Optimization of Horizontal Pod Autoscaling for Containerized Applications in Kubernetes using Time Series Analysis

DISSERTATION

Submitted in accordance with the requirements of the degree of

Integrated Master of Technology

in

Mathematics and Computing

By

Sujatro Majumder

Admission Number: 17JE003117

Under the supervision of

Dr. Shuvashree Mondal

Department of Mathematics & Computing

INDIAN INSTITUTE OF TECHNOLOGY (INDIAN SCHOOL OF MINES), DHANBAD May 2022

DECLARATION

The Dissertation titled “Optimization of Horizontal Pod Autoscaling for Containerized Applications in Kubernetes using Time Series Analysis” is a presentation of my original research work and is not copied or reproduced or imitated from any other person's published or unpublished work. Wherever contributions of others are involved, every effort is made to indicate this clearly, with due reference to the literature, and acknowledgement of collaborative research and discussions, as may be applicable. Every effort is made to give proper citation to the published/unpublished work of others if it is referred to in the Dissertation.

To eliminate the scope of academic misconduct and plagiarism, I declare that I have read and understood the UGC (Promotion of Academic Integrity and Prevention of Plagiarism in Higher Educational Institutions) Regulations, 2018. These Regulations have been notified in The Official Gazette of India on 31st July, 2018. I confirm that this Dissertation has been checked with the online plagiarism detector tool Turnitin (http:///www.turnitin.com) provided by IIT (ISM) Dhanbad and a copy of the summary report/report, showing Similarities in content and its potential source (if any), generated online through Turnitin is enclosed at the end of the Dissertation. I hereby declare that the Dissertation shows less than 10% similarity as per the report generated by Turnitin and meets the standards as per MHRD/UGC Regulations and rules of the Institute regarding plagiarism.

I further state that no part of the Dissertation and its data will be published without the consent of my guide. I also confirm that this Dissertation work, carried out under the guidance of Dr. Shuvashree Mondal, Assistant Professor, Department of Mathematics and Computing, has not been previously submitted for assessment for the purpose of award of a Degree either at IIT (ISM) Dhanbad or elsewhere to the best of my knowledge and belief.

Sujatro Majumder
Adm. No : 17JE003117
Integrated M.Tech.
Department of Mathematics and Computing

Forwarded By:
Dr. Shuvashree Mondal
Assistant Professor
Department of Mathematics and Computing

CERTIFICATE

This is to certify that **Mr Sujatro Majumder** (Admission No.: 17JE003117), a student of Integrated M.Tech. (Mathematics and Computing), Department of Mathematics and Computing, Indian Institute of Technology (Indian School of Mines), Dhanbad has worked under my guidance and completed his Dissertation entitled "**Optimization of Horizontal Pod Autoscaling for Containerized Applications in Kubernetes using Time Series Analysis**" in partial fulfilment of the award requirement of degree Integrated M.Tech. in Department of Mathematics and Computing from Indian Institute of Technology (Indian School of Mines), Dhanbad.

This work has not been submitted for any other degree, award, or distinction elsewhere to the best of my knowledge and belief. He is solely responsible for the technical data and information provided in this work.

Supervisor:
Dr. Shuvashree Mondal
Assistant Professor
Department of Mathematics and Computing
Indian Institute of Technology
(Indian School of Mines), Dhanbad

Forwarded by:
Dr. Ranjit Kumar Upadhyay
Professor and Head of Department
Department of Mathematics and Computing
Indian Institute of Technology
(Indian School of Mines), Dhanbad

Abstract

Name of the student: **Sujatro Majumder** Roll No: **17JE003117**

Degree for which submitted: **Integrated Master of Technology**

Department: **Department of Mathematics and Computing**

Thesis title: **Optimization of Horizontal Pod Autoscaling for Containerized Applications in Kubernetes using Time Series Analysis**

Thesis supervisor: **Dr. Shuvashree Mondal**

Month and year of thesis submission: **May, 2022**

With the rapid growth in cloud computing, businesses are rapidly switching to cloud platforms to meet their needs. Many companies are adopting the microservices architecture as it enables them to deploy, manage and scale their services in containerized applications in the cloud. Kubernetes is a popular orchestration tool which helps us manage these containerized services by automating deployment, scaling, and management through pods. Scalability becomes a major concern when a service faces variable workload throughout the day. We are proposing a proactive autoscaler which can act together with the existing pod autoscaler in Kubernetes. The study is not limited to Kubernetes—this autoscaling approach can also be implemented in any cloud platform for managing instances. With time series forecasting, we are determining the incoming workload beforehand, which will optimise cost and increase performance.

Acknowledgements

Foremost, I would like to express my gratitude to my mentor Dr. Shuvashree Mondal for her constant support and trust.

I would also thank my parents for their love and selflessness.

My sincere thanks to my batchmates and colleagues who helped me grasp the concepts I have worked on in this dissertation.

Contents

Abbreviations:

- VMs Virtual Machines
- K8s Kubernetes
- HPA Horizontal Pod Autoscaler
- VPA Vertical Pod Autoscale
- AWS Amazon Web Services
- AAS Amazon Auto Scaling
- EC2 Elastic Compute Cloud
- GCP Google Cloud Platform
- RNN Recurrent Neural Network
- LSTM Long Short Term Memory
- GRU Gated Recurrent Unit
- BiLSTM Bidirectional LSTM
- ARMA Autoregressive Moving Average
- ARIMA Autoregressive Integrated Moving Average
- SARIMAX Seasonal ARIMA with eXogenous factor

Chapter 1

Introduction

1.1 Objective

The objective of this dissertation is to improve the functioning of Horizontal Pod Autoscaling (HPA) in Kubernetes. We will explore the need for containerization and using microservices with Docker. We will study the architecture of Kubernetes and its vertical and horizontal autoscaling services. Next we aim to propose a “custom” pod autoscaler that would fit in the existing Kubernetes architecture. This proactive custom autoscaler shall be a time-series based autoscaler. We will discover some of the time-series forecasting models we have today and compare the results to find out which one gives the most suitable result. This model will be the core of our custom autoscaler component and it should work to enhance the working of HPA in managing pod deployment. This paper should provide a comprehensive guide to anyone trying to understand containerized applications and autoscaling features of Kubernetes.

1.2 Motivation behind Autoscaling

As the size and complexity of our applications have grown, so has the demand for containerized applications. Managing applications has become more cumbersome— to deploy new features, streamline workflows while following agile development methodologies. Using microservices architecture has helped us break an application into independent components and host them in containers like Docker. But it also requires manual intervention in deploying and scaling these services.

So we need an orchestration tool like Kubernetes to build these services which can handle multiple containers, schedule and scale them across different clusters, perform a health check, deploy replicas, and so on. As autoscaling is one of the most promising features a container orchestration tool can have, we have focussed on exploring the autoscaling services of

Kubernetes in this dissertation. By the end of this paper, we would like to conclude the ways we can improve the existing HPA component in Kubernetes.

1.3 Use Case: Netflix Scryer

Scryer is a predictive autoscaling engine designed by Netflix to provision/deprovision the AWS instances to handle traffic (requests received from their users). It is used alongside AWS Amazon Auto Scaling (AAS) to scale out instances to deliver uninterrupted service alongside reducing the cost.

Scryer is a proactive scaling engine, unlike AAS which is reactive in nature.[13] Through Scryer, Netflix decides on the number of AWS instances to handle the workload. While AAS reacts to real-time metrics and provisions the instances accordingly, Scryer predicts the needs beforehand (hence, proactive) and adjusts the instances.

Why Netflix went on to create its own autoscaling service was because AAS could not address some of their issues— like sudden spike/drop in viewers. Another issue was variable traffic patterns. Their data had daily and weekly seasonalities as people tend to use Netflix more during the night and on weekends. The data also had an upward trend which suggested their rapid growth over the time. Hence, to address all of these, Netflix came up with a predictive autoscaling engine called Scryer. Scryer has helped Netflix in improving cluster performance, reducing cost of EC2 instances and increasing service availability.

Chapter 2

Background

2.1 Virtualized environments

Containerization is one of the latest developments in the world of cloud computing. It has become an alternative to virtualization in recent years. Containerizing softwares enables developers to build, test and deploy applications across environments with little or no modification. In this section, we will explore the need for virtual machines and containers and how they often act together towards building production ready applications.

2.1.1 Need for Virtualization

A virtual machine (VM) is an emulation of an actual computer, hosted on a hypervisor on top of a physical machine. This hypervisor is a software layer which is responsible for creating these virtualized instances of each of the components that make up our machine. These include processors, RAM, storage, network cards, etc. By VMs, we generally mean machines that are performing hardware level virtualization.
Virtualization has been around for quite some time and is considered the base of the first generation of cloud computing. It is the basic foundation on which major cloud computing services such as Amazon EC2 and VMware vCloud are built.

2.1.2 Need for containerization

The problem with traditional approaches to software development is sometimes the code, due to library and resource dependencies, seem to work in a particular environment (a system/machine). However, the same code might result in bugs and errors when shifted to a

new environment. Examples of this might include anomalies in code when transferring a code from a Windows to Linux Operating System.

