ACIT5900

MASTER THESIS

in

Applied Computer and Information Technology (ACIT)

May 2023

Cloud-based Services and Operations

Revolutionizing Cloud Computing with Predictive Autoscaling using transformer model: Improving Resource Utilization

Fatema Tuz Sabiha

Department of Computer Science
Faculty of Technology, Art and Design

Table of Contents

List of Figures

Acknowledgement

I express my heartfelt gratitude to my supervisor, Assoc. Prof. Raju Shrestha, for his valuable time, expert guidance, unwavering support, and insightful comments throughout my study. His motivation has been instrumental in my progress and success.

I also would like to extend my thanks to my husband and three children for their constant support and patience during this challenging journey.

Abstract

The adoption of cloud computing by small as well as large organizations has been rapidly increasing now a days. While cloud computing can be cost-effective, it can also become very expensive if proper care is not taken. In order to ensure high availability, cloud providers often tend to overprovision resources, leading to resource wastage and financial losses. Therefore, there is a growing need for efficient resource management in cloud computing. Recognizing the growing interest among researchers in utilizing machine learning models for optimizing resource utilization in cloud computing, this study aims to enhance resource utilization by automating the scaling of a traffic controller in a cloud environment by using a transformer model, which have gained popularity recently. The proposed approach in this research involves training and utilizing a time series forecasting model to implement an autoscaling strategy that can dynamically allocate resources based on actual and predicted future demand in cloud computing. To implement the proposed model, a transformer model was trained using publicly available data offline and used to predict future traffic. The predicted value was then utilized to calculate the target utilization and fed to a Kubernetes-based Event-Driven Autoscaler (KEDA) component for autoscaling an ingress controller integrated with a microservice application running in the cloud. The model was tested in four different scenarios, including without autoscaling, with Horizontal Pod Autoscaling (HPA), with KEDA, and with the implemented transformer model. The experimental results show that the proposed model did not significantly outperform HPA in terms of the performance metrics considered. However, the proposed model exhibited a trend of changing utilization levels while maintaining a stable response time, suggesting a possibility of improving resource utilization with further investigation and fine-tuning.

Chapter 1: Introduction:

Over the last few years, cloud computing has gained widespread popularity due to its cost efficiency, flexibility, and scalability, and has been adopted by small, medium, and large companies. Flexible scalability, known as autoscaling feature in cloud architecture, refers to the process of adding or removing resources automatically based on demand. Autoscaling optimizes resource utilization (Schuler et al., 2021) for the providers and provides reliability to users. In public clouds, this feature like infrastructure maintenance is also maintained by the cloud providers.

An ingress controller is a crucial component in cloud computing that distributes incoming requests to applications running in a cluster. By scaling an ingress controller, centralized traffic management is achieved, resulting in better control and management of incoming requests. Additionally, an ingress controller can manage traffic across the cluster, leading to improved resource utilization compared to independent application scaling. This approach ensures that resources are utilized more efficiently, resulting in better overall system performance. Kubernetes, the widely used open-source container orchestration framework autoscales instances using *Horizontal Pod Autoscaler* (HPA) based on CPU or memory utilisation deciding on predefined thresholds(Schuler et al., 2021). One of the challenges of using this autoscaler is it requires expert knowledge or good understanding of the application to define the threshold, (Phung & Kim, 2022). It is not that the user can only autoscale depending on cpu and memory utilisation, but also can autoscale their services based on custom metrics using an adopter. A recently developed Kubernetes-based Event Driven Autoscaler (KEDA) has gained popularity as an adaptor or autoscaling solution for Kubernetes environments due to its diverse range of scalers and ability to scale down to zero, a feature not offered by HPA. However, although auto-scalers scale instances automatically, there is a delay in the process of adding or creating a new instance, commonly known as a *cold start*. This delay can hamper the availability and performance of the service. On the other hand, to meet the SLA (Service Level Agreement) agreement, cloud providers often overprovision resources, which can result in increased cost as well as under utilised resources.

Keeping these issues under consideration, this paper proposes an autoscaling policy which will scale up or down resources based on predicted future workloads using machine learning techniques. Machine learning, similar to cloud computing, is a formidable technology. Currently, there is a growing interest in employing machine learning techniques in cloud computing to enhance resource utilisation. Numerous studies have been carried out to optimise and improve resource utilisation in the cloud, utilising various machine learning techniques.

The proposed autoscaling policy aims to utilize a transformer model to scale an ingress controller integrated with an application based on predicted incoming request rate. In natural language processing, the transformer model has already shown big success. It's usage is now being investigated in time series forecasting and that is one of the reasons for choosing transformer for this project. By leveraging this model, the system seeks to optimize resource utilization while ensuring high availability and low latency of an application in cloud environment.

1.1 Problem Statement:

Current autoscaling methods in cloud computing often rely on scaling applications based on the current workload, which can negatively impact application performance by causing delays in resource allocation during sudden traffic surges. To mitigate this issue, cloud providers frequently overprovision resources, resulting in resource wastage. Predicting future workload can aid autoscalers in preparing for incoming traffic surges. To address this challenge, this project proposes an autoscaling policy that utilizes workload predictions to improve resource management.

1.2 Research Question:

Can resource utilization be improved by dynamically autoscaling a traffic controller in cloud computing through using the transformer machine learning forecasting model?

1.3 Motivation behind the project:

Recently, there has been growing interest in using machine learning (ML) techniques to optimize cloud computing resource utilization. Many researchers have explored different techniques and achieved significant improvements, mostly focusing on virtual machine (VM)-based monolithic architecture. However, there are fewer studies on microservices and serverless applications, which are more complex than monolithic architectures but widely used nowadays. Among those researches they have not considered much on the fact of over provisioning resources and assuring Quality of service (QoS) at the same time(Wang et al., 2022). To be more specific to my knowledge no studies has been published utilizing machine learning model to scale request traffic controller to ensure high availability, low resource usage and low latency. On the other hand, the transformer model, a Machine learning technique that has gained popularity recently and has demonstrated promising results in various fields but not used much in cloud resource management.

These facts motivated the exploration of the potential of the transformer model, for forecasting future workload where workload varies considerably, in improving resource utilization as well as latency. By using this model, I hope to improve resource utilization through auto-scaling ingress controller based on predicted future workload.

1.4 Structure of the paper:

This thesis' remaining sections are structured as follows:

- Background in Chapter 2 includes a synopsis of relevant technologies and a literature assessment.
- The strategies and tools to be employed are described in Chapter 3 of the methodology. focuses on the project's architecture and design.
- Chapter 4 goes into great detail about the implementation.
- Results are illustrated and analyzed in Chapter 5 along with a succinct explanation.
- Conclusions from Chapter 6 include project constraints and suggestions for future development.

Chapter 2: Background

This chapter provides an introduction to the key concepts related to the project and an overview of related research works. The related concepts are presented to establish a foundation for understanding the project, while the related research works provide a context for the current project.

2.1 Cloud computing

Cloud computing is a modern popular technology that delivers computing resources such as storage, server, software, and databases over the internet. This technology has changed the traditional business model in the technology sector. Companies of all sizes, from small to large, are now adopting this technology instead of investing a significant amount in capital expenses associated with building and maintaining infrastructure. This allows organisations to utilise resources as required and scale them up or down, thereby improving efficiency and reducing costs by paying only for the usage.

2.1.1 Types of cloud

According to (Rashid & Chaturvedi, 2019) there are four main categories that cloud computing can be grouped into.

- Public Cloud: This type of cloud is shared by hundreds of thousands of customers and is easily accessible by the public or organisations. Cloud providers like Amazon, Microsoft, Google are dominating the market at present. They rent infrastructure and services at a given cost.
- Private Cloud: These are accessed only by authorised users within by a specific business or organisation. These can be managed by themselves or by a third party (Eshete, 2020).
- Community Cloud: This type of cloud is shared among organisations with similar interests, such as Salesforce or QTS Datacenters.
- Hybrid Cloud: A hybrid cloud is a combination of public and private cloud where organizations can take some advantages of public clouds while having full control on their sensitive resources.

2.1.2 Cloud Services

Cloud services refer to the computational resources and services offered by cloud providers. The most commonly used types of cloud services include:

- Infrastructure as a Service (IaaS): This service provides virtualized computing resources such as virtual machines, processors, storage, and networking to customers on a pay-as-you-go basis. Instead of investing in their own infrastructure, organisations can rent this service from cloud providers. IaaS is also known as Hardware as a Service.
- Platform as a Service (PaaS): PaaS is used to develop, run, and test applications without having to worry about software, configuration, or hardware. Developers can focus on building their applications while the cloud provider manages the underlying infrastructure.
- Software as a Service (SaaS): This service offers software applications, such as Microsoft Outlook, to users. With one subscription every employee of an organization can use a service. They don't need to install the application locally.
- Function as a Service (FaaS): It is also known as serverless computing. This service make developers work easier by allowing them to only focus on writing code for a function. They don't need to worry about rest of the work like resource allocation or management of servers. Cloud providers automatically provision resources based on requests, and this service supports a variety of programming languages. AWS Lambda, Azure Functions, and Google Cloud Functions are widely used FaaS platforms.
- Data as a Service (DaaS): It is a valuable cloud computing service for organisations that require large-scale data processing. Through a network connection, users can access vast datasets provided by DaaS. In addition, DaaS offers users a suite of data management and analytics tools for more efficient and effective data processing.

