA with cooling down Algorithm Over-Provisioning Accuracy

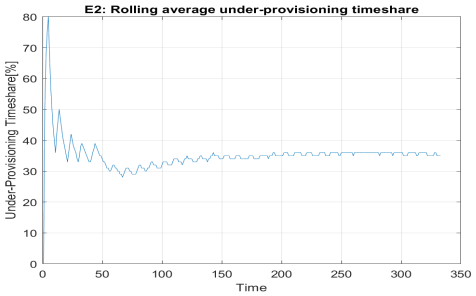


(e) HPA without cooling down Algorithm Over-Provisioning Accuracy

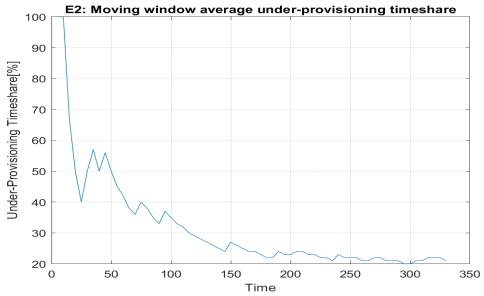
Figure C.2: Comparison of The Over-Provisioning Accuracy Percentage for All Algorithms (E2: NASA Dataset)



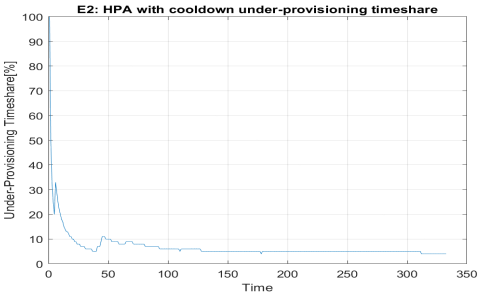
(a) One-step history Algorithm Under-Provisioning Timeshare



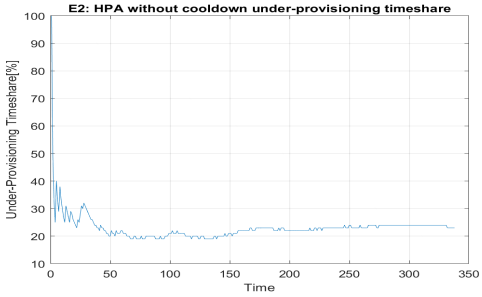
(b) Rolling Average Algorithm Under-Provisioning Timeshare



(c) Moving Window Average Algorithm Under-Provisioning Timeshare

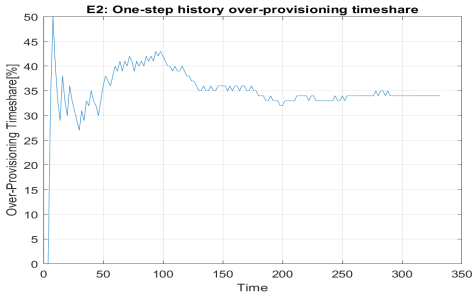


(d) HPA with cooling down Algorithm Under-Provisioning Timeshare

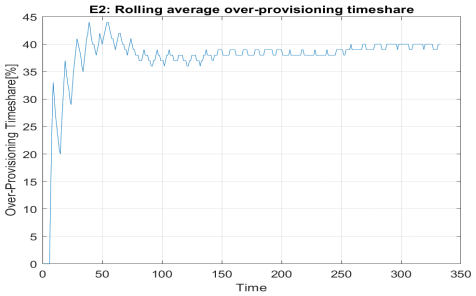


(e) HPA without cooling down Algorithm Under-Provisioning Timeshare

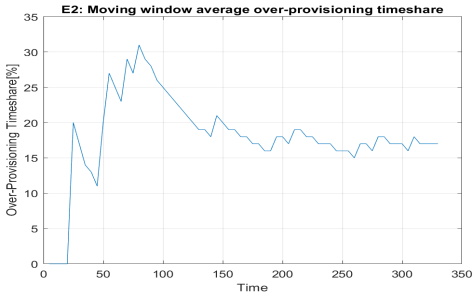
Figure C.3: Comparison of The Under-Provisioning Timeshare Percentage for All Algorithms (E2: NASA Dataset)



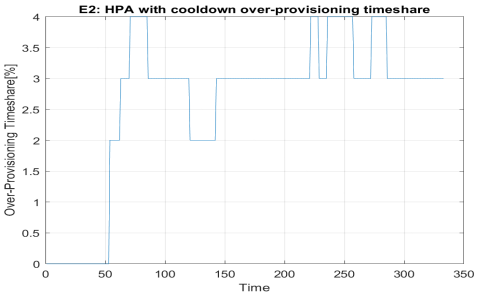
(a) One-step history Algorithm Over-Provisioning Timeshare



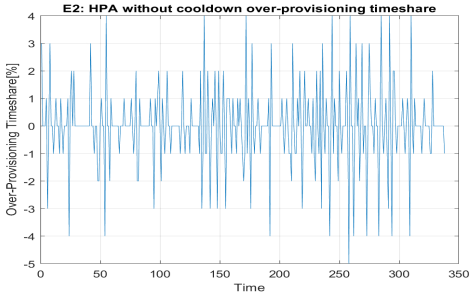
(b) Rolling Average Algorithm Over-Provisioning Timeshare



(c) Moving Window Average Algorithm Over-Provisioning Timeshare

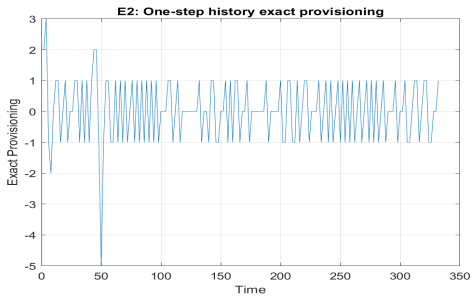


(d) HPA with cooling down Algorithm Over-Provisioning Timeshare

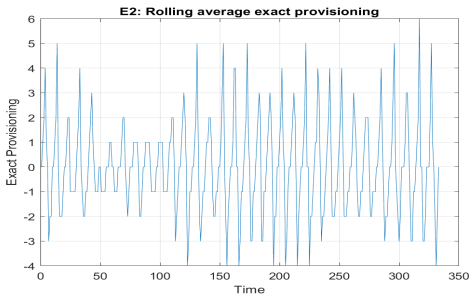


(e) HPA withoout cooling down Algorithm Over-Provisioning Timeshare

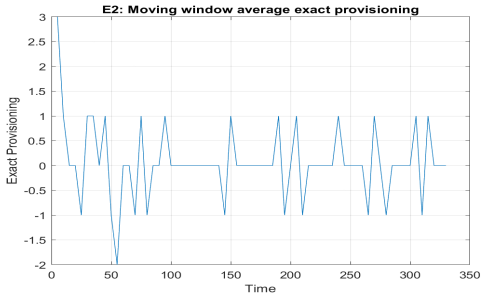
Figure C.4: Comparison of The Over-Provisioning Timeshare Percentage for All Algorithms (E2: NASA Dataset)