Containers are basically a type of operating system virtualization. A single container can be used to run anything from a large application to a small microservice. It bundles the application, its dependencies (libraries and other binaries), and configuration files into one package, running in an isolated environment.[1]

They are smaller (lightweight) than virtual machines, so they can be deployed much faster. The containers as well as the processes inside them are isolated, that is, they are unaware of the processes running in other containers unless specified explicitly. Containers are also very versatile, which means they are portable between different cloud platforms. All these advantages make containers a growing alternative to VMs in development environments. However, it's not necessary that VMs will be completely replaced by containers. VMs are still the favourite technology when it comes to provisioning infrastructure in an monolithic and enterprise applications.[2]

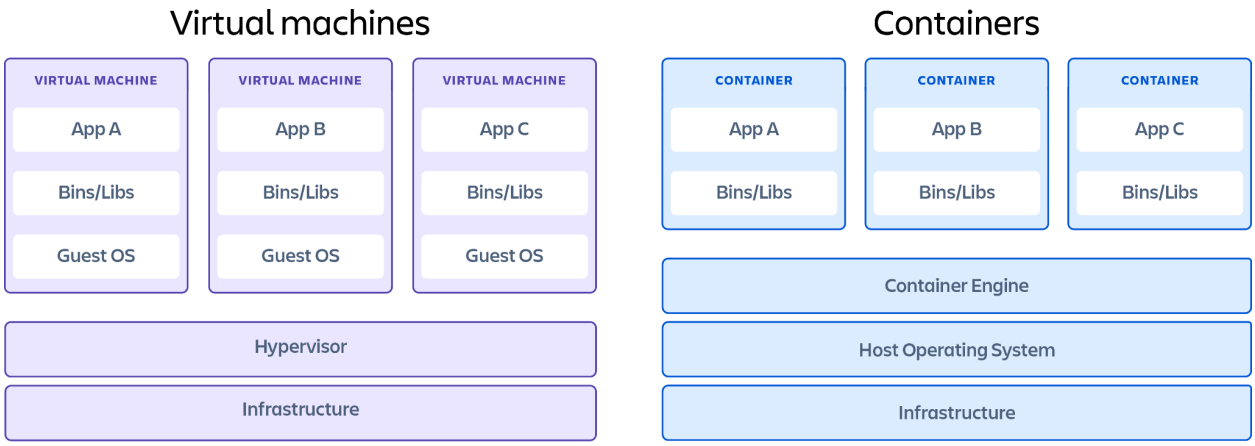

Figure 2.1: Virtual Machines vs Containers

Why containerization has become so popular is because it addresses the problem of "It works on my machine!". In DevOps for example, when multiple professionals collaborate, there's an increased chance of environmental conflicts, and this is where container engines come handy. [3] Through a container, a software can be isolated from its runtime environment and thus be deployed anywhere as they act as standalone units, each having different code and dependencies.

2.2 Docker

2.2.1 Introduction and Architecture

There are many container architectures available (Solaris, BSD jails, containerd and rkt) but by far the most popular has been Docker. It uses several Linux kernel features like namespaces and control groups to deliver its functionality. Docker uses namespaces to provide workspaces called containers. Namespace partitions the kernel resources (memory, storage, network) such that they can be used by these containers (processes) in an isolated environment. ControlGroups (or Cgroups in short) determines how much host machine resources to be given to containers. Thus, they provide a mechanism for aggregating/partitioning tasks into groups with specified behaviour.[4]

Docker has a client-server architecture. For example, when we ask docker to carry out some operation, say to start up a container, the docker client (*docker*) will translate this into a REST API call, and send it to docker daemon (*dockerd*). Docker daemon evaluates these requests and manages Docker objects (containers, volumes, etc) by interacting with the underlying OS.

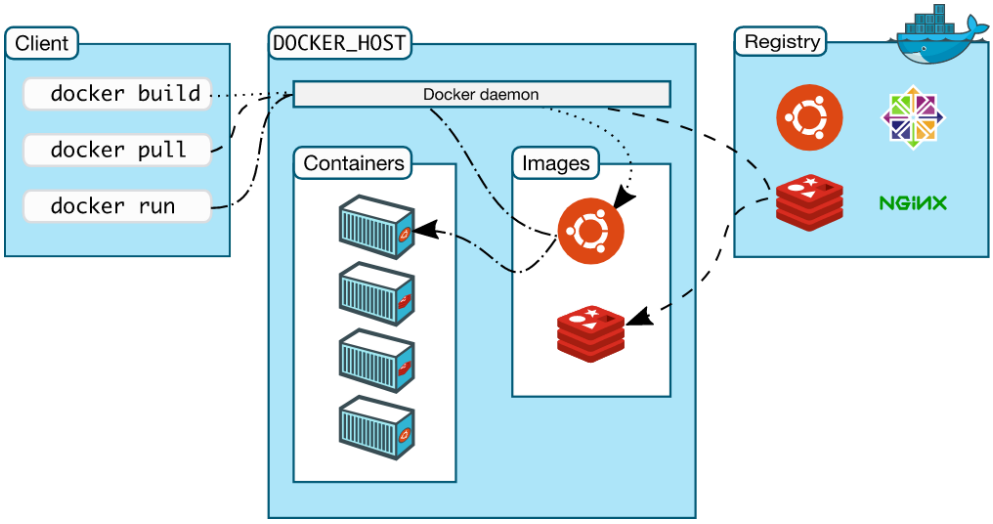

Figure 2.2: Docker architecture

2.2.2 Docker and Microservices

Microservices break an application into individually deployable services. The microservice architecture allows for each service to scale, deploy and survive independently without

disrupting other services. A microservices framework creates a massively scalable and distributed system, enabling more reliable CI/CD pipelines.[5] Organisations are increasingly adopting microservices because they want to enable faster application deployment and maintenance. The developers are dockerizing these microservices as the delivery and management of the apps become easier, making them deployable and scalable in an isolated manner.

2.3 Kubernetes

2.3.1 Introduction to Kubernetes

Now that we have understood the importance of containerization in agile development, let's try to understand why managing them properly can be a real challenge. In a production environment, we need these containers to be up and running with no downtime, parallelly scaling up and scaling out as the workload changes. For example, say we need to scale out during weekdays as the number of users for our application increases. Or maintaining replica sets for our services and ensuring there is no downtime. It would be convenient if these things are monitored by a system. This is where Kubernetes come into play. It is a container orchestration tool for deploying, managing and automating applications.[6] Thus, Kubernetes (or K8s in short) provide us with service and storage orchestration, load balancing, autoscaling, self-healing (failovers) and configuration management.

2.3.2 Kubernetes Architecture

As the dissertation focuses on optimising pod autoscaling techniques for Kubernetes, understanding its components is essential. So, let's understand its architecture in detail.
Now, whenever we deploy K8s, we get a cluster. Inside this cluster, we have a master-slave architecture: control planes (master nodes) and worker nodes that run the containerized applications.

These worker nodes in Kubernetes host something called the Pods. These are the smallest deployable computation units in K8s. A pod is a set of one or multiple containers, sharing resources like storage and network. When we use Docker containerization in K8s clusters, a Pod

represents a group of Docker containers with shared namespaces and shared file system storage. We can allow pods, controllers, volumes, and services to work together while keeping them separated from other components within the cluster. This is done by developing namespaces, which also apply consistent resource configurations. The shared context is defined as a group of isolating components— like Linux namespaces and cgroups.

Pods can be used in two ways in Kubernetes clusters:

- The "one-container-per-Pod'' model, where each pod runs a single container inside. Kiubernetes manages these pods and not the containers directly. This is the most common use case while working with Kubernetes.
- Pods running multiple containers. Sometimes, multiple containers are required to run together shared storage and network access. A Pod can encapsulate these containers thus forming a single unit of service.

These pods are managed by the worker nodes (referred to as “nodes” from here on) and are scheduled to them by the control plane (master node). Each of these nodes could be a virtual machine having its own Linux environment. In a production environment, multiple machines host the control plane and multiple worker nodes are put inside the cluster. This way, the system provides both high availability and fault-tolerance.

For each worker node to perform, there must be three components running on it:[7]

- First is kubelet, which makes sure that the containers described in the PodSpecs are up and running and healthy. Thus it is responsible for taking that configuration and actually running a pod or starting a pod with a container inside and then assigning resources from that node to the container like RAM and storage. We can have hundreds of these worker nodes which will run pods with containerized applications inside them.

- For these applications to communicate with one another and with services outside the cluster, we expose these applications using K8s Services. To implement these services, we use a network proxy called a kube-proxy inside each node. kube-proxy maintains network rules on these nodes, and knows which services are defined in the cluster and maintains the rules for load-balancing. It thus manages the forwarding logic in a performant way with low overhead. For example, if an application is making a request to a database instead of service just randomly forwarding the request to any replica of the database it will actually forward it to the replica that is running on the same node as the

pod that initiated the request thus avoiding the network overhead of sending the request to another machine.

- The third is the container runtime engine (Docker, containerd, etc). As the name suggests, it is mainly responsible for running containers. Kubernetes also supports a lightweight runtime exclusive to K8s— an engine called CRI-O (from Container Runtime Interface).