2.1.3 Resource management in Cloud

As mentioned previously, access to shared computing resources such as computing power, storage, network bandwidth, software, and services is made possible via the cloud computing platform. These resources are directly related to a system’s performance, functionality and cost. In order to achieve high performance, scalability, and availability, effective management of these resources requires assuring optimal exploitation of existing resources while minimizing costs Due to unpredictable actions of a huge user group as well as complex infrastructure of the cloud make effective resource management challenging. If incoming traffic of a system is predictable the management or resource allocation can become more easier than handling an unplanned spike.

Challenges are different in different cloud service models, IaaS, Paas, Saas, Faas, Daas and so do their corresponding strategies. Various tools and techniques are used for resource management. Among them virtualization, monitoring and automation are the most common. A systematic approach is essential for effective cloud resource allocation techniques. As per (Marinescu, 2022) there exists four basic mechanisms to implement resource management policies in cloud. They are namely control theory, Machine learning, utility based and Market-oriented mechanisms. Control theory employs feedback mechanism which can predict only local behavior whereas machine learning techniques can be applied for coordinating multiple autonomic system managers without the need of a performance model of the system. Market-oriented mechanism also doesn’t require a performance model but Utility-based approach needs performance model along with a coordinator to coordinate cost with end user performance. Use of these mechanisms depend on the structure and need of the cloud environment.

2.1.4 Advantages and Disadvantages of Cloud Computing

Some advantages and disadvantages as per (Apostu et al., 2013) of cloud computing are pointed out below.

- **Advantages:**
    1. Cost efficiency: For start-ups and small companies that intend to use intensive computing techniques, cloud computing can provide a cost-effective solution. By adopting cloud computing, these companies can reduce infrastructure costs, such as setting up computing resources and acquiring licenses. Furthermore, cloud computing offers cheaper maintenance and upgrades since companies do not need to hire experts for these tasks. There are many flexible pricing options available, including

one-time payments and pay-as-you-go models, which make it a very affordable option for any company.

2. Unlimited data storage, Backup and recovery: Cloud computing makes data backup, restore, and storage simpler, more efficient, and more economical. Because all data is stored in the cloud, businesses don't have to worry about physical resources or the risks associated with them, such as damage, theft, or loss. Contrary to typical methods of data storage, cloud service providers handle data backup and recovery, substantially simplifying the process. Businesses have access to a wide range of backup and recovery options, including routine automatic or manual backups, whole system recovery or recovery of specific files and folders. This enables enterprises to select the best backup and recovery solution that meets their demands and ensures the integrity and availability of their data.
3. Quick and easy application deployment: With cloud computing, developers can save time by not having to develop their own infrastructure, environments, or tools. This is because the cloud provides a platform that makes it possible to deploy applications quickly and easily, eliminating the need for labor-intensive setup or configuration.
4. Scalibility: One of the main benefits of cloud computing is scalability. To adapt to shifting business demands, cloud infrastructure can automatically scale up or down. Since no additional hardware or infrastructure is required, businesses can automatically add or remove resources as needed. To accommodate the increased demand, more computing resources, for instance, can be introduced during times of high demand and then removed when the demand declines. This flexibility ensures that businesses are only paying for the resources they actually require while enabling them to react swiftly to changes in demand.
5. Mobility: Businesses have more flexibility and agility because of cloud computing's mobility, which enables them to react quickly to changing customer demands and market situations. Applications and data are stored in the cloud, making them accessible from any location with an internet connection and on any device. Due to this mobility, businesses can give their employees access to the data and applications they require,

regardless of where they are or what device they are using. Employees, for instance, can work remotely whether at home or on the go without being bound to a certain gadget or place. In addition to lowering maintenance expenses for on-premises gear and infrastructure, this can boost productivity and teamwork.

- **Disadvantages:** Despite of many advantages, cloud technology has some disadvantages that a company should consider while taking decision on migrating to cloud or adopting cloud. Some major disadvantages are discussed below.
  1. Data Security: Data security is a critical concern when it comes to cloud computing, as all sensitive data is stored and handled by a third-party cloud provider. This can introduce the risk of data breaches and other security threats. Therefore, it is essential to carefully consider the reliability and security measures before entrusting them with sensitive data. Choosing a reputable and trustworthy provider with robust security protocols can help to mitigate these risks and ensure the safety and confidentiality of sensitive data.
  2. Cost: Although cloud computing is considered cost effective, it can be more expensive sometimes if not managed and handled properly. Pay-as-you-go service can be beneficial for a company but sometimes it can add unexpected expenses if the usage exits expectation and resources are not managed carefully. Therefore, business organizations should have a clear idea of their usage pattern and need to do a correct calculation to avoid unexpected cost and less resource wastage in a long run.

There are two primary aspects of cloud computing. One is Service Level Agreements (SLA) and the other is Quality of Service (QoS). Cost can be affected by these two factors. SL Agreements between the client and provider includes performance indicators like response time, uptime, and availability. On the other hand QoS provides guarantee of a specific level of service to the end-user. It is related to network capacity and latency.

Higher service level agreements (SLAs) and quality of service (QoS) levels in cloud computing typically incur a higher cost due to the need for more resources to meet the agreed-upon service levels. Providers may overprovision resources to ensure the SLAs and QoS are met, resulting in resource wastage and increased cost. This is an area of concern that requires attention to ensure that the cost of cloud computing is optimized without sacrificing service quality.

2.2 Autoscaling

Autoscaling is a major feature in cloud computing which is directly related to cost. Customers can use additional resources by allocating resources automatically when the demand is high and they only pay for their usage. Resources in this way don't remain unutilized when the demand is low. There are various types of auto-scaling available to meet specific needs.

- Reactive auto-scaling: It involves continuous monitoring of relevant metrics such as CPU utilization, memory usage, network traffic, and request latency. This process dynamically adjusts resource utilization through various tools and technologies, such as Kubernetes or other cloud provider services. There are two types of reactive auto-scalers: Horizontal Pod Autoscaling (HPA) and Vertical Pod Autoscaling (VPA).
    I. HPA is commonly used for stateless applications that do not require persistent data storage. It adds or removes instances of an application to adjust to changes in traffic volume. When traffic increases, more instances are added, and when traffic is low, instances are removed.
    II. Vertical Pod Autoscaling adjusts the capacity of instances of an application depending on load. This is typically used for stateful applications that require storage. It adjusts resources such as CPU, RAM, or other resources allocated to the instances.
- Predictive Autoscaling: This type of autoscaling uses mainly machine learning algorithms to predict future loads from historical data or related factors. This type of autoscaling is used usually where the load is highly variable. It helps to predict the load and autoscale based on that.
- Scheduled Autoscaling: Scheduled autoscaling is helpful where there is a clear trend of incoming loads like periods during day time or specific days in a month. The scaling process can be configured to get triggered on the specific time when the demand is high. This helps optimizing cost while serving high demand.

2.3 Machine learning

Machine learning is a branch of Artificial Intelligence that has the ability to learn from data without requiring explicit programming (Bi et al., 2019). Its main objective is to extract insights and patterns from data to make predictions about new data. Machine learning algorithms can be classified into four main categories:

- Supervised learning: In this type of learning, an algorithm is trained on a labeled dataset where each data point has a corresponding label. The algorithm learns the pattern of the labels and predicts the label of new data.
- Unsupervised learning: In unsupervised learning the algorithm learns pattern or data relationship by getting trained on unlabeled data.
- Semi-supervised learning: It involves the use of both labeled and unlabeled data to train a model. This approach is particularly useful when labeled data is limited due to expense or time requirements. By incorporating a large amount of unlabeled data with a limited amount of labeled data, the model can be trained to improve its performance.
- Reinforcement learning: In this learning type, the system learns through interacting with an environment iteratively and receiving positive or negative feedbacks on a given task on some data(Bi et al., 2019). It learns to maximize the positive feedbacks over time.

2.4 Literature Review

Some related researches to this project are discussed below.