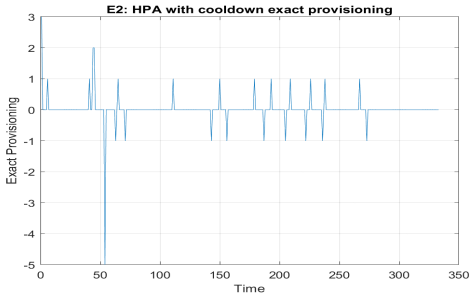
(a) One-step history Algorithm Exact Provisioning



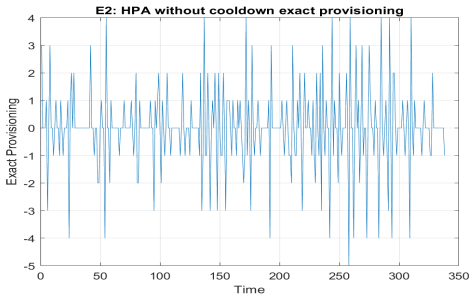
(b) Rolling Average Algorithm Exact Provisioning



(c) Moving Window Average Algorithm Exact Provisioning

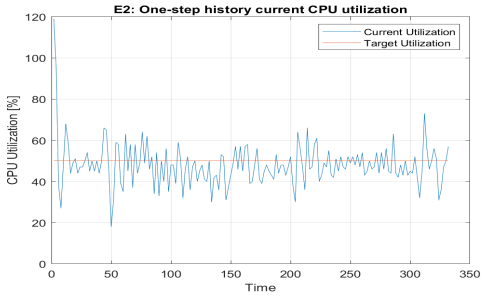


(d) HPA with cooling down Algorithm Exact Provisioning

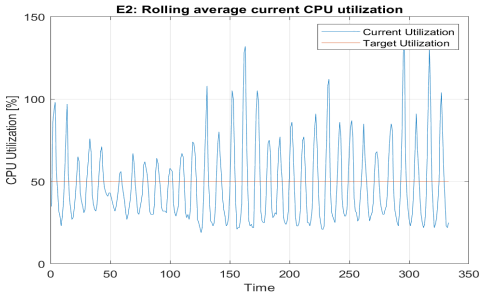


(e) HPA without cooling down Algorithm Exact Provisioning

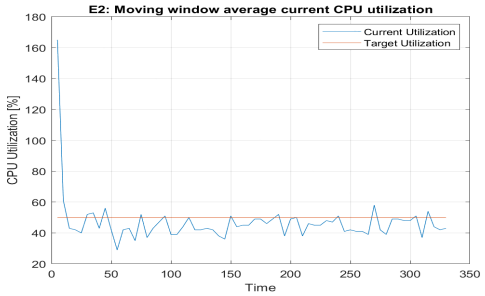
Figure C.5: Comparison of The Exact Provisioning for All Algorithms (E2: NASA Dataset)



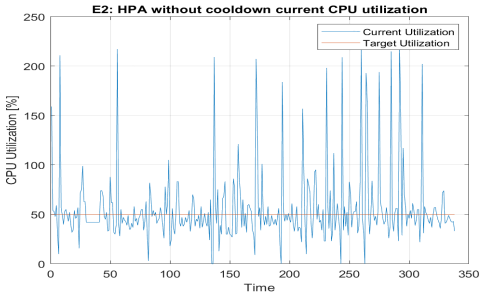
(a) One-step history Algorithm Current Metrics



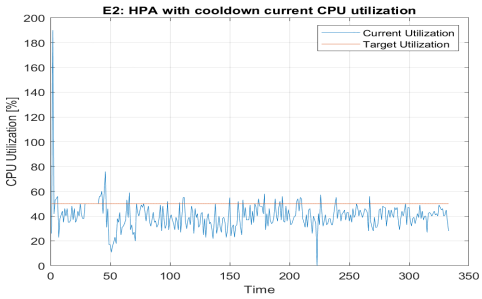
(b) Rolling Average Algorithm Current Metrics



(c) Moving Window Average Algorithm Current Metrics

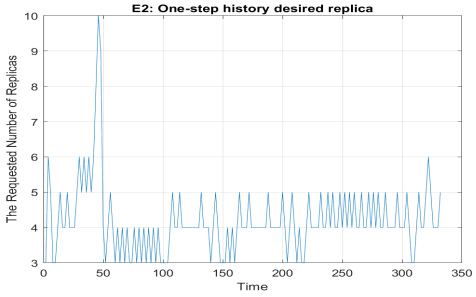


(d) HPA with cooling down Algorithm Current Metrics

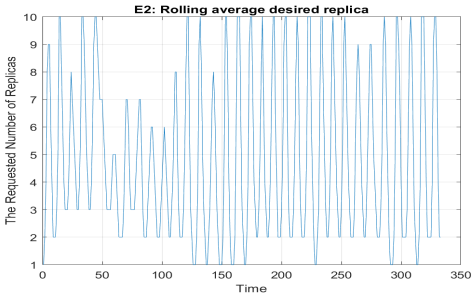


(e) HPA without cooling down Algorithm Current Metrics

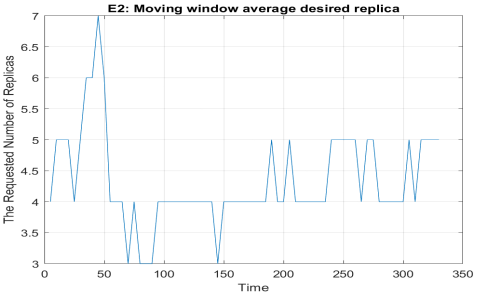
Figure C.6: Comparison of The CPU Utilization Percentage for All Algorithms (E2: NASA Dataset)



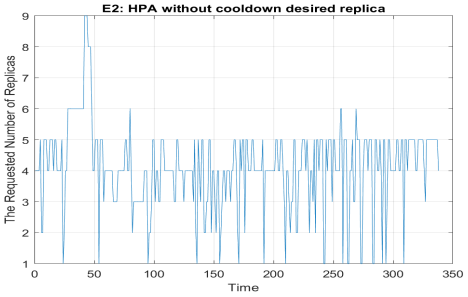
(a) One-step history Algorithm Desired Replica



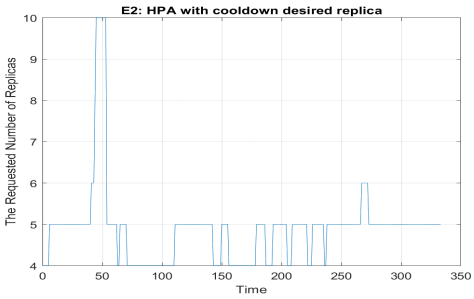
(b) Rolling Average Algorithm Desired Replica