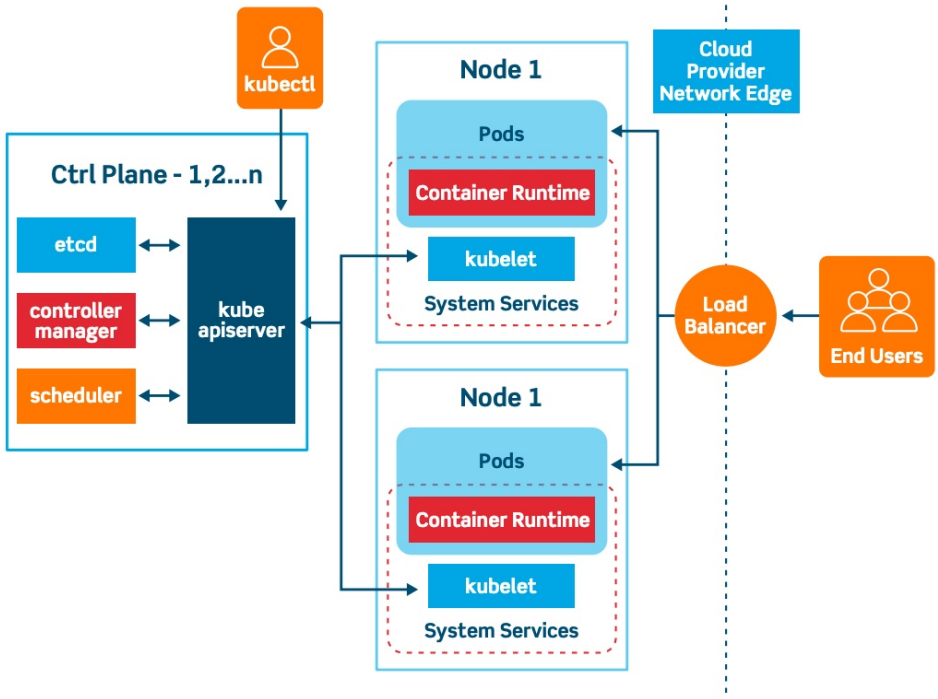

Figure 2.3: Architecture of a Kubernetes cluster

Now the question arises: how do we interact with this cluster? How do we decide on which node a new application pod should be scheduled based on their current resources utilisation? How do we monitor the health of each pod from a top level? If a replica pod dies how can we actually monitor it and then reschedule it or restart it again? The answer is all these management level services are handled in the master node, or the control place in Kubernetes lingo.

The control plane has four components which together monitor and handle these events:

- We need a server through which we can interact with the K8s cluster. This is done by kube-apiserver, which is the K8s API Server. The data for the pods, services, replicationcontrollers, etc. are validated and configured by the server. The REST operations and the cluster's shared state frontend are also handled and provided by the API server. Like whenever we want to schedule new pods, deploy new applications, create new services or any other components we have to talk to this API server on the master node and the API server then validate our request and forward it to the required processes. Other examples include querying the status of our deployment and cluster health etc.

- Next component we need is a scheduler, which is called kube-scheduler. For example, if we want to deploy a new pod, after kube-apiserver validates our request, kube-scheduler assigns that pod to a worker node. Now, instead of just randomly assigning to any node, kube-scheduler takes into consideration factors like: resource requirements, resource constraints, data proximity and deadlines. It also considers the availability of resources in each of the worker nodes so that the whole cluster can run efficiently. The point to note here is that it does not actually start the pod for us within a container (that job is done by kubelet in the worker node), it only decides (schedules) the pod.

- Next component is called kube-control-manager. Now what happens when a pod dies? The control plane should keep track of these by doing health checks and try to recreate the pod (by instructing the scheduler) in order to maintain the desired cluster state. There are different types of controllers inside kube-control-manager to handle different kinds of situations. Node controller: to notice and respond when a node goes down. Job controller: to watch for Job objects, and create Pods to run them. Endpoints controller: to populate the endpoint objects. Service Account & Token controllers: to create default accounts and API access tokens for new namespaces.

- Last but not least is the kubernetes etcd. K8s uses etcd as a consistent and high availability key-value database. It stores the configuration of the Kubernetes cluster and also the actual state and the desired state of the system. All these other components of the control plane like kube scheduler and kube control manager rely heavily on etcd as all the data they use to function are stored inside this database. etcd offers high

durability and high availability, as it is backed up periodically etcd runs as a multi-node cluster.

2.3.3 Scalability and Autoscaling in Kubernetes

In cloud computing, autoscaling is referred to as a technique for dynamically allocating and deallocating computational resources. In Kubernetes, we add these resources to the pods hosting an application when there is an increase/decrease in traffic (workload).
We need to intelligently allocate resources like CPU and memory as we do not want our end users to experience latency. Also, keeping extra pods running will consume unnecessary hardware and software resources, thus increasing the cost of running the application service.
Thus for applications to be highly available, fault tolerant and efficient at the same time, we use autoscaling.

2.3.3.1 Reactive vs Proactive Autoscaler

In reactive approaches, we analyse the current state of the cluster with predefined thresholds and traffic metrics.[8] However, especially when traffic varies quite rapidly, it might result in slower response from the server. This is because these pods require some time to initialise and run properly. Thus, sometimes, it may be less effective to take a reactive approach.
Proactive approaches, on the other hand, analyse the past data and make an informed decision about the number of pods to deploy next. Of course, predictions are not always accurate and it might take a reactive approach if there is an unpredictable surge/fall in traffic. But proactive autoscaling works better in most cases, especially when there is a pattern in the data which the models can predict.

2.3.3.2 Vertical Pod Autoscaling

Vertical scaling means to scale up (or scale down), that is, increase (or decrease) the capacity of the existing machine by adding resources. In contrast, horizontal scaling is about increasing (or decreasing) the number of machines with similar configurations. It is the ability to scale in or scale out to deal with the incoming traffic. With respect to pods in K8s, vertical Pod autoscaling helps us to analyse and set memory resources required by Pods. Every scheduling decision is made based on the resource requests.[9] Additionally, resource limits are just a hard limit for the kubelet to kill your pod if it exceeds these limits. When we define vertical autoscaling, the

scheduler will try to use those values to deploy our pod. Let us consider an example to understand this. Suppose we run a pod for an application with these default settings with 100m CPU and 200Mi memory resources. The VPA engine might determine that we need 150m CPU and 300Mi memory instead for the application pod to run correctly. Accordingly it will keep the same request to limit ratio (cpu:- 1:2, memory:- 4:5) we originally configured, and proportionally set the new values.

```
#Default settings
requests:
 cpu: 100m
 memory: 200Mi
limits:
 cpu: 200m
 memory: 250Mi
```

```
#VPA recommendation
requests:
 cpu: 150m
 memory: 300Mi
limits:
 cpu: 300m
 memory: 375Mi
```

2.3.3.3 Horizontal Pod Autoscaling

For sudden increases in resource usage, we use the Horizontal Pod Autoscaler (HPA) for short. In horizontal autoscaling, the response is to deploy more pods for an increased workload (traffic). Kubernetes implements the HPA in a loop, with a sync period of fifteen seconds. During each period, the controller manager notes the CPU utilisation and other resources.

According to the algorithm mentioned in the official docs,[10] the logical operation of Kubernetes HPA is defined as:

desiredReplicas = ceil[currentReplicas x (currentMetricValue/desiredMetricValue)]

That means, if the desired metric value is 150m and the current metric value is 300m, the ratio will be 300.0 / 150.0 = 2.0. Hence, HPA will increase the number of replica pods to two times of the initial value. Before deciding on the final values of the next round of pod deployments, the control plane also takes in factors like if any metric is missing, how many pods are ready, etc.

Chapter 3

Custom Autoscaler and Architectural Design

3.1 Custom Pod Autoscaler (proposed)

The default Horizontal Pod Autoscaler (discussed in Chapter 2) in K8s is basically a reactive autoscaler. It is a control loop that decides on pod deployment based on resource metrics (CPU utilisation) etc. It does not readily depend on the workload/traffic which is a major factor behind deploying the application pods in the first place. In this dissertation, we suggest a proactive approach to autoscale pods horizontally based on time-series forecasting. This way, especially when we are dealing with traffic that depends on a pattern/trend/seasonality, proactive deployment would ensure high availability and cost optimization.

Many works [11–12] related to autoscaling in cloud computing have been proposed. However, autoscaling solutions related to containerized applications and Kubernetes Pods are still emerging.

We have used statistical models like ARIMA and also machine learning models like LSTM and compared the results. These models can be implemented (using a MAPE-K loop) inside the "custom pod autoscaler" component described in the next section.

3.2 Proposed Architectural Design

This section discusses where and how exactly the optimised horizontal pod autoscaler would fit in. The architecture proposed is based on the MAPE-K loop.[5] The MAPE-K adaptation loop is quite popular with Robotics operations, where it acts like an autonomic manager and receives data from one set of resources (sensors) and gives instructions to another set of resources (receivers).

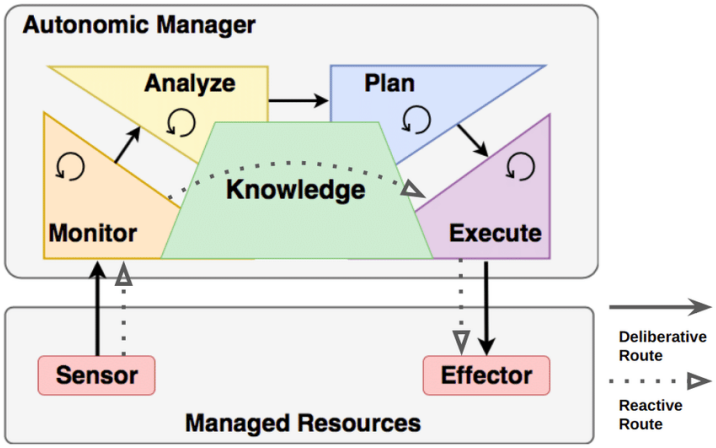

Figure 3.1: MAPE-K Architecture

A similar approach can be taken for designing the Custom Pod Autoscaler. The MAPE-K includes the Monitor, Analyse, Planning, Execution phases and a Knowledge Base:

- In the Monitor phase, we collect the number of HTTP requests made by the users on our server, which is an aggregation of pods with a load balancer handling these requests.