An extensive examination of machine learning-based container orchestration methods in cloud computing settings had been presented recently by (Zhong et al., 2022). The authors introduced detailed classifications to categorize existing research based on shared characteristics and performed a comparative analysis of the investigated techniques using the proposed classifications. Additionally, they identified several open research challenges and potential future research directions. The study concluded that container orchestration systems can effectively utilize machine learning algorithms to model behavior and predict multi-dimensional performance metrics, leading to enhanced resource provisioning decisions in response to shifting workloads in complex environments. However, the constantly changing and diverse nature of cloud workloads and environments significantly increases the complexity of orchestration mechanisms.

(Marie-Magdelaine & Ahmed, 2020) proposed a novel autoscaling framework for cloud-native applications that scales dynamically in both horizontal and vertical directions. The framework employs a Long Short-Term Memory (LSTM) forecasting model that relies on observability data to predict application workload. To validate the effectiveness of their approach, the authors developed a proof of concept and tested it in four distinct scenarios with a consistent load application. The results from the experiments demonstrate that the proposed framework was successful in optimizing the Quality of Service (QoS) while improving application performance.

(Goli et al., 2021) introduced an innovative predictive autoscaling method for microservice applications that leverages various machine learning techniques to forecast the behavior of microservices for different workloads and microservice graphs. They have trained two models one for predicting CPU and another for request rate prediction of each microservice, using two datasets per microservice. During the training process, the authors used Linear Regression, Random Forest, and Support Vector Regressor algorithms, and evaluated their performance in terms of mean absolute error (MAE), mean squared error (MSE), root mean squared error (RMSE), and R2 score.Unlike other approaches, this method avoids transferring loads to other services to maintain performance and Quality of Service (QoS). The proposed model is based on the MAPE-K control loop and surpasses the HPA autoscaler in terms of response time and throughput.

In 2021, Khaleq and Ra (Khaleq & Ra, 2021) suggested an intelligent autonomous autoscaling system that comprises two machine learning models. The first model employs a generic autoscaling algorithm to determine the microservice's resource demand, while the second model identifies HPA autoscaling threshold values based on the resource demand and Quality of Service (QoS) using reinforcement learning agents. The findings indicate that using this system results in a 20% improvement in response time compared to the default autoscaling paradigm.

A workload burst aware autoscaling method has been introduced by (Abdullah et al., 2020)where the autoscaler detects bursts in dynamic workloads using workload forecasting and resource prediction. The goal was to minimize response time and avoid service-level objectives (SLO) violation. The authors explored various machine learning techniques to be used as the model learning algorithm and opted for

Decision Tree Regression for the resource prediction model, and regression techniques for workload forecasting. They evaluated their model against reactive and predictive autoscaling methods and observed superior performance.

In a recent study, (Wang et al., 2022) tackled the limitations of rule-based and learning-based schemes for autoscaling in large-scale cloud systems by introducing DeepScaling, a deep neural network-based framework. The proposed method maintains stable CPU utilization while ensuring quality of service and reducing resource over-provisioning. The researchers employed a spatio-temporal graph neural network to accurately forecast the workload of each service by incorporating service call graphs, and a deep neural network to estimate CPU utilization. Furthermore, they utilized a reinforcement learning model to generate optimal autoscaling policies for services with different workloads. The experiments conducted on a production cloud service showed that DeepScaling outperforms the state-of-the-art autoscaling approach by improving CPU utilization by 24.6% per day and saving 14% more resources. A six-month deployment of DeepScaling on 135 production microservices resulted in an average saving of over 30,000 CPU cores per day.

(Phung & Kim, 2022) aimed to optimize resource usage and response time of an application while satisfying Quality of Service (QoS) requirements with minimal cost by utilizing machine learning techniques in serverless computing through the popular serverless workload management tool, Knative. The authors addressed the delayed response in scaling pods on the Knative platform, caused by the lack of knowledge of future workload. To solve this issue, they proposed an autoscaling policy for the Knative platform using the Knative Pod Autoscaler (KPA) that autoscales workload based on the number of pods calculated using a Bi-LSTM machine learning forecasting model. KPA, by default, considers two metrics for autoscaling: concurrency and request rate. The authors designed their forecasting model to predict future request rates, which they evaluated using the Root Mean Square Error (RMSE) and Mean Absolute Error (MAE) metrics after tuning the training parameters to obtain the lowest possible values. They evaluated the model using two different resource-consuming applications, one utilizing CPU and memory exclusively, and the other less. The primary experiment showed better results than the Knative scheme, motivating the authors to carry out their experiment on a larger production environment.

The delay issue of the Knative platform is also addressed by (Zhang et al., 2022) but they have adopted reinforcement learning to develop an adaptive autoscaling framework, able to scale both horizontally and vertically, for serverless services that are more prone to delay. They characterized the services and developed service profiles based on their performance with different resource allocation using the Q-learning algorithm. The autoscaling approach was then designed based on the resource service profile. To evaluate their framework, they deployed three different services: a sensitive image detection service, a face image recognition and processing service, and a natural language processing service. They compared the performance of their system with Knative KPA and Libra in terms of cost and resource utilization for different workload scenarios, such as burst, gentle and decreasing, as well as plunge increasing and stable. The evaluation results showed that their framework outperformed the other two autoscaling tools in terms of cost and resource utilization.

Chapter 3: Methodology

The chapter begins with a brief overview of the technology and tools that have been chosen for the development of the project, followed by a thorough examination of the design and architecture.

2.1 Tools or Technologies To be Used

3.1.1 Docker

Docker is an open-source software framework that is designed to enable developers to build, deploy, and manage applications in containers. Containers are lightweight and portable packages including the source code and its dependencies. These packages can be deployed in any environment which installs all the dependencies so that the application can be run without spending time on finding and installing dependencies. Unlike virtual machines, containers only contain the essential data required for the application, making them compact and resource-efficient (Docker, 2020). Docker's features allow for scalability, portability, easy management, and cost-effectiveness, making it a powerful tool for microservices and a foundation for cloud-native applications. Furthermore, In developing and testing software applications this tool can also be utilized.

3.1.2 Kubernetes

Kubernetes is a popular open-source container orchestration tool. It manages containerized applications across multiple hosts,providing features such as automated deployment, scheduling, monitoring, and scaling. Additionally, Kubernetes includes self-healing, fault-tolerance, and automatic load balancing capabilities. The platform is also highly extensible, enabling developers to customize their applications without modifying the source code. It also allows the users to declare their desired state of an application(Kubernetes, 2019).

The common practice is to create a cluster which can be of one or multiple masters associated with required number of workers. The master manages the cluster while the workers run the applications deployed on that cluster. Master and workers all are nodes (virtual or physical machines) required to run the applications.

A Pod in Kubernetes is a fundamental unit of deployment that comprises one or more containers with the same network namespace and IP address, enabling them to communicate with each other(Kubernetes, 2019). Kubernetes is responsible for creating, monitoring, and managing Pods, and automatically restarting them if they fail. Kubernetes dynamically scales the number of Pods up or down based on demand to maintain application performance.

Ingress is a powerful feature of Kubernetes that allows to manage traffic routing to applications, running inside Kubernetes, in a flexible and scalable way. It is an API object that allows services running inside a cluster to be accessed by external world. Multiple services can be accessed through ingress with a single IP address.

In this project a Kubernetes cluster will be used with one master and two nodes.

Architecture of Kubernetes:

An architectural diagram of a single master with two nodes is shown

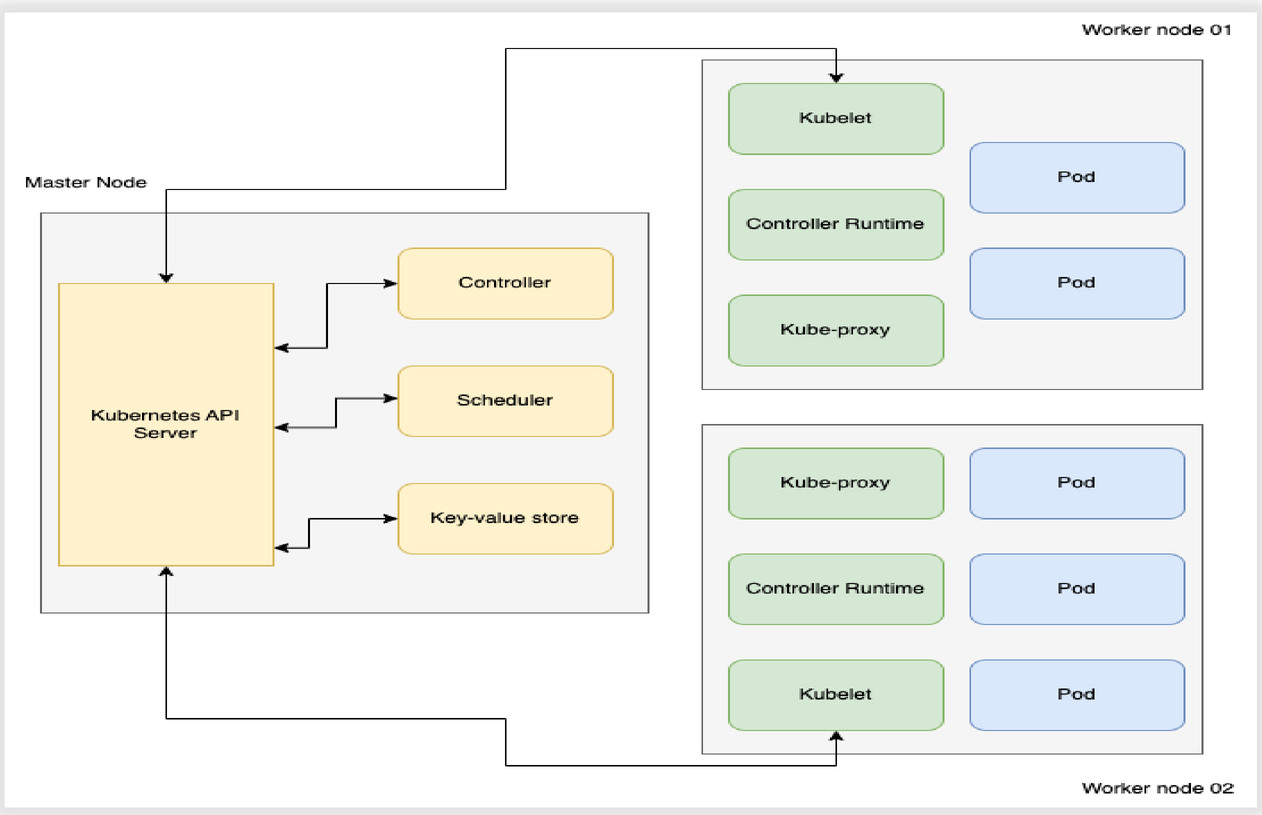

Figure 3. 1 Architectural diagram of a Kubernetes cluster

A master node of a Kubernetes cluster has four major components.

1. Kubernetes API Server: It serves as a central gateway to the cluster, enabling both external users and internal system components to interact with the cluster through the same API. This component handles all REST requests for configurations and modifications, providing a unified interface for managing the cluster.
2. Controller: The controller manages the state of the objects present in the cluster. It retrieves the desired state of objects from the API server and compares it with their current state. If there is a discrepancy between the two, the controller takes the necessary actions to reconcile the state of the object with the desired state, which may include scaling up or down the number of replicas, updating the configuration, or restarting the pods.
3. Scheduler: The API server sends work requests to the scheduler; it accepts and distributes the workload to the worker nodes based on their status. It maintains information about the nodes' quality, ranks them accordingly, and assigns tasks to the most suitable nodes.
4. Key-value store: This component, also known as etcd, stores all the configuration details and the cluster state. It acts like a database.

The worker nodes consist of four major components described below.

1. Kubelet: Kubelet is a crucial program that runs on all nodes in a Kubernetes cluster. Its main function is to enable the worker nodes to be part of the entire cluster, by receiving new assignments from the master's API server, executing them, and reporting back to the master. Additionally, Kubelet monitors the pods running on the nodes it is installed and reports their status to the master.
2. Kube-proxy: It works as a load balancer or network proxy for services running on a node.
3. Container runtime: Container runtime is responsible for managing containers running on the node. It also manages the resources needed for the containers. Docker is nowadays a widely used container runtime tool.
4. Pod: A Pod is the smallest unit that can be deployed in a worker node, and it may consist of one or more containers. While scaling to handle workload, pods are dynamically created or removed.

Autoscaling in Kubernetes:

Within a Kubernetes cluster, there are various approaches to scaling, as discussed in section Autoscaling. However, this project focuses on Horizontal

Pod Autoscaling (HPA), which allows for automatic scaling of workload resources based on demand.

Kubernetes can dynamically adjust the number of replicas for a particular workload based on metrics such as CPU utilization or custom metrics with the use of a HorizontalPodAutoscaler. This means that the number of replicas can increase or decrease based on the current workload demands while providing a more efficient allocation of resources.

How Kubernetes HPA works:

In Kubernetes the configured HPA works as a Kubernetes API resource and controller that enables automatic scaling of replica sets or deployments based on observed CPU utilization, memory utilization, or other custom metrics (Authors, March 30, 2023). The HPA controller runs in a loop, with a default interval of 15 seconds, which can be customized by the administrator.

The HPA is implemented as a Kubernetes API resource. It can be created, updated, and deleted just like other Kubernetes resources.

The configuration file for an HPA is where the intended scaling policy, target resource, and target metric value are provided. The HPA controller can be configured to monitor a variety of metrics, including CPU and memory usage as well as custom metrics. It can be customized to modify the number of replicas in accordance with average usage of all pods related to the target resource.

This makes HPA a flexible and powerful tool for managing workload resources in a Kubernetes cluster.

Algorithm:

The Kubernetes HorizontalPodAutoscaler (HPA) utilizes the following fundamental equation to scale a workload as per the official documentation of kubernetes (Authors, March 30, 2023):

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

As previously stated, the HPA determines the required number of pods or desiredReplicas by using this equation and observed metrics. To further illustrate this, let's consider an example of a deployment with three replicas that needs to be scaled based on CPU utilization.

Suppose the current CPU utilization of the deployment is 60%, and the desired value is 50%. Applying the formula, the calculated number of desired replicas is as follows:

desiredReplicas = ceil[3 * (60 / 50)] desiredReplicas = ceil[3 * 1.2] desiredReplicas = ceil[3.6] desiredReplicas = 4

Thus, the desired number of replicas in this case would be four. This means that the HPA controller would scale the deployment by adding one more pod to the existing three.

It is to be noted that, currentReplicas value is the average metric value of all pods running related to the deployment, not a single pod. Suppose, after the specified interval, the HPA controller found three pods running on the current time with CPU utilization values of 60%, 40%, and 30%, respectively. The controller will consider the average of these values as the CurrentReplicas which is (60 + 40 + 30) / 3 = 43.33%. Since the current value is below the desired or target value of 50%, the HPA controller will calculate the desired number of replicas as follows:

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

desiredReplicas = ceil[3 * (43.33 / 50)]

desiredReplicas = ceil[2.6]

desiredReplicas = 3

So the HPA controller will remove a replica, as the current number of replicas (4) is higher the desired number of replicas (3).

Kubernetes Ingress:

Kubernetes pods are designed to communicate within themselves inside the cluster, but they are not directly accessible from outside the cluster. To enable external traffic to reach services running within the cluster, Kubernetes provides an API object called ingress, which acts as a routing mechanism for incoming traffic. However, the ingress object itself does not handle user traffic; an Ingress controller is required to handle and manage the traffic. An Ingress controller acts as a specialized load balancer that bridges external users and Kubernetes services, simplifying the management of service traffic. The use of an Ingress controller is more cost-effective than using a load balancer provided by cloud providers, and it offers additional features like SSL termination, URL-based routing, and support for multiple protocols. By providing a single entry point for incoming traffic, an ingress controller simplifies traffic management and improves the security of applications running in a Kubernetes cluster.