(c) Moving Window Average Algorithm Desired Replica

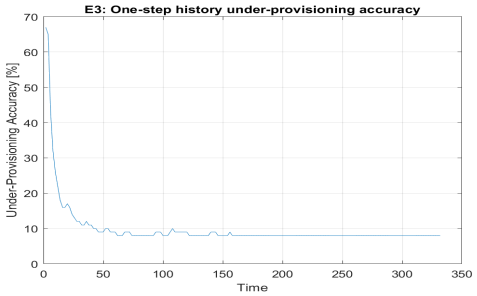


(d) HPA with cooling down Algorithm Desired Replica

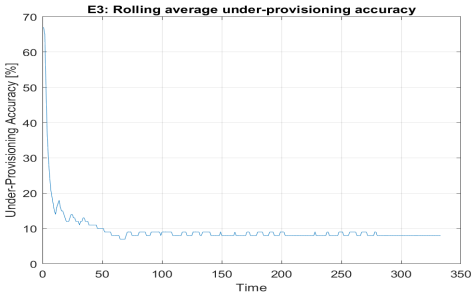


(e) HPA without cooling down Algorithm Desired Replica

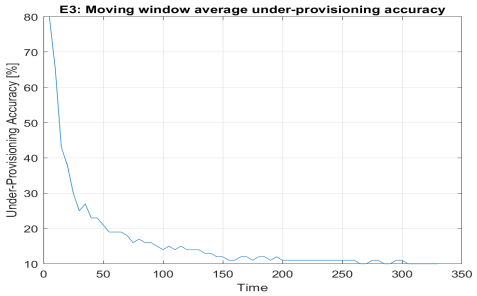
Figure C.7: Comparison of The Desired Replica for All Algorithms (E2: NASA Dataset)



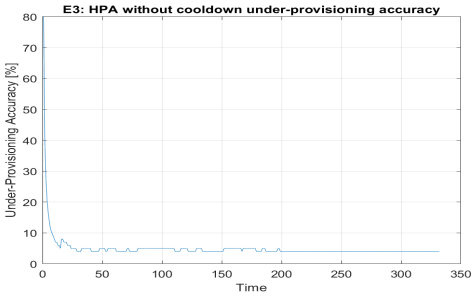
(a) One-step history Algorithm Under-Provisioning Accuracy



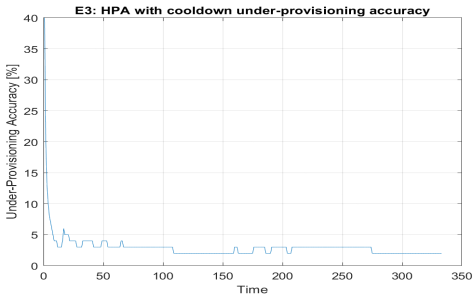
(b) Rolling Average Algorithm Under-Provisioning Accuracy



(c) Moving Window Average Algorithm Under-Provisioning Accuracy

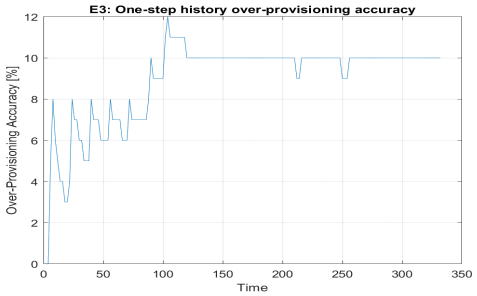


(d) HPA with cooling down Algorithm Under-Provisioning Accuracy

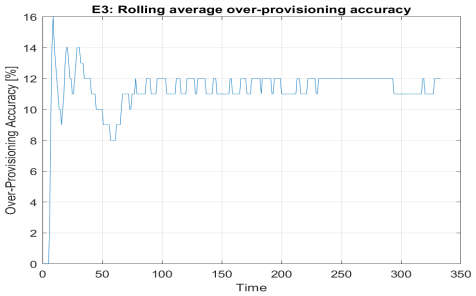


(e) HPA without cooling down Algorithm Under-Provisioning Accuracy

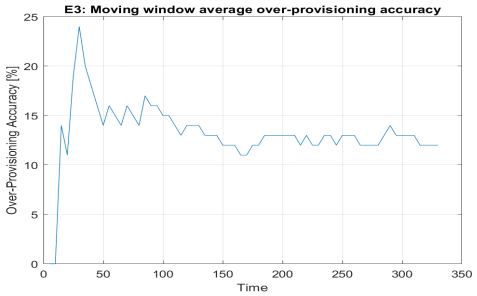
Figure C.8: Comparison of The Under-Provisioning Accuracy Percentage for All Algorithms (E3: FIFA Random)



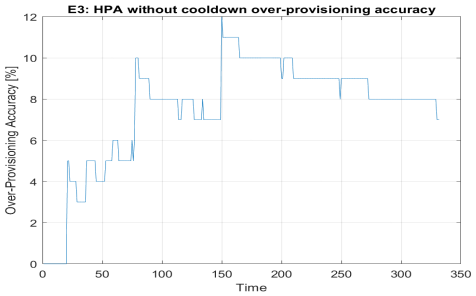
(a) One-step history Algorithm Over-Provisioning Accuracy



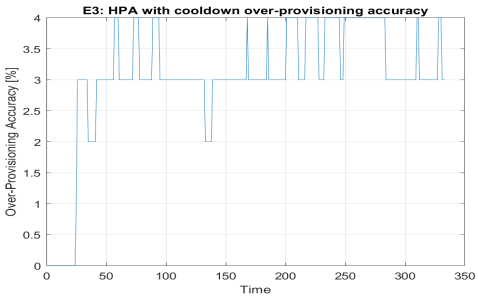
(b) Rolling Average Algorithm Over-Provisioning Accuracy