- The Analysis phase has the forecasting models such as ARIMA and LSTM models which we've used in the study. This is located in the Custom Pod Autoscaler component we've introduced in Figure 3.2.

- In the Planning phase, The Custom Pod Autoscaler scales in or scales out the number of pods based on the prediction done in the Analysis phase.

- The Execution phase is handled by Kubernetes kube-apiserver. It receives command from the Custom Pod Autoscaler service and provisions or deprovisions the number of pods accordingly.

- The common knowledge base is at the centre and includes logs, metadata and policies. It gets updated continuously by our Custom Pod Autoscaler and Kubernetes API server. This makes the system self adaptive and ensures higher reliability and availability.

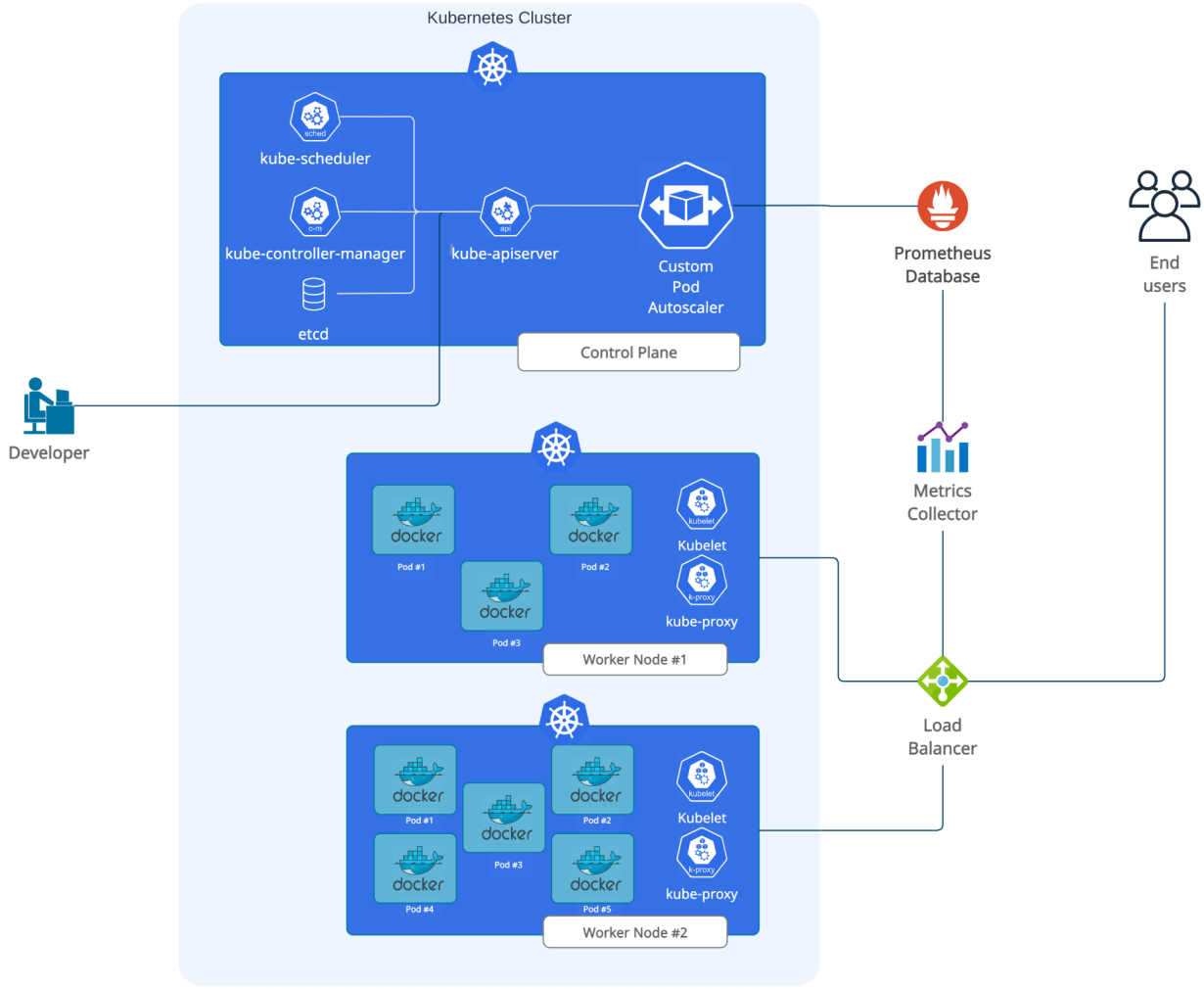

Figure 3.2: Kubernetes Cluster with Custom Pod Autoscaler

We are already familiar with typical Kubernetes architecture (*figure*). Let us discuss some of the newly introduced components/features:

- Custom Pod Autoscaler: Arguably the most important added component which should feature inside the control plane (master node). It is supposed to receive data from the Prometheus time-series database about the HTTP request hits per second received by our load balancer. This request rate is directly proportional to the number of pods we have deployed for our web applications. Apart from Prometheus, Custom Pod Autoscaler is also supposed to receive information from etcd database using kube-apiserver, data

like number of pods running, pods replica sets, etc. The data from Prometheus and etcd are used by the forecasting models to predict the upcoming request rate. The autoscaler then instructs the kube-apiserver to scale in or scale out the number of deployed pods on the basis of predicted workload.

- Load Balancer: The load balancer regulates the incoming HTTP requests and distributes them using techniques like round-robin or using resource-based dynamic algorithms. The requests pass through the load balancer to an available pod running in the cluster, whereas the application metrics (resource utilisation, average response time, application throughput) are sent to the metrics collector.

- Metric collector: The metrics collector is an intermediary component that watches the actions of the load balancer and collects metrics like the incoming request rate and distribution of requests among the worker nodes. Then, it sends this information to Prometheus for storing in the database and making it available to our custom autoscaler.

- Prometheus: The Prometheus component is an application for collecting the data from the metric collector and storing it in the form of a time-series database. This database is maintained so that the custom autoscaler can access this data to make predictions. Prometheus should not be confused with an event logging system. Promotheus exposes this database using an API service which is accessed by our custom autoscaler.

Chapter 4

Implementation and Results

4. 1 Exploring the Data

The dataset has five years of daily traffic (number of http requests per day) on a website called statforecasting.com. The data is quite relevant as the variables have both weekly and yearly seasonality. In concept, the patterns we observe here are comparable to what we'd find in real-time time series datasets generated by resource monitoring systems such as Prometheus.
The dataset has day of week, date, page loads, unique visits, first timers and returning visitors count. A visit is defined as a series of hits on one or more pages on the site over the course of one day. The data was collected through a traffic monitoring service known as StatCounter.[17]

	Row	Day	Day.Of.Week	Date	Page.Loads	Unique.Visits	First.Time.Visits	Returning.Visits
0	1	Sunday	1	9/14/2014	2,146	1,582	1,430	152
1	2	Monday	2	9/15/2014	3,621	2,528	2,297	231
2	3	Tuesday	3	9/16/2014	3,698	2,630	2,352	278
3	4	Wednesday	4	9/17/2014	3,667	2,614	2,327	287
4	5	Thursday	5	9/18/2014	3,316	2,366	2,130	236

Figure 4.1: A peek into the dataset we have considered

Next, we need to decide on the metric (field) to build the predictions models on. Now, in K8s HPA, we generally use resource metrics like CPU utilisation to decide if we need to scale. However, this metric is a misfit for our forecasting algorithms here as CPU utilisation itself is a result of autoscaling. A metric has to satisfy these following criteria to be predictable:

- It is relatively stable and preferably has a recurring pattern.
- It is not dependent on the cluster performance.

Considering these two conditions, Page.Loads (which is defined by the number of http requests made to the website per day) seems to be an ideal metric. Hence, we'll be focusing on these

two following fields: Date (in '%m-%d-%Y' format), and Page.Loads as Count. The models will be based on univariate time series as the Count field only depends on Date field here.
There are 1935 rows of data from September 14, 2014, to December 31, 2019. We can clearly observe yearly seasonality in our case. The trend is somewhat stationary (neither increasing nor decreasing), however, we need tests to verify this. Below (Figure 5.2) is what our data looks like over a span of 5 years.

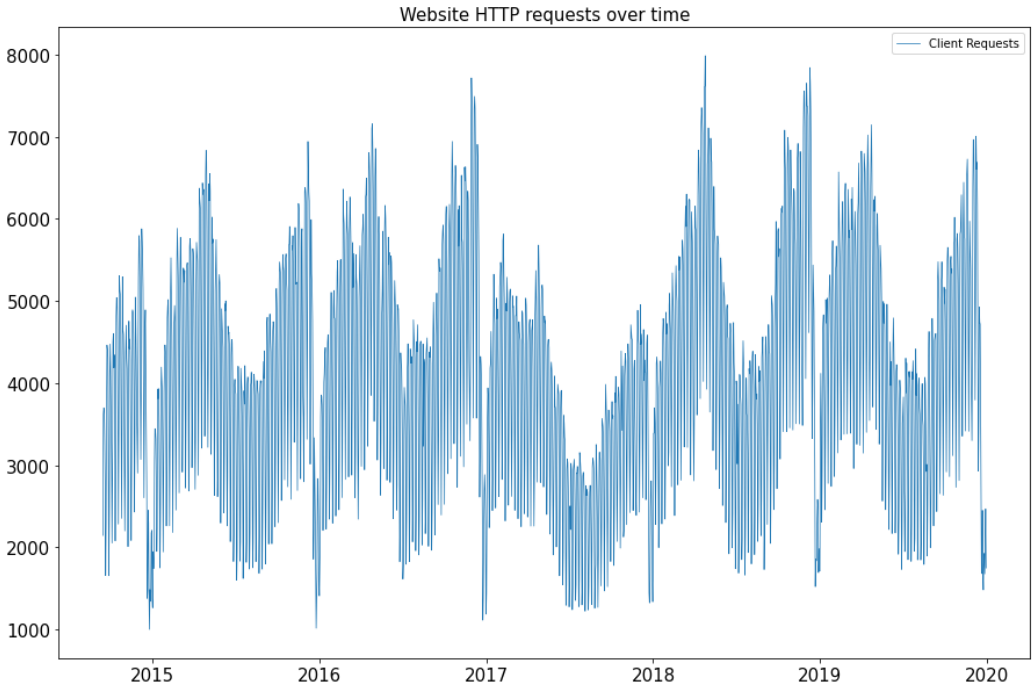

Figure 4.2: Plotting count of daily HTTP requests over time

The noise-like attribute disappears when we take the weekly average of the daily hits on our website (Figure 5.3). But when it comes to deploying pods in Kubernetes, we will however be required to predict this number every few seconds/minutes. Therefore, in this experiment, we shall not take the weekly average but the daily count as it is more practical.