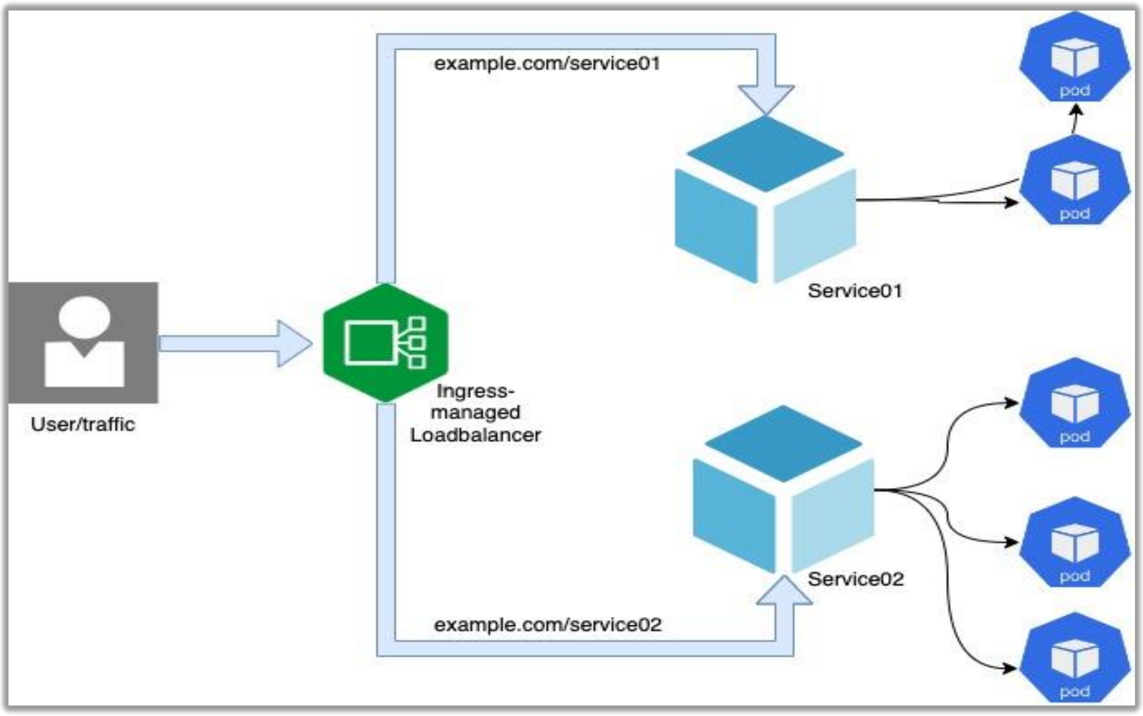

Figure 3. 2 An Ingress controller routing request

In Kubernetes the ingress controller works same as a deployment controller which manages and controls a group of identical pods. The primary task of the controller is to monitor and track the ingress resources and change their states as per user requirement. There can be multiple ingress controller present in a Kubernetes cluster. Any modification made to the ingress resources, such as their creation, modification, or deletion, triggers the controller to update its configuration accordingly. Figure 3. 2 shows how an ingress controller routes traffic to services.

3.1.3 Prometheus

Prometheus is an open-source monitoring and alerting tool used for cloud-native applications. It is very easy to integrate with other cloud-native tools. Prometheus collects metrics from a defined target and alerts the user when a defined condition is met at regular intervals. The collected metrics are stored in a time-series database. Using PromQL query language the database can be queried. Prometheus is a powerful tool that helps to identify and debug issues in the application by providing detailed metrics about its performance. In this project, Prometheus will be used to monitor and collect the required metrics from the metric server of Kubernetes.

3.1.4 Grafana

Grafana is an open-source visualization tool often used in conjunction with another monitoring tool. It has a user-friendly UI to customize dashboards, display real-time visualization of data as well as getting alerts. It will be used with conjugation of Prometheus for visualizing metrics from Prometheus database.

3.1.5 KEDA

KEDA stands for Kubernetes-based Event Driven Autoscaler, is an automated scaling tool that allows the scaling of any container in a Kubernetes cluster based on events that required to be processed. This innovative tool offers a wide range of built-in scalers, making it simple to integrate with other cloud-native applications. Even though KEDA works alongside Kubernetes' Horizontal Pod Autoscaler (HPA) it can scaling to zero which HPA is not capable of. Thus, this tool makes scaling more efficient than ever before. In this project KEDA will be used to integrate Prometheus metrics into the cluster HPA for autoscaling depending on the metrics value.

3.1.6 Transformer Model

The transformer model has become a widely popular neural network architecture and is increasingly being used to replace other models like convolutional and recurrent neural networks (Merrit, 2022). OpenAI utilized Transformers in their language models, achieving state-of-the-art performance on a range of benchmark datasets. Additionally, DeepMind also utilized Transformers in their program AlphaStar, which famously defeated a top professional Starcraft player (Giacaglia, 2019). This success in the gaming industry highlights the versatility of Transformers beyond just natural language processing, and suggests their potential for use in a wide range of applications.

Unlike other models, it uses only self-attention mechanisms to compute representations of input and output sequences, making it highly effective for processing sequential data. Additionally, the transformer model performs parallel processing, which allows it to run faster than other models. One of the main advantages of the transformer model is that it eliminates the need for costly and time-consuming labeled data training by finding mathematical patterns between elements (Merrit, 2022). The transformer model has demonstrated its effectiveness in various natural language processing tasks, including machine translation, text summarization, and question-answering. Recently, its applicability has expanded to other domains, such as Biochemistry in protein folding. In this project, the transformer model will be utilized to investigate its impact on predicting workloads in cloud environment. Specifically, it will be employed as a time-series prediction model to anticipate future workloads, optimizing resource utilization in cloud computing.

Transformer Architecture:

The Transformer was originally designed and developed for translation purposes (Face, 2023). The architecture comprises of an encoder and a decoder as shown in **Error! Reference source not found.Error! Reference source not found.**. The figure is taken from the paper "Attention is all you need". The encoder, positioned on the left side of the architecture, takes input sequences in a specific language. In contrast, the decoder obtains input from the encoder in the targeted or desired language. The encoder is designed to utilize all the words in a sentence as the translation of a word depends on other words of the sentence. On the other hand, the decoder works sequentially and only has access to the words it has already translated. During training, the decoder is provided with the entire target sentence to speed up the process, but it is restricted from using future words to avoid gaining an unfair advantage. Different languages have different grammatical rules in placing the words in orders as well as some words in the sentence can make the context different. Keeping this in mind the decoder's first attention layer is designed to have all the inputs from past but the second layer gets only the input sequence from the encoder. Thus it gains a full idea of the sentence for the prediction.

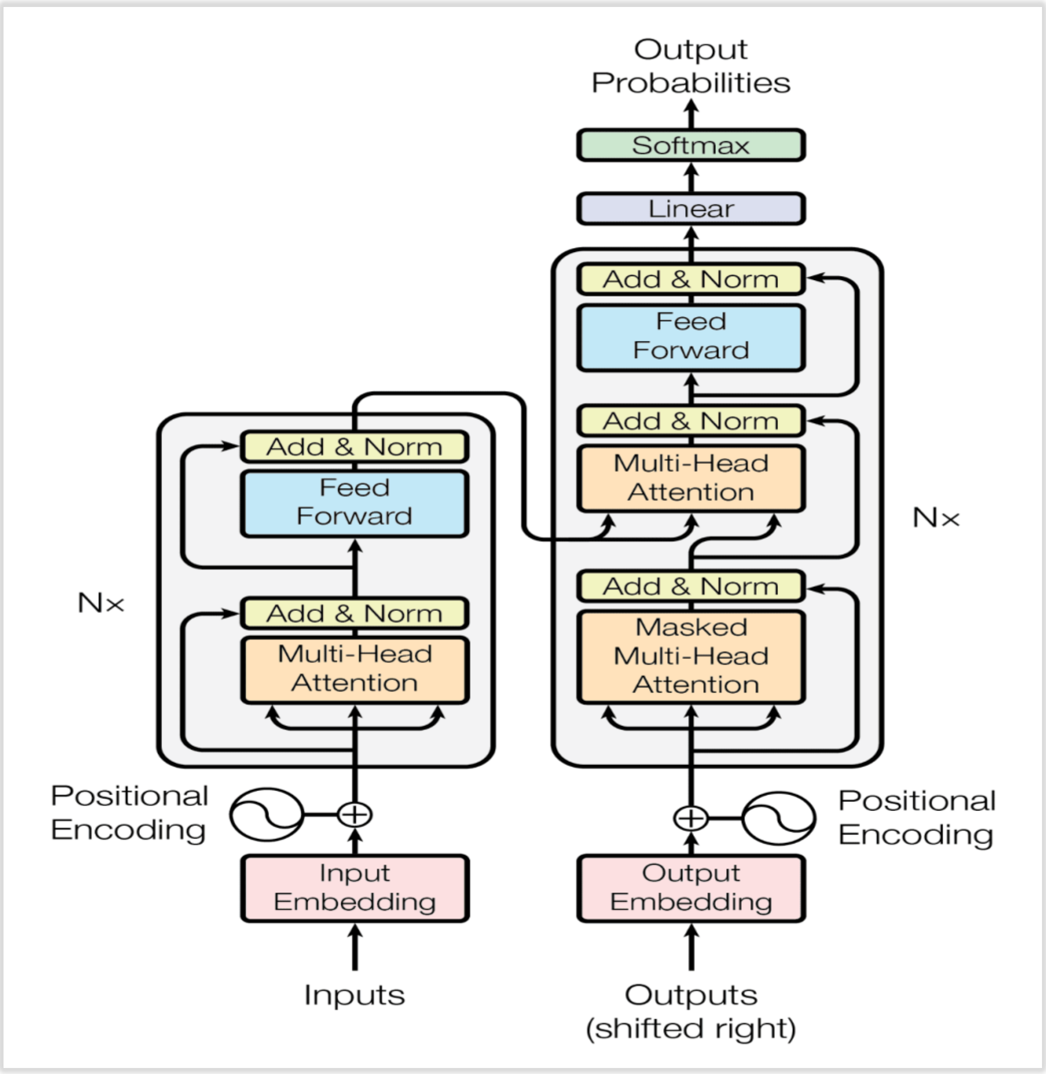

Figure 3. 3 Architecture of transformer model(Vaswani et al., 2017)

3.1.7 Locust

An opensource load testing tool called Locust is used to evaluate the functionality of online applications. By specifying the number of virtual users, the quantity of requests to be made, and the intervals between requests, it is possible for users to imitate real-world user scenarios. It is built in Python. The tool may produce thorough data on request response times, failure rates, and other crucial metrics, as well as providing real-time monitoring of the performance of the application. Any size application, from a tiny website to massive distributed systems, can be loaded tested by it. In this project, Locust will be used to simulate load to test the implemented system's performance and compare it with other state-of-the-art autoscaling policies.