(c) Window Algorithm Over-Provisioning Accuracy

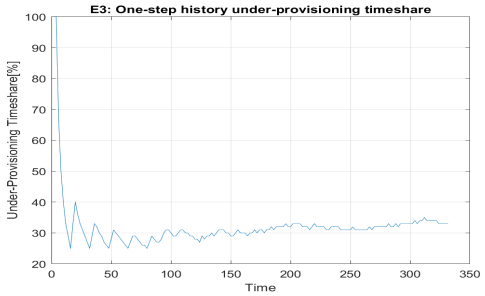


(d) HPA with cooling down Algorithm Over-Provisioning Accuracy

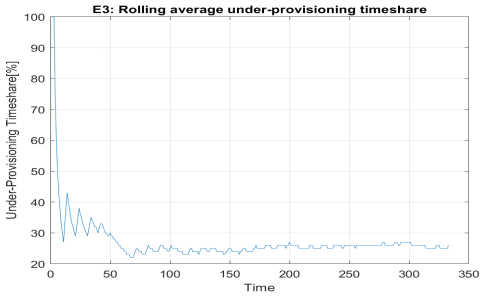


(e) HPA without cooling down Algorithm Over-Provisioning Accuracy

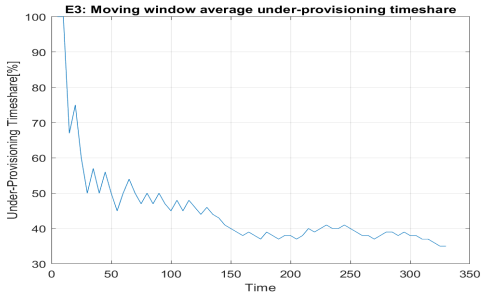
Figure C.9: Comparison of The Over-Provisioning Accuracy Percentage for All Algorithms (E3: FIFA Random)



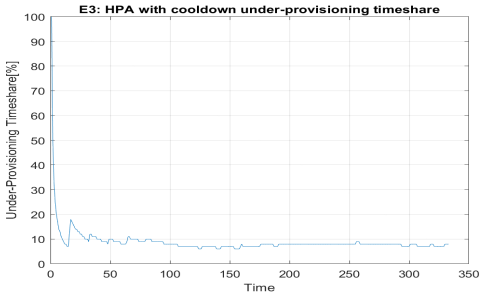
(a) One-step history Algorithm Under-Provisioning Timeshare



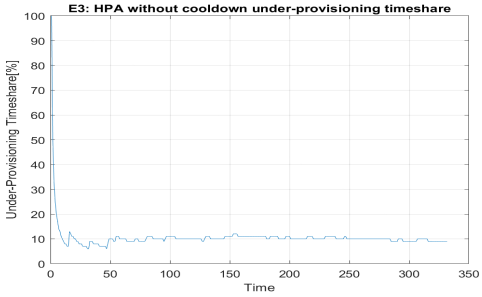
(b) Rolling Average Algorithm Under-Provisioning Timeshare



(c) Moving Window Average Algorithm Under-Provisioning Timeshare

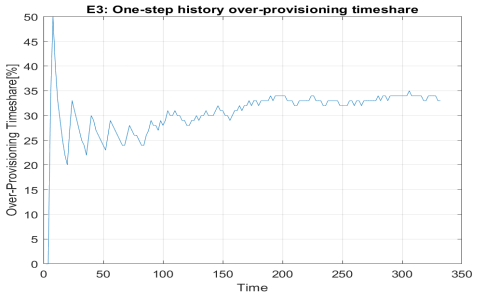


(d) HPA with cooling down Algorithm Under-Provisioning Timeshare

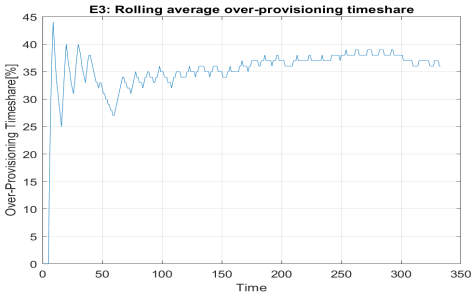


(e) HPA without cooling down Algorithm Under-Provisioning Timeshare

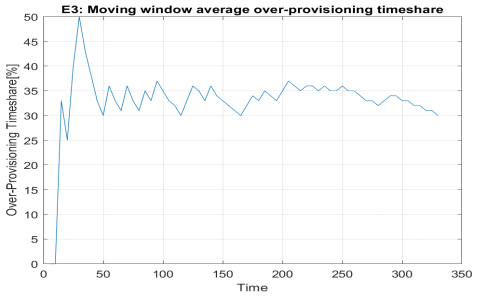
Figure C.10: Comparison of The Under-Provisioning Timeshare Percentage for All Algorithms (E3: FIFA Random)



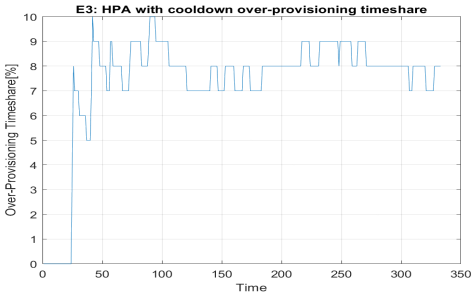
(a) One-step history Algorithm Over-Provisioning Timeshare



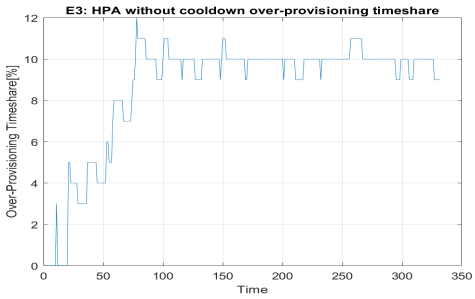
(b) Rolling Average Algorithm Over-Provisioning Timeshare



(c) Moving Window Average Algorithm Over-Provisioning Timeshare

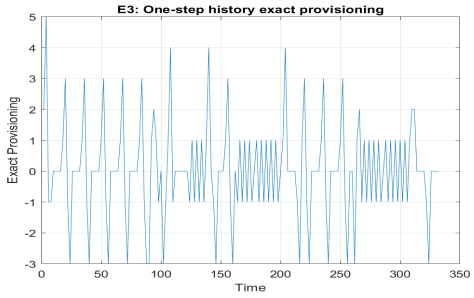


(d) HPA with cooling down Algorithm Over-Provisioning Timeshare

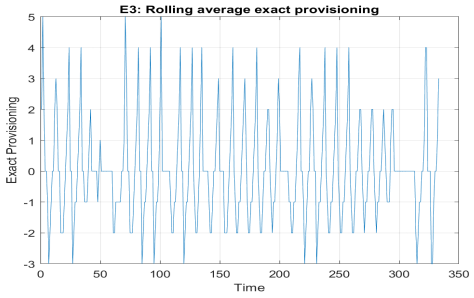


(e) HPA withoout cooling down Algorithm Over-Provisioning Timeshare

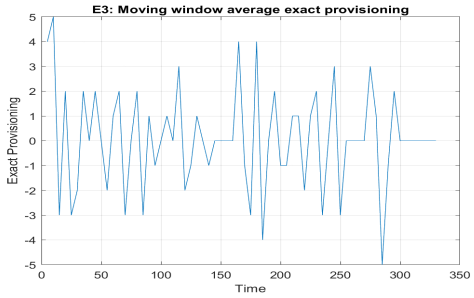
Figure C.11: Comparison of The Over-Provisioning Timeshare Percentage for All Algorithms (E3: FIFA Random)