We can decompose a time series into four components:

- Trend: Trend is the tendency of the data to go upward/downward/remain stationary over a long period of time. For example, in our case, the trend seems stationary. This has been further verified by Augmented Dickey-Fuller test.

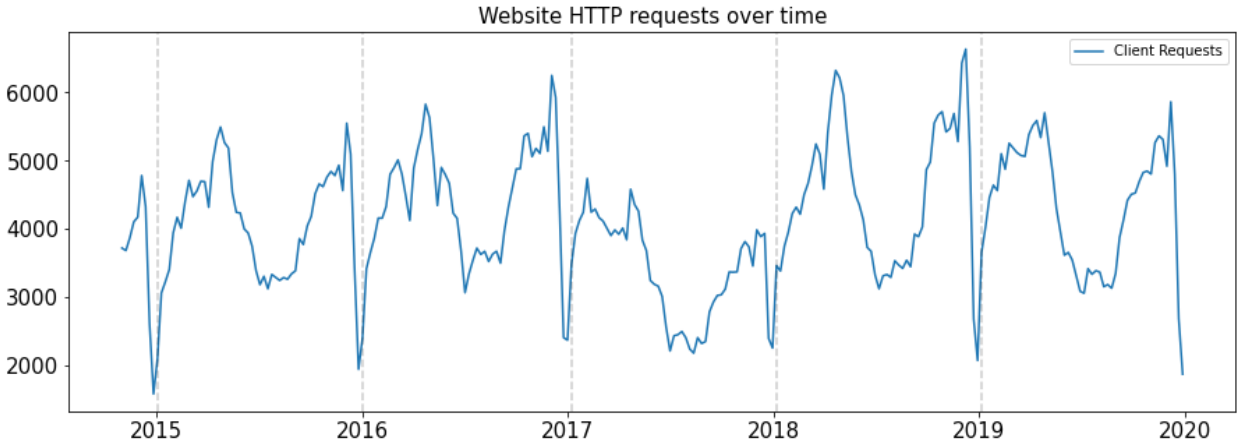

Figure 4.3: Plotting count of weekly average HTTP requests over time.

- Seasonality: Seasonality is the patterns in the observations that repeat over time. From Figure 5.2 and 5.3, our data clearly has yearly seasonality. Website visits are lowest around New Year. As it is an educational website, this makes sense to have lower website visits during that time of year. On closer investigation, we observe that we also have weekly seasonality. Saturdays and Sundays have the lowest website hits, whereas midweek numbers are the highest in contrast (Figure 5.4).

- Cyclic Variations: Cyclic variations can be treated as recurring upward or downward movements in the observed pattern. The period of the cycle is usually greater than a year. They are different from seasonal variations as they are not that regular and can last to longer periods.

- Irregular Variations: Also called residual variations or simply residue, these are fluctuations that are short, erratic in nature and follow no predictable pattern. These are the irregularities in the data after taking the above factors (trend, seasonality, cyclic variations) out.

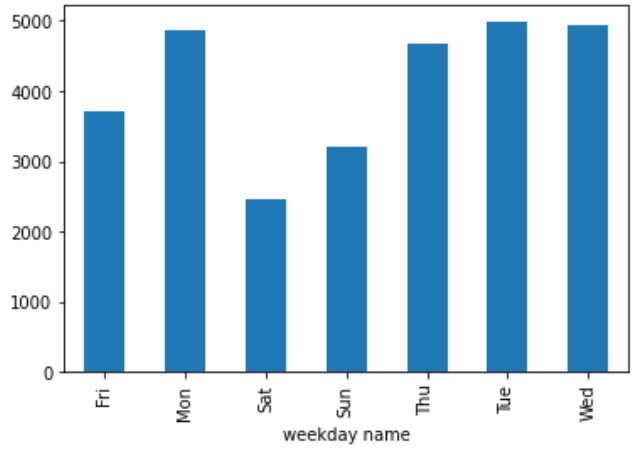

Figure 4.4: Day-wise distribution of data

Augmented Dickey Fuller (ADF) Test

Augmented Dickey Fuller (ADF) Test is a Unit Root Test which we will use to determine if our data is stationary. A time series is stationary if it meets the following criteria:

- It has a constant mean.
- It has a constant standard deviation.
- Constant covariance between periods of identical distance

When a unit root is present, it suggests that our time series data is not stationary. To make the series stationary, the number of unit roots should correspond to the number of differencing operations.

The ADF test takes the Dickey-Fuller equation and includes high order regressive process in the model. In the ADF Test for stationarity, the null hypothesis (H_0) is that the time series has a unit root and therefore is non-stationary.[18] We get two important metrics when we perform the ADF test:

- First is the ADF Statistic— the more negative ADF statistic is, the more likely we are to reject the null hypothesis.

- The second one is the p-value—

- If p-value > 0.05: Our Time Series is non-stationary. We do not reject the null hypothesis (H_0), the data still has a unit root and is non-stationary. If this is the case, we do differencing operations until we get this p-value <= 0.05.
- Else when p-value <= 0.05: Our Time Series is stationary. We reject the null hypothesis (H_0), which means the data does not have a unit root and is stationary.

When we perform ADF test on our dataset, we get the following result:

```
ADF Statistic: -4.071149
p-value: 0.001081
Critical Values:
      1%: -3.434
      5%: -2.863
      10%: -2.568
```

From the p-value in our test, it is clear that our data is already stationary and we do not require any further differencing operation.

AutoCorrelation Function (ACF)

In simple terms, correlation is a statistical measure of the relationship between two variables. Assuming Gaussian distribution of the time-series variables, we can consider the correlation between the current observation with a few observations taken previously, also called lags. Because this is done in the same series of data, it is called autocorrelation or a serial correlation.[19]

Now what is an AutoCorrelation Function (ACF)? It is simply the plot of the autocorrelation of a time series by lag. When we plot the ACF for our dataset, we get the result shown in Figure 4.5. The blue region represents the confidence interval (95% by default). Correlations inside this region are not statistically significant.

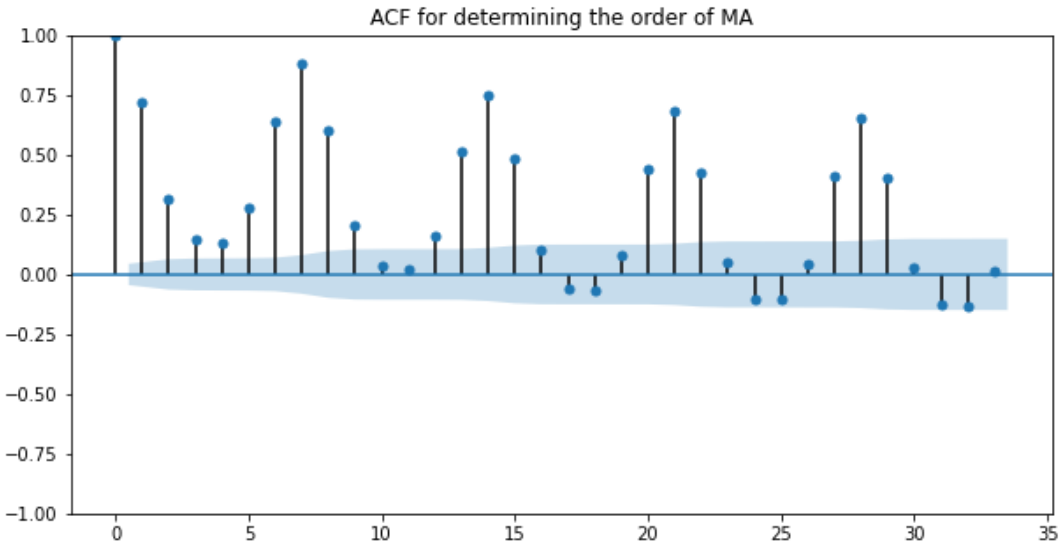

Figure 4.5: ACF plotting for our dataset

Partial Autocorrelation Function (PACF)

In general, the partial autocorrelation is defined as the correlation that results after removing the effect of any correlations due to the terms at shorter lags. In other words, a partial autocorrelation is measured between an observation in a time series with a previous observation by removing the relationships with intervening observations.