3.2 Design and Architecture

The proposed system architecture is designed to provide efficient scaling capabilities to any cloud native application. The architecture consists of three main components, namely the Application block, the Autoscaling-Controller (AC), and the Autoscaler. The block diagram is shown in Figure 3. 4 and the blocks are explained briefly below.

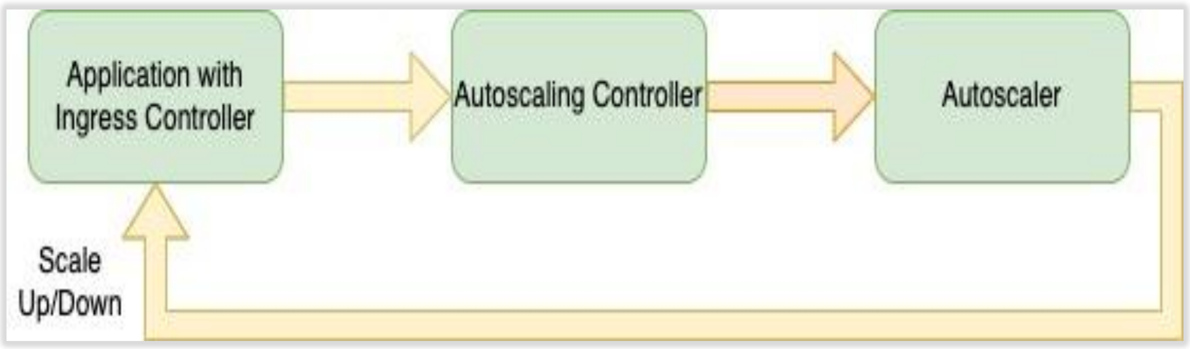

Figure 3. 4 Designed component relationship

- Application: The Application block is responsible for handling external requests to the deployed application. It comprises of an application and an ingress controller, which acts as a load balancer to route incoming requests to the appropriate microservice (Densify).
- Autoscaling-Controller (AC): It is the core component of this architecture, responsible for monitoring the incoming request rates and predicting the future request rate based on historical data. The AC comprises three sub-components, namely the Monitor, Predictor, and Calculator.
  - I. Monitor: The Monitor component is responsible for collecting, monitoring and exposing request rate metric data as time series in a time-series database. It continuously tracks the incoming request rates to the ingress controller and feeds the data to the time-series database.
  - II. Predictor: This sub-component is a time-series forecasting machine learning model that parses data from the Monitor and predicts the future request rates for the next 1 minute. It utilizes advanced forecasting techniques to analyze the historical data and predict the future values. This predictor is an integral part of the autoscaling system as it predicts the future request rate and make the

autoscaler aware of the incoming load to get prepared for scaling which will help to have low latency as well as improved resource utilization while maintaining availability.

III. Calculator: The Calculator calculates the targeted CPU utilization based on the current pods running related to the service and the predicted future request rates. It uses a sophisticated algorithm to balance resource utilization with application performance. This algorithm takes into account the current state of the controller, desired utilization of the resources and the predictions made by the Predictor.

- Autoscaler: The final component of the system architecture is the Autoscaler, which is responsible for implementing the scaling decisions made by the AC. Using an event-driven approach, the Autoscaler automatically scales the ingress controller up or down based on the targeted CPU utilization calculated by the Calculator.

The proposed system architecture aims to provide an efficient and effective approach to scaling any kind of applications as an ingress controller is efficient to be used controlling traffic to microservices as well as serverless applications. By leveraging the Autoscaling-Controller, the system can dynamically adjust to changing traffic loads, ensuring that the application is always available and responsive to incoming requests.

Chapter 4: Implementation

In the previous chapter, the architectural design of the system model was explained, while in this chapter, the step-by-step process to implement the designed architecture is illustrated.

The implemented architecture of the system is shown in Figure 4. 1.

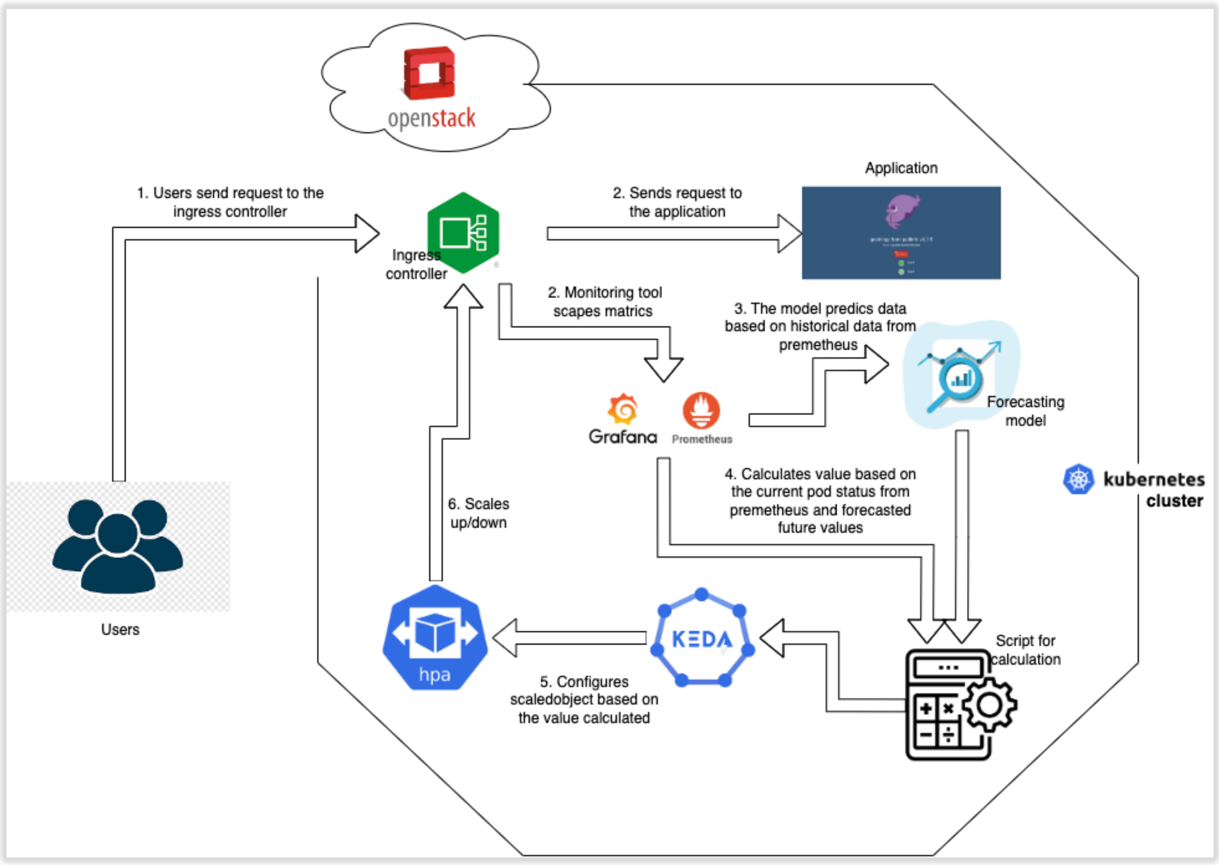
openstack
Application
1. Users send request to the ingress controller
Ingress controller
2. Sends request to the application
2. Monitoring tool scapes matrics
3. The model predics data based on historical data from premetheus
Grafana
Prometheus
Forecasting model
kubernetes cluster
4. Calculates value based on the current pod status from premetheus and forecasted future values
6. Scales up/down
Users
hpa
KEDA
Script for calculation
5. Configures scaledobject based on the value calculated

Figure 4. 1 Architectural diagram of the implemented model

The implementation process can be broken down into the following major stages.

4.1 Training the prediction model

A time series forecasting transformer model was developed using darts library. Dart is a python library which has a vast range of models for time series data processing and forecasting. The prediction model was developed for forecasting incoming future request rates. For training the model trace data from Alibaba cloud was used. The trace data is publicly available and releases real time metric data of microservices collected over 12 hours of period in 2021(Alibaba, 2021).

The detailed workflow of training the transformer forecasting model is described below step by step.