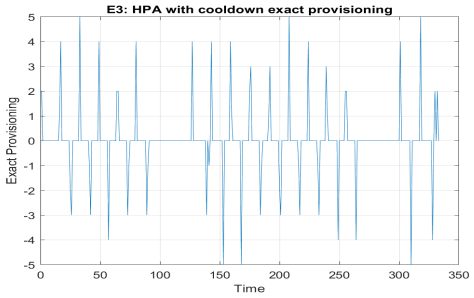
(a) One-step history Algorithm Exact Provisioning



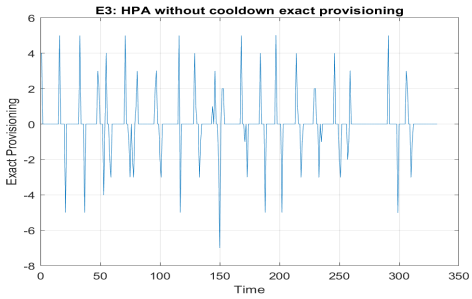
(b) Rolling Average Algorithm Exact Provisioning



(c) Moving Window Average Algorithm Exact Provisioning

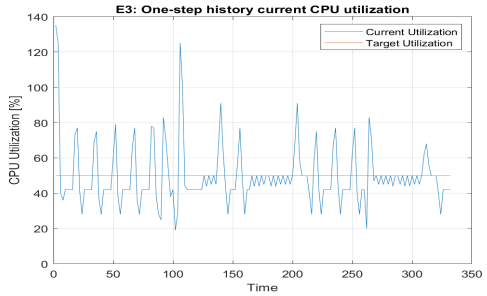


(d) HPA with cooling down Algorithm Exact Provisioning

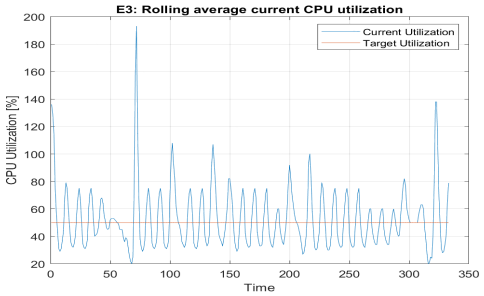


(e) HPA without cooling down Algorithm Exact Provisioning

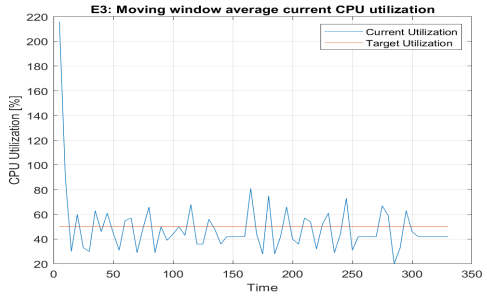
Figure C.12: Comparison of The Exact Provisioning for All Algorithms (E3: FIFA Random)



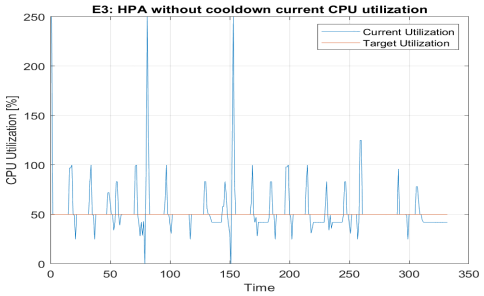
(a) One-step history Algorithm Current Metrics



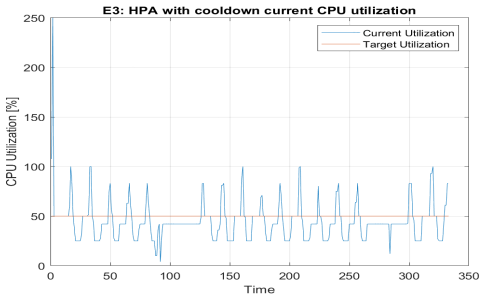
(b) Rolling Average Algorithm Current Metrics



(c) Moving Window Average Algorithm Current Metrics

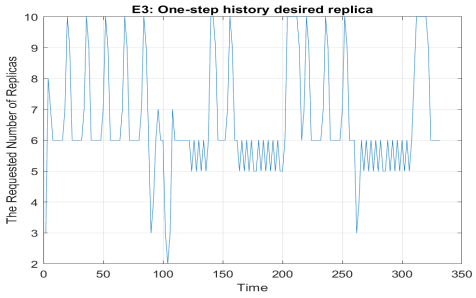


(d) HPA with cooling down Algorithm Current Metrics

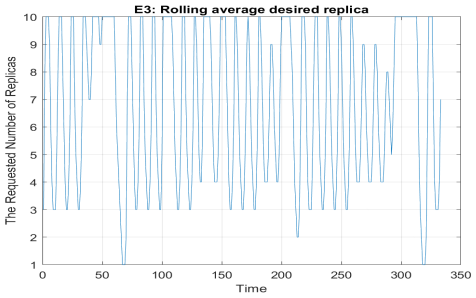


(e) HPA without cooling down Algorithm Current Metrics

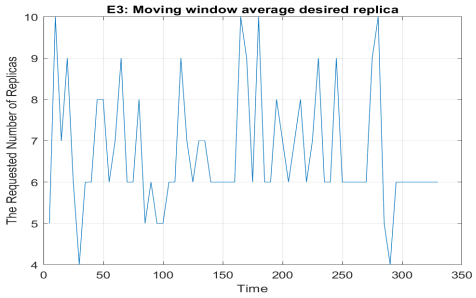
Figure C.13: Comparison of The CPU Utilization Percentage for All Algorithms (E3: FIFA Random)



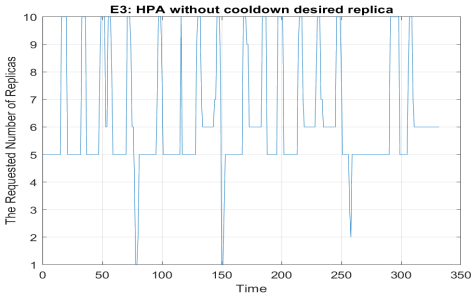
(a) One-step history Algorithm Desired Replica



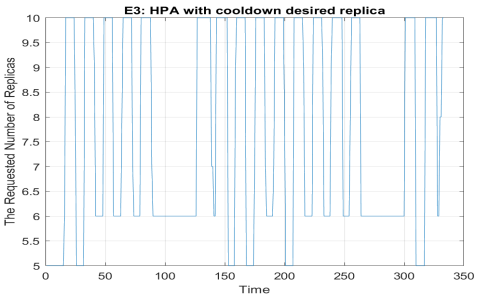
(b) Rolling Average Algorithm Desired Replica