As we know, the autocorrelation has both the direct correlation and indirect correlations with prior observations. These indirect correlations can be treated as a linear function of the correlation of the observation with intervening observations. Partial autocorrelation aims at removing these indirect correlations in the autocorrelation, hence it's called 'partial' autocorrelation.

Like ACF, PACF is also the plot of the partial autocorrelation of a time series by lag. We get the following plot in Figure 4.6 when we apply PACF on our dataset.

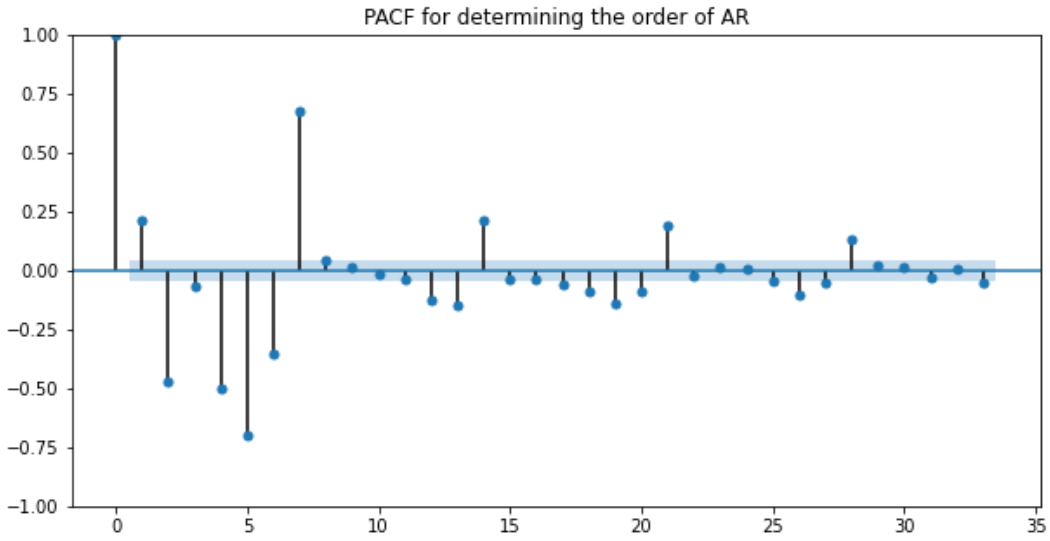

Figure 4.6: PACF plotting for our dataset

4.2 Forecasting Models

4.2.1 ARIMA

The Autoregressive Integrated Moving Average (ARIMA) model helps us to make future predictions by applying statistical analysis and linear regression on time-series data. The acronym can be broken into three parts:

- The first part, AR, means autoregression. It means the model uses regression on itself to establish the relationship between a data and its corresponding past values.

- The second part is short for Integrated, which is a measure of stationarity in the data. As we discussed in the ADF Test, a time-series data is made stationary by differencing operations to remove unit roots.

- The third part stands for moving average (MA), which suggests that the predicted value at a certain timestamp depends linearly on its corresponding past values.

Each of these components in the model accept integer parameters, which altogether represents the type of ARIMA model. These parameters are explained below:[20]

- The parameter p corresponds to the autoregressive (AR) part of the model. It is the number of lags (thus called lag order) on which the current value in the time series depends. The value of p is determined from the PACF.

- The parameter d (degree of differencing) represents the number of unit roots, which are removed by subsequent differencing. It indicates that after performing differencing this many times, our data will be stationary. The value of d is determined by the Augmented Dickey-Fuller Test we've covered earlier.

- The parameter q corresponds to the moving average (MA) part of the model. It is the number of forecast errors its value is determined from the ACF.

We have applied ARIMA with (p,d,q)=(7,0,7). We have determined these values from ADF Test (for d), PACF plot (for p) and ACF (for q) discussed above. We have got an NRMSE score of 0.148.

We have also tried the SARIMAX model, which is an extension of the ARIMA class of models. Applying the SARIMAX model (S for seasonal), we have got NRMSE of 0.168. Though there is a yearly seasonality in the data (Figure 4.3), we have chosen the parameters so as to avoid an overcomplicated model.

4.2.2 Facebook Prophet

Facebook Prophet is another additive regression model for time-series forecasting. It is easier to use as it automatically finds seasonal trends, so it does not require prior knowledge of time series statistical tools. It is capable of predicting time series with multiple seasonalities, which is exactly what we need for our dataset. It is represented as the sum of three time dependent functions with an an error term:

$$y(t) = g(t) + s(t) + h(t) + \epsilon_t$$

where g(t) is growth, s(t) for seasonality, h(t) stands for holidays and an error term e_t.[21] These components are described below:

- The growth term g(t) is the component that accounts for the trend in the data.
- The seasonality term s(t) accounts for the weekly and the yearly seasonality in the data.
- The holidays term h(t) represents the list of holidays or outliers, which should be explicitly mentioned for better prediction.
- The error term e_t is the random noise or the residual part.

With Prophet, our NRMSE value comes around 0.073, which is significantly lower than the NRMSE of ARIMA at 0.148.

4.2.3 LSTM

Next, we have tried a few machine learning models. Recurrent Neural Network (RNN) is often a good place to start with for sequence prediction using machine learning. We have used Long Short-Term Memory (LSTM), which is an improved category of RNNs that uses “gates” for improving the shortcomings faced by standard RNNs. These are the vanishing gradient and exploding gradient problems. Moreover, LSTMs are also better at learning from long sequences of inputs than simple RNNs.

We will now briefly discuss the components of LSTM. An LSTM cell comprises three gates— a forget date, an input gate and an output gate (Figure 4.7). Each of these gates has a sigmoid function layer to determine how much information to pass through, and a pointwise multiplication operator.[22]

- Forget Gate: It determines the quantity of Long Term Memory (LTM) to store and discards the rest from the cell state using a sigmoid function.

- Input Gate: The input gate determines which values in the cell state we should update using a sigmoid layer and adds a new vector of candidate values to the state using a tanh layer.

- Output Gate: Finally, this gate determines the information we will process to our output of the current event using a sigmoid layer and a tanh layer.

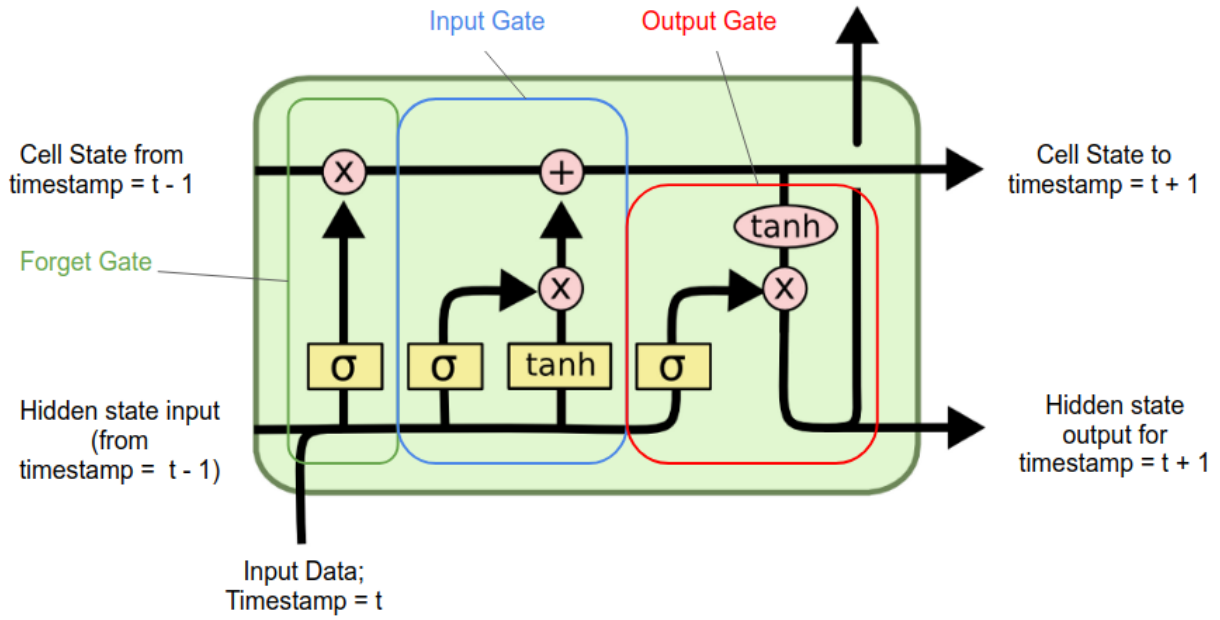

Figure: 4.7: Components of an LSTM cell

For our dataset, we have created the model with a layer of 50 neurons. We have taken the loss function as mean square error and Adam optimizer. Next we have trained our model with batch size 64 with 100 epochs. The NRMSE is 0.061 for this LSTM model, which is slightly better than the Facebook Prophet model.

4.2.4 GRU

Gated Neural Unit (GRU) is a faster and a simpler model than LSTM, which also works using the concept of "gates". In the previous section, we have seen how gates in LSTM are built with sigmoid activation function and pointwise multipliers. GRU gates also have the same components. GRU combines the data from the cell state in LSTM to a single hidden state to transfer information. Let's look at the two gates a GRU cell comprises (Figure 4.8)—

- Reset Gate: As the name suggests, how much past data is to be discarded is determined by this gate.