- Data Selection and Downloading: Data for training was downloaded. The trace data 2022 was not available for download at the time the process started. Among four different tables available in the trace data MS_MCR_RT_Table was selected as the prediction model is designed to predict request rates and this table contains the microservice call rate and response information. It has 24 tar files each containing data of 30 minutes. The table has five attributes namely timestamp, msname, msinstanceid, metrics and value.
    1. Timestamp: Timestamp of metrics recorded. For twelve hours range from 0 to (12 * 60 * 60 * 1000).
    2. msname: Name of the microservices the metric is recorded for.
    3. msinstanceid: The container id of the microservice. One microservice can have multiple containers.
    4. metrics: Call rates with different communication paradigms and its corresponding response time.
    5. value: The value of the metrics in number of calls per second.
- Data Preprocessing: For training data need to be processed. The downloaded files were untared and read first and then converted to a dataframe using pandas library. From the dataframe at first the timestamp, msname, metric and value columns were selected. From there the data was sorted for only one specific microservice. Then in the dataframe only the timestamp and value columns were selected. The dataframe then were converted to a time series data.
- Training the model: The preprocessed dataset was split to train and validate the model in 70/30 ratio.

  The parameters that were used to train the TransformerModel class was taken from the official documentation of Darts. The parameters are listed below with the value used.
    - **input_chunk_length:** It defines the number of input time steps. It can be only an integer and was set to 12.
    - **output_chunk_length:** Number of time steps to get from the model. It is also an integer value. In this project the intention was to predict next 1 min of request rates. For that 4 future time steps were taken into account.

- **batch_size:** Number of samples used in one iteration of the training process. A larger batch size can increase the speed of the training but at the same time demands more memory. Here the default batch size 32 was used.
- **n_epochs:** It determines the number of times the algorithm will iterate through the entire dataset. It is set to 200 as per an example showed in Darts official website.
- **Model_name:** The model name is given resource_transformer.
- **nr_epochs_val_period:** This parameter specifies the number of n_epochs after which the validation loss will be evaluated.
- **d_model:** It defines the number of expected features in the encoder/decoder inputs. It was set to 16.
- **nhead:** It determines the frequency of applying the attention mechanism each time focusing on a different aspect of the input. This nhead was set to 8, meaning that the computation involves 8 heads of size d_model/nhead=16/2=2 each. This configuration results in low-dimensional heads that are more suitable for learning from univariate time series.
- **num_encoder_layers:** It specifies the number of layers in the encoder. It was set to 2 as per the example showed in the Darts documentation.
- **num_decoder_layers:** Number of layers in the decoder. It was set to 2 as well.
- **dim_feedforward:** After the attention mechanism the output is fed to a feedforward network. This parameter determines the dimension of the feedforward network. It depends on the complexity of the problem and the amount of training data available. In this project it was set to 128.
- **dropout:** It was set as the default value 0.1 which means in each layer 10% of the neurons will be set to zero randomly while training. This parameter helps to prevent overfitting.
- **force_reset:** This parameter has a default value of “False,” but in this case, it has been set to “True” to force a reset of any previously existing models with the same name.

Evaluating the trained model:

After training the model its performance was evaluated by the validation set in terms of mean absolute error (MAE), Root Mean Squared Error (RMSE), and Mean Absolute Percentage Error (MAPE).

MAE measures the average absolute difference between the actual and the predicted value. It shows how close the predicted value is to the actual on average. Typically, a lower MAE value indicates a better model, but this depends on various factors. The trained model in this project showed a MAE value of 1.40 with the validation dataset which means the predicted value is 1.40 units off from the actual values.

The square root of MSE is the RMSE value. It is also widely used for evaluating a model as it outputs the error in the same unit as the outcome variable which makes it easier to interpret (TrainDataHub, 2022). The RMSE value of 1.81 obtained from the trained model suggests that the average difference between the predicted and actual values is 1.81 units, which is not very high.

MAPE shows the error in percentage making it easy to understand. A MAPE value of 1.49 was found from the trained model which means the predicted value is on average 1.49% off from the actual data.

Even though the evaluation metric were within an acceptable level the performance of the model can be improved by hyperparameter tuning.

4.2 Building the environment

The design of the proposed model adopts a cloud-native approach, taking advantage of the benefits offered by infrastructure-as-a-service (IaaS) resources in the cloud. For this project, the university's private openstack cloud platform, ALTO, was utilized as the cloud platform. To create a production-ready single master, multinode cluster in ALTO, the kubeadm tool was employed. As the proposed model does not account for cluster autoscaling, the lightweight kubeadm tool was deemed suitable for cluster creation. The cluster used in this project consisted of one master node with 16GB RAM, 8 vCPUs, and 160GB disk, and two worker nodes, each with 4GB RAM, 2 vCPUs, and 40GB disk.

4.3 Deploying and integrating different components

In accordance with the designed architecture, a microservice app and a Nginx ingress controller were deployed to serve as the application block. Additionally, Kube-prometheus-stack was deployed for monitoring purposes, while the trained model was transferred to the cloud instance to function as the predictor.

- Sample App: A simple microservice app called Podinfo was deployed in the cluster. Podinfo is a small web app developed in Go, demonstrates the ideal techniques for operating microservices on Kubernetes, and is utilized for end-to-end testing and workshops by CNCF initiatives (stefanprodan, 2021).

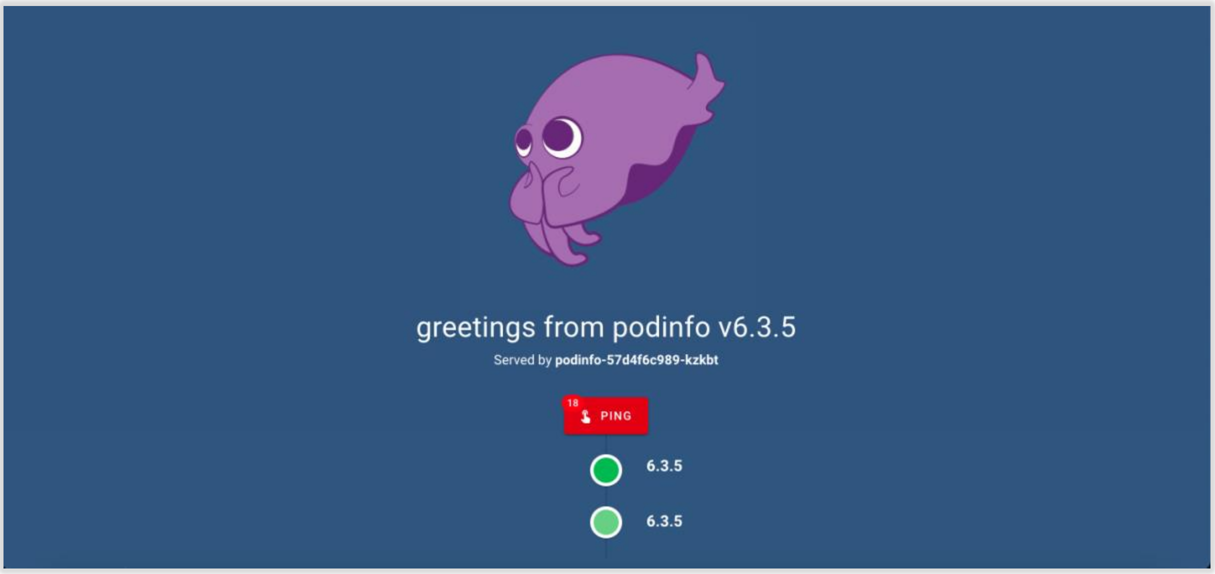
greetings from podinfo v6.3.5
Served by podinfo-57d4f6c989-kzkbt
18
PING
6.3.5
6.3.5

Figure 4. 2 The screenshot of the sample application

For traffic routing to the app nginx ingress controller was also deployed in a created namespace called ingress-nginx in the cluster. An ingress resource was created then to integrate the app with the controller. After integrating the microservice app with the nginx ingress controller using an ingress resource, a load testing was conducted using locust to determine the maximum request rate that the controller can handle while maintaining application performance. The application deployment and ingress resource configuration file are mentioned in the (Appendix A: Configuration files)

- Kube-prometheus-stack: Kube-prometheus-stack is a monitoring solution for Kubernetes and its workloads, which is open-source and maintained by the Prometheus community. This tool was deployed in the cluster, which included Prometheus and Grafana with Prometheus operator and exporter. In the model architecture Prometheus is responsible for gathering metrics from the ingress controller and storing them in its database. On the other hand, Grafana is used for visualizing these metrics.