(c) Moving Window Average Algorithm Desired Replica

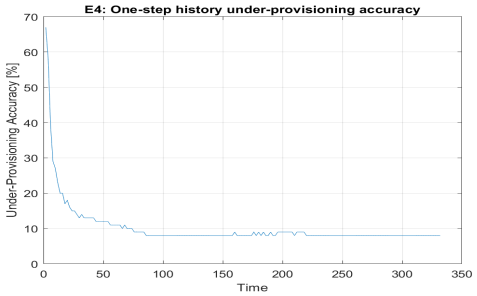


(d) HPA with cooling down Algorithm Desired Replica

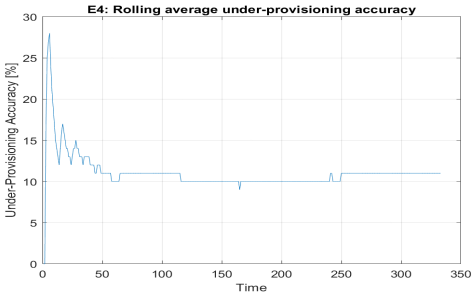


(e) HPA without cooling down Algorithm Desired Replica

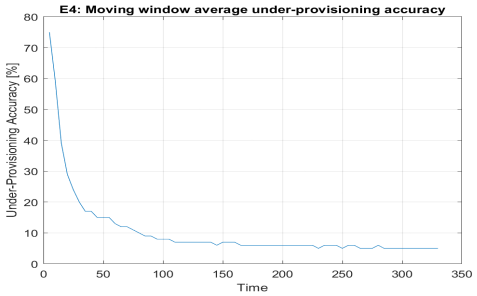
Figure C.14: Comparison of The Desired Replica for All Algorithms (E3: FIFA Random)



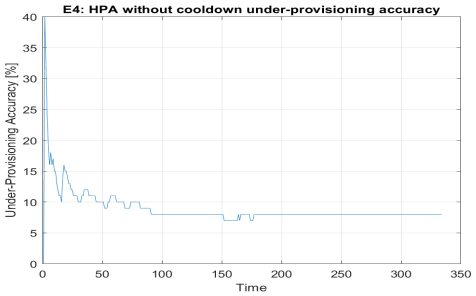
(a) One-step history Algorithm Under-Provisioning Accuracy



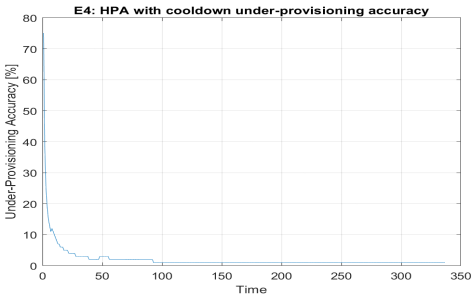
(b) Rolling Average Algorithm Under-Provisioning Accuracy



(c) Moving Window Average Algorithm Under-Provisioning Accuracy

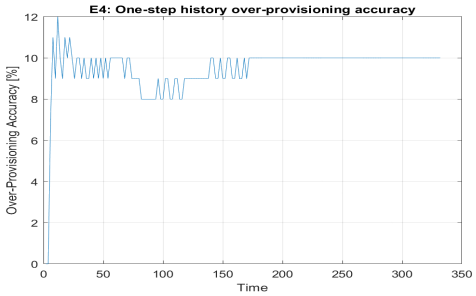


(d) HPA with cooling down Algorithm Under-Provisioning Accuracy

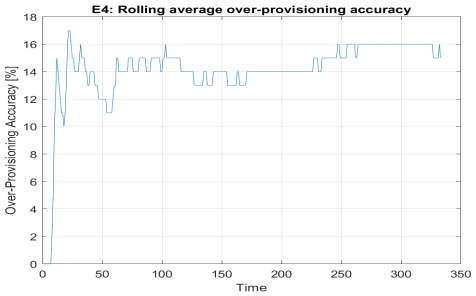


(e) HPA without cooling down Algorithm Under-Provisioning Accuracy

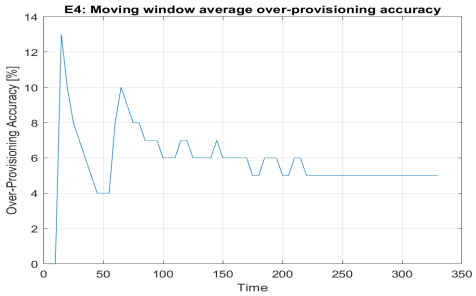
Figure C.15: Comparison of The Under-Provisioning Accuracy Percentage for All Algorithms (E4: NASA Repeat)



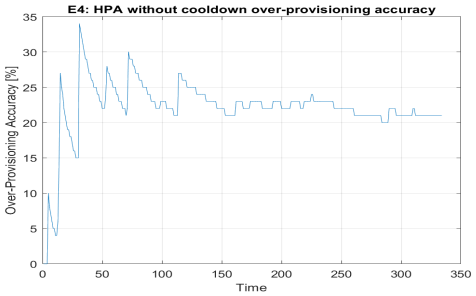
(a) One-step history Algorithm Over-Provisioning Accuracy



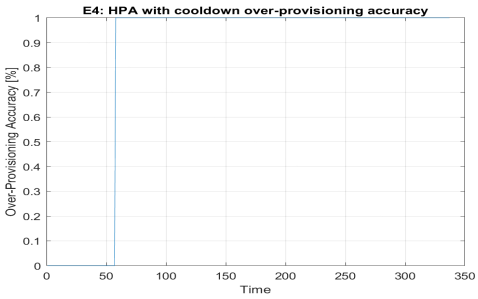
(b) Rolling Average Algorithm Over-Provisioning Accuracy



(c) Window Algorithm Over-Provisioning Accuracy

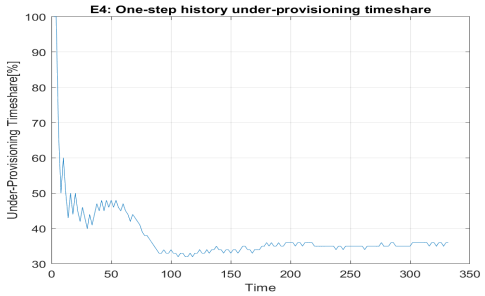


(d) HPA with cooling down Algorithm Over-Provisioning Accuracy

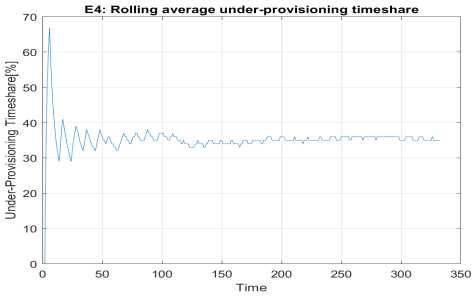


(e) HPA without cooling down Algorithm Over-Provisioning Accuracy

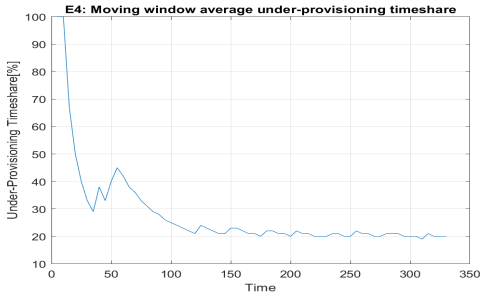
Figure C.16: Comparison of The Over-Provisioning Accuracy Percentage for All Algorithms (E4: NASA Repeat)