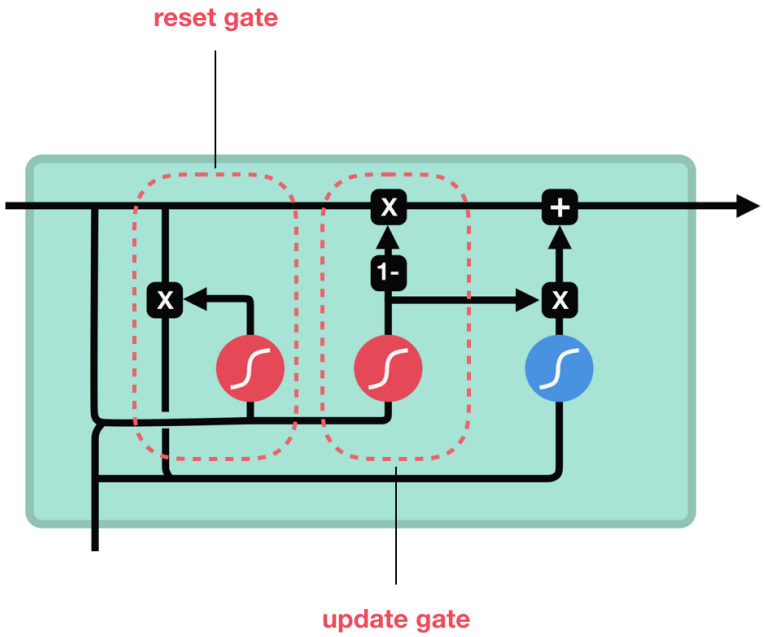

Figure 4.8: Components of a GRU cell

- Update Gate: The update gate of GRU works similar to the combined functionalities of forget gate and input gate in LSTM— it adds new data by updating the values in the hidden state and decides how much data from the previous timestamps should be considered in the next step.

GRU shares many properties of LSTM. As GRU is less complex than LSTM due to a lesser number of gates, it is faster in computational speed.

The model we have created has an input layer and a hidden layer of 64 neurons each. We have used Adam optimizer and mean square error as loss function. We have made the model more robust to changes by introducing a Dropout function which randomly drops 20% units from the network. The NRMSE metric for this model is 0.059, which is an improvement to the LSTM model discussed above.

4.2.5 BiLSTM

A Bidirectional LSTM is an improvement to a typical LSTM in the sense that it is trained by the input data both from left to right (forward layer) and right to left (backward layer). This

increases the context available to the BiLSTM model.[23] In terms of time-series forecasting, the model understands the pattern that precedes a certain timestamp, and the pattern that follows it. Figure 4.8 shows how the two layers combine to calculate the predicted value y_t at a certain time t. This generally results in better forecasting as the long term dependencies are learnt more accurately.

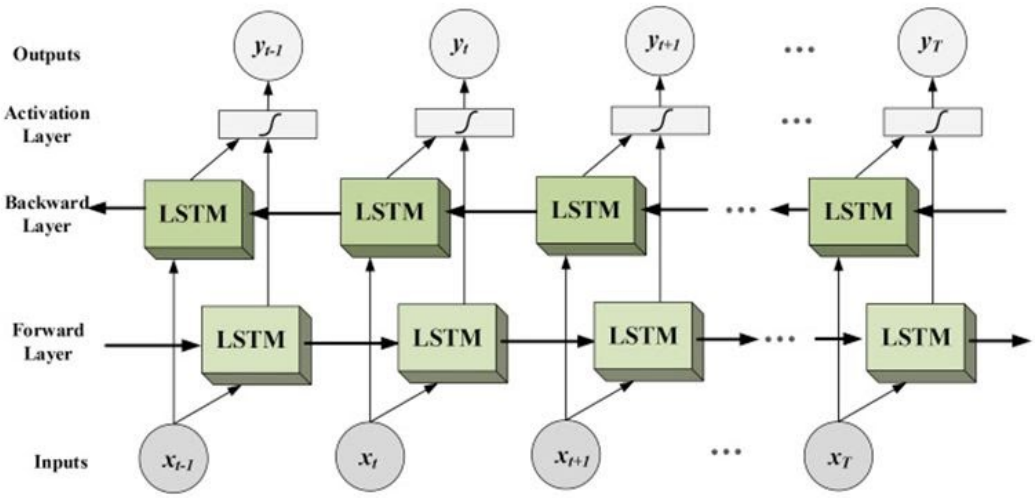

Figure 4.9: Overview of BiLSTM architecture

BiLSTM models often give better results compared to ARIMA and LSTM, which is seconded by our findings. Our BiLSTM model is similar to the GRU model discussed above. We first created an input layer of 64 neurons. Then a hidden layer with another 64 neurons and finally a neuron for the output layer. We have used Adam optimizer and mean square error as loss function. The NRMSE metric is 0.055, which is slightly better than the GRU model.

4.3 Evaluation Metrics

Each of the models we have used tends to minimise the loss value, which is a measure of penalty incurred due to poor prediction. The primary evaluation metric we've chosen here is normalised RMSE (NRMSE), which is derived from root mean square error (RMSE). We have also used mean absolute error (MSE), which subsequently backs up our comparison between the models.

4.3.1 Mean Absolute Error (MAE)

Mean Absolute Error (MAE) is the average of absolute differences between the predicted (forecasted) value of the dependent variable and the actual (recorded) value of the variable. MAE is also treated as an L1 loss function and it is calculated as:

$$\mathrm{MAE} = \frac{\sum_{i=1}^{n} |y_i - x_i|}{n} = \frac{\sum_{i=1}^{n} |e_i|}{n}.$$

where y_i is the prediction and x_i the true value, and $|y_i - x_i| = |e_i|$ is the absolute error.

4.3.2 Root Mean Square Error (RMSE)

Root Mean Square Deviation (RMSD) or Root Mean Square Error (RMSE) is the rooted mean taken over all the differences between the predicted (forecasted) and true (recorded) values of the time dependent variable in the time-series data. We take the root because that gives the error the same unit as the dependent variable itself. RMSE is really a representation of the spread of the error.

RMSD is given by:

$$\mathrm{RMSD} = \sqrt{\frac{\sum_{i=1}^{N} (x_i - \hat{x}_i)^2}{N}}$$

where N is the number of data points, x_i is the true value and $\hat{x_i}$ is the predicted value.

4.3.3 Normalised RMSE

Normalised Root Mean Square Error (or NRMSE) is useful because it is a fractional representation of RMSE over the spread of the data (dependent variable). There are two ways to calculate NRMSE:

- The RMSE is normalised to the range (maximum - minimum) of the dependent variable.
- The RMSE is normalised to the mean of the dependent variable.

Thus the NRMSE can be represented as:

$$NRMSE = \frac{RMSE}{mean(y)} \; OR \; NRMSE = \frac{RMSE}{y_{max} - y_{min}}$$

We have used the first method in our results where y_{max} and y_{min} are the maximum and minimum observed values respectively.

4.4 Results and Analysis

We have used classical forecasting techniques such as ARIMA and also the trending machine learning techniques such as LSTM, GRU and Prophet. To measure the performance of these models, the metrics we have used are mean absolute error (MAE) and root mean square error (RMSE). Normalised RMSE (NRMSE) is also calculated as it represents the RMSE as a fraction of the spread of the data. The results are shown in the table below.

Model Type	**MAE**	**RMSE**	**NRMSE**
ARIMA	813.779	1030.451	0.148
SARIMAX	949.059	1172.071	0.168
LSTM	342.605	422.780	0.061
BiLSTM	287.900	380.991	0.055
GRU	301.383	413.783	0.059
Prophet	360.193	510.246	0.073

Table 5.1: Experiment Results with MAE, RMSE, NRMSE values

We observe that BiLSTM has outperformed all the other models for our dataset (the lower, the better). It has an MAE of 287.900, RMSE of 380.991 and NRMSE of 0.055 (5.5% error), all of which are decisively lower than that of other models we have trained. Furthermore, the time taken in training the BiLSTM model is almost comparable to that of the LSTM model.

We can see that our BiLSTM model outperforms the ARIMA model by 63% reduction in error

rate. It is also superior to Prophet and LSTM models as it reduces the error rate by 25.3% and 9.9% respectively.

Classical models like ARIMA (NRMSE 0.148) and SARIMAX (NRMSE 0.168) have the highest error values. We have tried to avoid training these models with very high parameters as it—

- increases the AIC and BIC scores (the lower the better) by increasing the complexity of these models
- takes significantly more computation time than all the other models

We observe that LSTM, GRU and Facebook Prophet have also done much better than the ARIMA models. Prophet took the least time to build. However, BiLSTM appears to be the most preferable choice from a time versus accuracy standpoint.

Chapter 5

Conclusion

The concept of cloud computing and microservices provides a new method of agile, resilient service which is both autonomous and highly scalable. Autoscaling is one of the essential mechanisms of cloud that features cost optimization and high availability. The dissertation focuses on how we can optimise the horizontal pod autoscaling feature in Kubernetes.

As we have seen in Chapter 2, the default Horizontal Pod Autoscaling (HPA) technique is a reactive autoscaler. That is, HPA dynamically allocates pods based on the current workload. When spiking or dropping below a certain threshold, HPA triggers the addition or removal of pods from the worker nodes. Though these reactive methods can address issues like sudden spikes by scaling out aggressively, it is better if we can scale out proactively, especially when we've found some evidence/pattern in the past data to do so. From a cost perspective point also, running more than necessary servers (or pods) is clearly not the best solution.

That is why we have introduced a custom pod autoscaler in Chapter 3. We have discussed exactly where the autoscaler could fit in inside the Kubernetes cluster and proposed performance optimization using MAPE-K architecture. With this proactive autoscaler, we can address issues like rapid in-demand spikes, outages and variable traffic patterns we often see in a cloud-native application. Moreover, by applying time series forecasting to pod autoscaling, we are also harnessing the past data which is usually discarded by a typical reactive autoscaling method.

For our dataset, we've seen that the BiLSTM model has outperformed all the other models. It is more accurate and a lot faster than the ARIMA model, especially in long term forecasting. Its speed is comparable to the simple LSTM model, when examined with different workloads. However each dataset is different, hence, it is best to try out all our options and then decide which model works best for us.

As we discussed in Chapter 3, the default Horizontal Pod Autoscaler (HPA) is reactive in nature. The aim of the dissertation is not to replace the default HPA but to add to its capabilities. With

the custom HPA, we can determine the pods we need in advance, which makes it a proactive autoscaler.

There will be unexpected surge/outages in traffic where our custom autoscaler will not be able to predict our needs. In those cases, the default HPA should serve as an excellent backup for us, adding/removing pods based on those unanticipated circumstances. On the other hand, proactive autoscaler could determine if there is a predictable surge/fall in workload based on the pattern in the previous data. Thus, these two systems complement each other. Together they could provide a more efficient and robust solution to autoscaling in Kubernetes and Cloud computing as a whole.

Finally, the proactive horizontal autoscaler we discussed is not limited to Kubernetes. It can also be integrated in cloud platforms such as AWS and GCP to ensure cost optimization and quicker response times.

Bibliography

1. IBM Cloud Team, IBM Cloud. “Containers vs. Virtual Machines (VMs): What's the Difference?” IBM, 9 April 2021, https://www.ibm.com/cloud/blog/containers-vs-vms.
2. Clancy, Molly. “Will Containers Replace Virtual Machines?” Backblaze, 14 December 2021, https://www.backblaze.com/blog/will-containers-replace-virtual-machines/.
3. Brown, Glenn. “It Works on My Machine! How Container Technologies Like Docker Can Revolutionize Continuous Integration.” USENIX, https://www.usenix.org/conference/ures14/technical-sessions/presentation/it-works-my-machine-how-container-technologies.
4. Petazzoni, Jérôme. “Anatomy of a Container: Namespaces, cgroups & Some Filesystem Magic.” Slideshare, 19 August 2015, https://fr.slideshare.net/jpetazzo/anatomy-of-a-container-namespaces-cgroups-some-filesystem-magic-linuxcon
5. Avi Networks. “What is Microservices? Microservices Definition and Related FAQs.” Avi Networks, https://avinetworks.com/glossary/microservice/.
6. “Kubernetes Overview.” Kubernetes, 16 June 2021, https://kubernetes.io/docs/concepts/overview/.
7. “Kubernetes Components.” Kubernetes, 30 April 2022, https://kubernetes.io/docs/concepts/overview/components/.
8. “What is Auto Scaling? Definition & FAQs.” Avi Networks, https://avinetworks.com/glossary/auto-scaling/.
9. Megyesi, Daniel. “Vertical Pod Autoscaler deep dive, limitations and real-world examples.” Medium, 21 February 2020, https://medium.com/infrastructure-adventures/vertical-pod-autoscaler-deep-dive-limitations-and-real-world-examples-9195f8422724.
10. “Horizontal Pod Autoscaling.” Kubernetes, 30 April 2022, https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/.
11. H. Zhang, G. Jiang, K. Yoshihira, H. Chen and A. Saxena, "Intelligent Workload Factoring for a Hybrid Cloud Computing Model," 2009 Congress on Services - I, 2009, pp. 701-708, doi: 10.1109/SERVICES-I.2009.26.

12. R. N. Calheiros, E. Masoumi, R. Ranjan and R. Buyya, "Workload Prediction Using ARIMA Model and Its Impact on Cloud Applications' QoS," in IEEE Transactions on Cloud Computing, vol. 3, no. 4, pp. 449-458, 1 Oct.-Dec. 2015, doi: 10.1109/TCC.2014.2350475.
13. Jacobson, Daniel. "Scryer: Netflix's Predictive Auto Scaling Engine | by Netflix Technology Blog." Netflix TechBlog, https://netflixtechblog.com/scryer-netflixs-predictive-auto-scaling-engine-a3f8fc922270.
14. Yuan, Danny. "Scryer: Netflix's Predictive Auto Scaling Engine — Part 2 | by Netflix Technology Blog." Netflix TechBlog, https://netflixtechblog.com/scryer-netflixs-predictive-auto-scaling-engine-part-2-bb9c4f9b9385.
15. Ahmad, Kamal Ramli. "Adaptation of MAPE-K and Fuzzy Q-Learning in SLA management." NASA/ADS, https://ui.adsabs.harvard.edu/abs/2020JPhCS1529b2100K/abstract.
16. Nguyen, Tuan Anh. "A Self-healing Framework for Online Sensor Data." the University of Groningen research portal, https://research.rug.nl/en/publications/a-self-healing-framework-for-online-sensor-data
17. "Daily website visitors (time series regression)." Kaggle, https://www.kaggle.com/datasets/bobnau/daily-website-visitors.
18. Brownlee, Jason. "How to Check if Time Series Data is Stationary with Python." Machine Learning Mastery, 30 December 2016, https://machinelearningmastery.com/time-series-data-stationary-python/.
19. Brownlee, Jason. "A Gentle Introduction to Autocorrelation and Partial Autocorrelation." Machine Learning Mastery, 6 February 2017, https://machinelearningmastery.com/gentle-introduction-autocorrelation-partial-autocorrelation/.
20. "Autoregressive Integrated Moving Average (ARIMA) Definition." Investopedia, https://www.investopedia.com/terms/a/autoregressive-integrated-moving-average-arima.asp.
21. Letham, Ben. "Prophet: forecasting at scale - Meta Research | Meta Research." Meta Research, https://research.facebook.com/blog/2017/02/prophet-forecasting-at-scale/.
22. Graves, Alex. "Understanding LSTM Networks -- colah's blog." Colah's Blog, 27 August 2015, https://colah.github.io/posts/2015-08-Understanding-LSTMs/.
23. S. Siami-Namini, N. Tavakoli and A. S. Namin, "The Performance of LSTM and BiLSTM in Forecasting Time Series," 2019 IEEE International Conference on Big Data (Big Data), 2019, pp. 3285-3292, doi: 10.1109/BigData47090.2019.9005997.

Digital Receipt

This receipt acknowledges that Turnitin received your paper. Below you will find the receipt information regarding your submission.

The first page of your submissions is displayed below.

Submission author:	Sujatro Majumder
Assignment title:	Sujatro_Thesis
Submission title:	Optimization of Horizontal Pod Autoscaling for Containerize...
File name:	Thesis_Sujatro.pdf
File size:	1.88M
Page count:	40
Word count:	8,728
Character count:	46,371
Submission date:	11-May-2022 11:18AM (UTC+0530)
Submission ID:	1833612480

Optimization of Horizontal Pod Autoscaling for Containerized Applications in Kubernetes using Time Series Analysis

DISSERTATION

Submitted in accordance with the requirements of the degree of
Integrated Master of Technology
in
Mathematics and Computing

By
Sujatro Majumder
Admission Number: 17JE003117

Under the supervision of
Dr. Shuvashree Mondal

Department of Mathematics & Computing
INDIAN INSTITUTE OF TECHNOLOGY (INDIAN SCHOOL OF MINES), DHANBAD May 2022

Optimization of Horizontal Pod Autoscaling for Containerized Applications in Kubernetes using Time Series Analysis

ORIGINALITY REPORT

8%	6%	3%	4%
SIMILARITY INDEX	INTERNET SOURCES	PUBLICATIONS	STUDENT PAPERS

PRIMARY SOURCES

1	Submitted to Heriot-Watt University Student Paper	1%
2	machinelearningmastery.com Internet Source	1%
3	medium.com Internet Source	<1%
4	Submitted to University of Sheffield Student Paper	<1%
5	kubernetes.io Internet Source	<1%
6	Nhat-Minh Dang-Quang, Myungsik Yoo. "Deep Learning-Based Autoscaling Using Bidirectional Long Short-Term Memory for Kubernetes", Applied Sciences, 2021 Publication	<1%
7	Submitted to University of Nottingham Student Paper	<1%
8	Submitted to UNIVERSITY OF LUSAKA Student Paper	

		<1 %
9	github.com Internet Source	<1 %
10	hdl.handle.net Internet Source	<1 %
11	mspace.lib.umanitoba.ca Internet Source	<1 %
12	www.ibm.com Internet Source	<1 %
13	Submitted to The University of Manchester Student Paper	<1 %
14	Submitted to University of Queensland Student Paper	<1 %
15	coderzcolumn.com Internet Source	<1 %
16	easychair.org Internet Source	<1 %
17	dspace.nm-aist.ac.tz Internet Source	<1 %
18	businessdocbox.com Internet Source	<1 %
19	trilogi.ac.id Internet Source	<1 %

20	www.geeksforgeeks.org Internet Source	<1 %
21	Submitted to University of Newcastle upon Tyne Student Paper	<1 %
22	lutpub.lut.fi Internet Source	<1 %
23	Bam Bahadur Sinha, R. Dhanalakshmi. "Evolution of recommender paradigm optimization over time", Journal of King Saud University - Computer and Information Sciences, 2022 Publication	<1 %
24	digitalcommons.njit.edu Internet Source	<1 %
25	onlinelibrary.wiley.com Internet Source	<1 %
26	Submitted to Erasmus University of Rotterdam Student Paper	<1 %
27	Submitted to University of South Australia Student Paper	<1 %
28	de.scribd.com Internet Source	<1 %