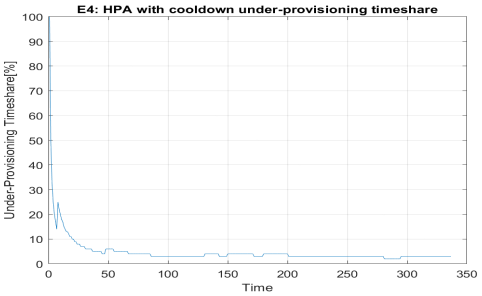
(a) One-step history Algorithm Under-Provisioning Timeshare



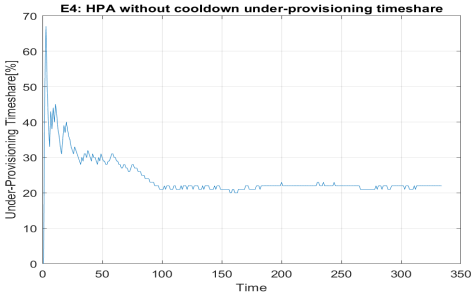
(b) Rolling Average Algorithm Under-Provisioning Timeshare



(c) Moving Window Average Algorithm Under-Provisioning Timeshare

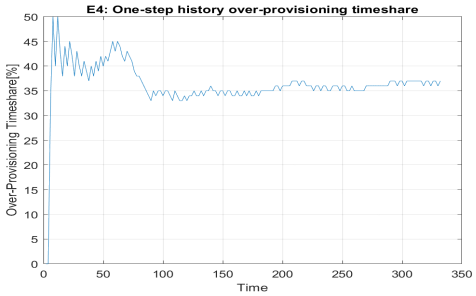


(d) HPA with cooling down Algorithm Under-Provisioning Timeshare

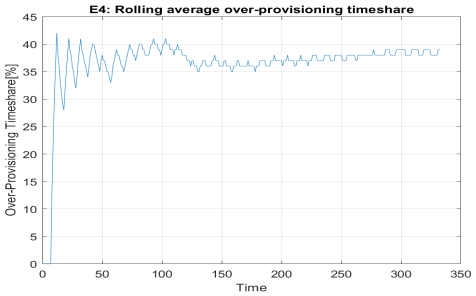


(e) HPA without cooling down Algorithm Under-Provisioning Timeshare

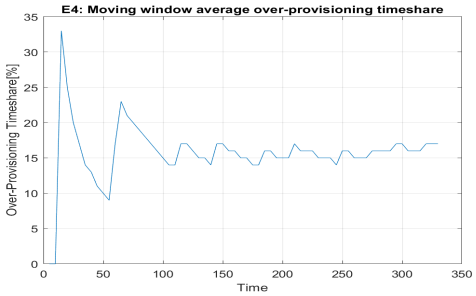
Figure C.17: Comparison of The Under-Provisioning Timeshare Percentage for All Algorithms (E4: NASA Repeat)



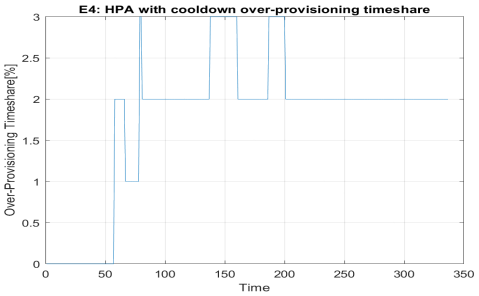
(a) One-step history Algorithm Over-Provisioning Timeshare



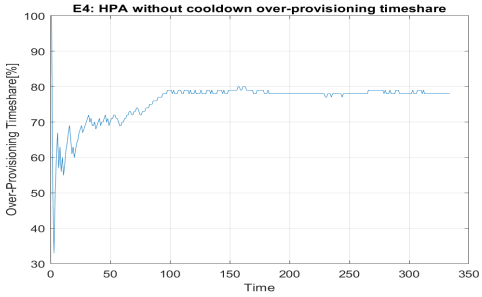
(b) Rolling Average Algorithm Over-Provisioning Timeshare



(c) Moving Window Average Algorithm Over-Provisioning Timeshare

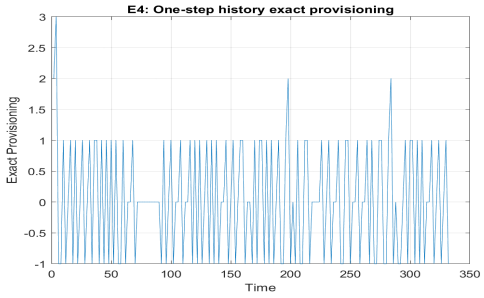


(d) HPA with cooling down Algorithm Over-Provisioning Timeshare

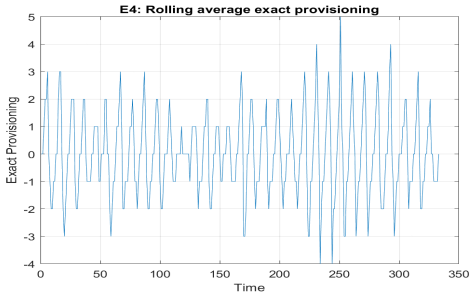


(e) HPA withoout cooling down Algorithm Over-Provisioning Timeshare

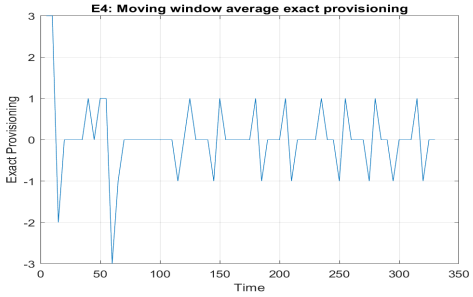
Figure C.18: Comparison of The Over-Provisioning Timeshare Percentage for All Algorithms (E4: NASA Repeat)



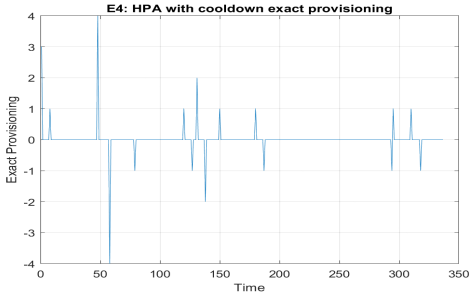
(a) One-step history Algorithm Exact Provisioning



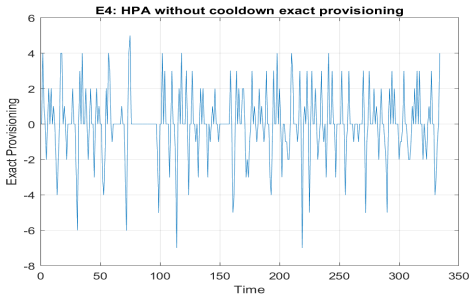
(b) Rolling Average Algorithm Exact Provisioning



(c) Moving Window Average Algorithm Exact Provisioning

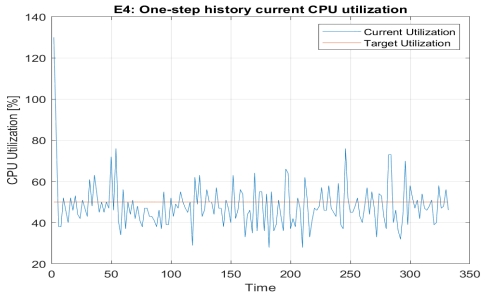


(d) HPA with cooling down Algorithm Exact Provisioning

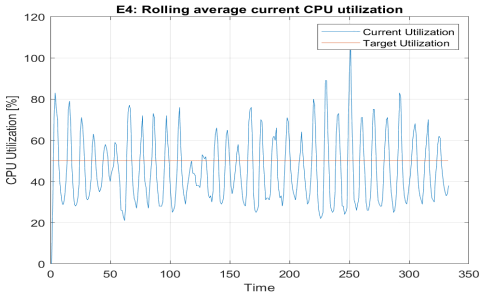


(e) HPA without cooling down Algorithm Exact Provisioning

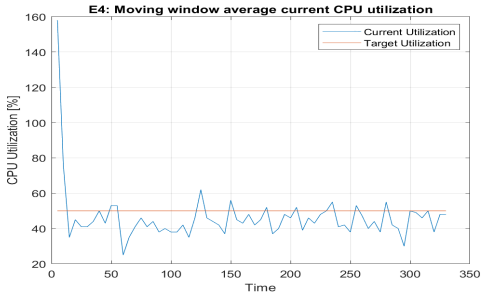
Figure C.19: Comparison of The Exact Provisioning for All Algorithms (E4: NASA Repeat)



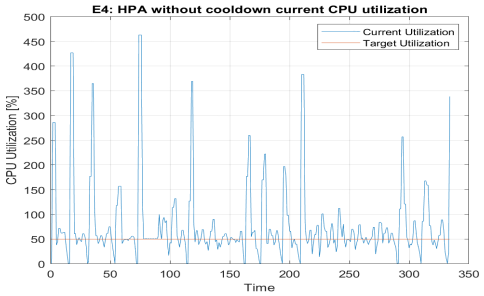
(a) One-step history Algorithm Current Metrics



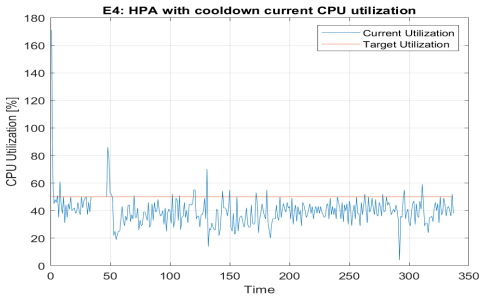
(b) Rolling Average Algorithm Current Metrics



(c) Moving Window Average Algorithm Current Metrics

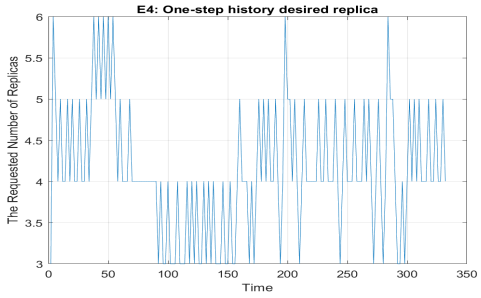


(d) HPA with cooling down Algorithm Current Metrics

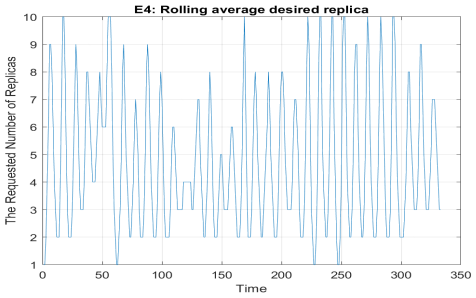


(e) HPA without cooling down Algorithm Current Metrics

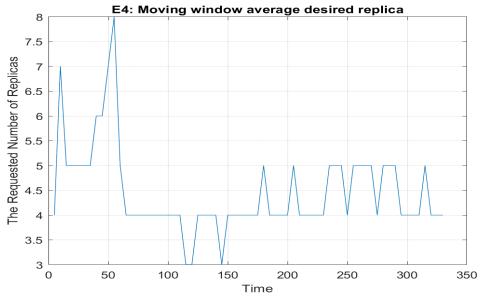
Figure C.20: Comparison of The CPU Utilization Percentage for All Algorithms (E4: NASA Repeat)



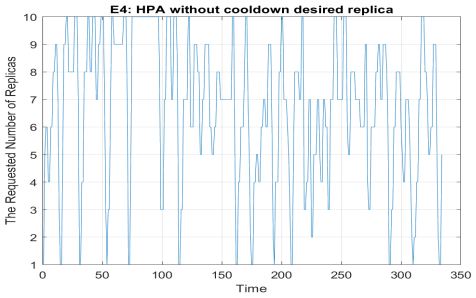
(a) One-step history Algorithm Desired Replica



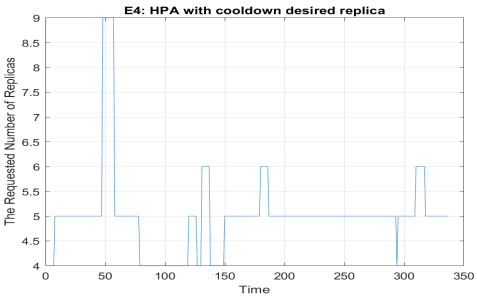
(b) Rolling Average Algorithm Desired Replica



(c) Moving Window Average Algorithm Desired Replica



(d) HPA with cooling down Algorithm Desired Replica



(e) HPA without cooling down Algorithm Desired Replica

Figure C.21: Comparison of The Desired Replica for All Algorithms (E4: NASA Repeat)