- Trained forecasting model: The transformer model, which was trained, was retrieved in cluster to forecast the upcoming request rates for the next one minute. A Python script was created in the cluster to load the trained model, extract the request rates data of the past one hour from the Prometheus server, predict the next one minute of incoming request rates and save the predicted values to a predicted_values.csv file. To automate this process, a cron job was configured to run the script every minute.
- Autoscaler: Keda was deployed in the cluster as the Autoscaler. As mentioned before (KEDA) KEDA offers a wide range of scalers. Among them Kubernetes Workload scaler was deemed appropriate for our intended purpose, as it scales applications based on the current status of the running resources. A ScaledObject was created in the same namespace as the ingress controller as its seach scope is limited to the namespace where it is deployed. It was configured in a way using the podSelector attribute to scale up or down the resources related to the controller. The "value" field of the ScaledObject is the target relation between the scaled workload and the current no of pods running for the related deployment (Authors, 2014-2023). This field has been selected in the architectural design of the proposed model to dynamically adjust based on the calculated value, which takes into account the current resource usage and predicted workloads. The content of the scaledobject yaml file is provided in (Appendix A: Configuration files)

4.4 Parameter calculation

To calculate the parameter for the 'value' field another python script was written. The script first loads the YAML file containing the configuration for the Kubernetes ScaledObject. It then reads in a CSV file containing the predicted request rates of next 4 steps of 15s interval (1min). Based on the maximum predicted RPS (requests per second) value and the maximum RPS per pod (determined through load testing), the script calculates the number of pods required to handle the predicted request rate. It then queries the Prometheus monitoring system to get the current number of pods running for the application. Using the number of required pods and the current number of running pods, the script calculates the parameter of the 'value' field and sets the new value in the ScaledObject YAML file. Here the value is multiplied by the desired_utilization to avoid over or under provisioning of resources. Finally, the script applies the updated ScaledObject YAML file to the Kubernetes cluster through the kubectl command line tool, which dynamically scales the nginx controller based on the predicted request rates. This script is meant to be run periodically (every min) through a cron job to ensure the application can handle the predicted workload while avoiding resource waste. Algorithm of the script is written bellow.

Algorithm 1: Calculate the parameter for the ScaledObject value field

```
input: predicted_values.csv= csv file containing the predicted values, Max RPS per pod based on load testing, desired_utilization_percentage
Output: Parameter for scaledObject value field.
1.    Import necessary libraries;
2.    initialize max_rps_per_pod = 70 (based on load testing), desired_utilization_percentage =70;
3.    load scaledObject configuration file;
4.    df = Load the predicted_values.csv file;
5.    max_rps = max RPS value of the df;
6.    scaled_workload_pods = max_rps / max_rps_per_pod;
7.    num_matching_pods= query request to Prometheus;
8.    value = (num_matching_pods / scaled_workload_pods) * desired_utilization_percentage / 100;
9.    write (value) to scaledObject file;
10.   configure the scaledObject using kubectl apply
```

4.5 Visualization and Data Collection:

Grafana was deployed in a Kubernetes cluster to visualize metrics such as call rate, number of pods related to the Nginx ingress, response rate, CPU utilization, and memory utilization. A dashboard was created to display these metrics. The data was downloaded from Grafana in the form of a CSV file.

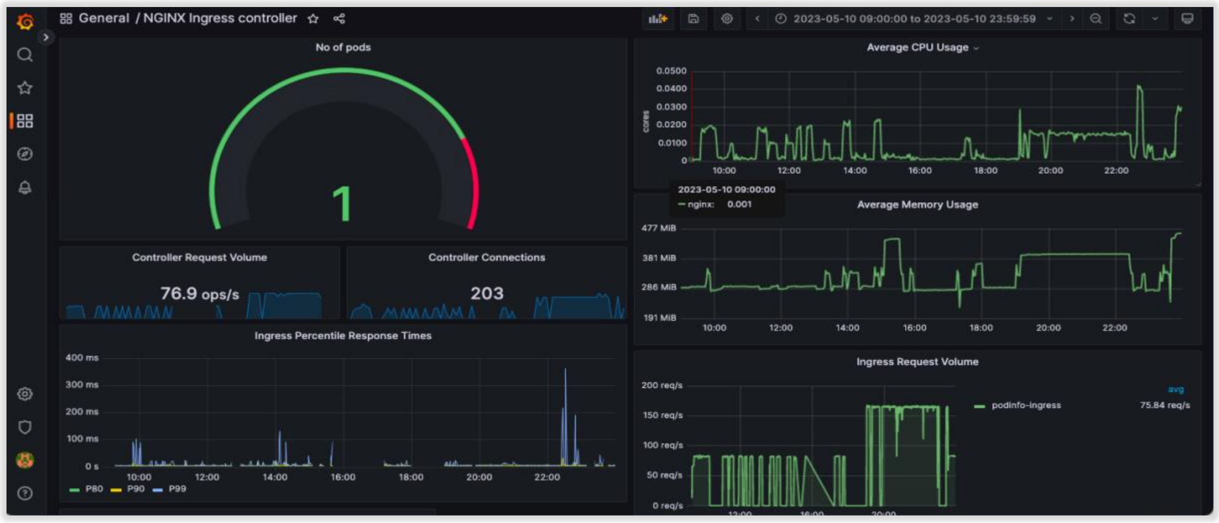

Figure 4. 3 Screenshot of the Grafana dashboard

Chapter 5: Results and Discussion

In this chapter there is a brief discussion on the testing experiments and its results, providing valuable insights into the system's performance and identifying potential areas for improvement.

5.1 Testing in different configurations

Once the system was implemented according to the designed model, load testing was conducted to observe its behavior. The purpose of load testing was to compare the outcomes imposing a sudden increase in traffic under different configurations of the autoscaling system and evaluate the performance of the system.

The experiment involved replicating a sudden traffic surge of 500 users with a spawn rate of 20 to the application, all within the same cloud environment, utilizing four different configurations of the autoscaling system (Marie-Magdelaine & Ahmed, 2020). A spawn rate of 20 means that every second, 20 users were added until the total number of users reached 1000.

The metrics of CPU utilization, memory utilization, and response time were considered for evaluating the implemented system.

- Configuration 1: Without any autoscaling policy.
- Configuration 2: Autoscaling was done using HPA. A yaml file was written and applied to configure an HPA named hpanginx for autoscaling the nginx ingress controller using targeted CPU utilization percentage 70 in the Kubernetes cluster. The HPA configuration file content is provided in (Appendix A: Configuration files)
- Configuration 3: KEDA was configured for autoscaling. KEDA's Kubernetes workload scaler was configured by applying a scaledobject in the same namespace the nginx ingress controller was deployed. The value field of the scaledobject was filled with '0.877'. It was calculated with the formula mentioned in Parameter calculation)

  Value = no of matched_pods / scaled workload pod.

  Value = 1/ (80/70) = 0.877, scaled workload pod was calculated by dividing approximate request rate / approximate request rate which can be handled per pod measured from initial load testing.

- Configuration 4: KEDA was integrated with the other components of the designed model in a way that the target CPU utilization percentage changes with demand to scale up or down resources.

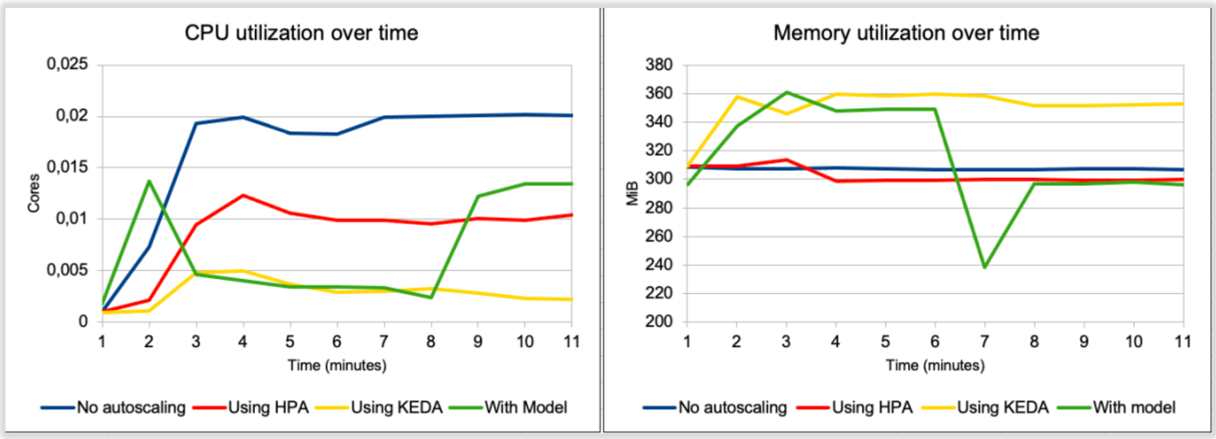

Figure 5. 1 Resource Utilization over time for different configurations (low request rate)

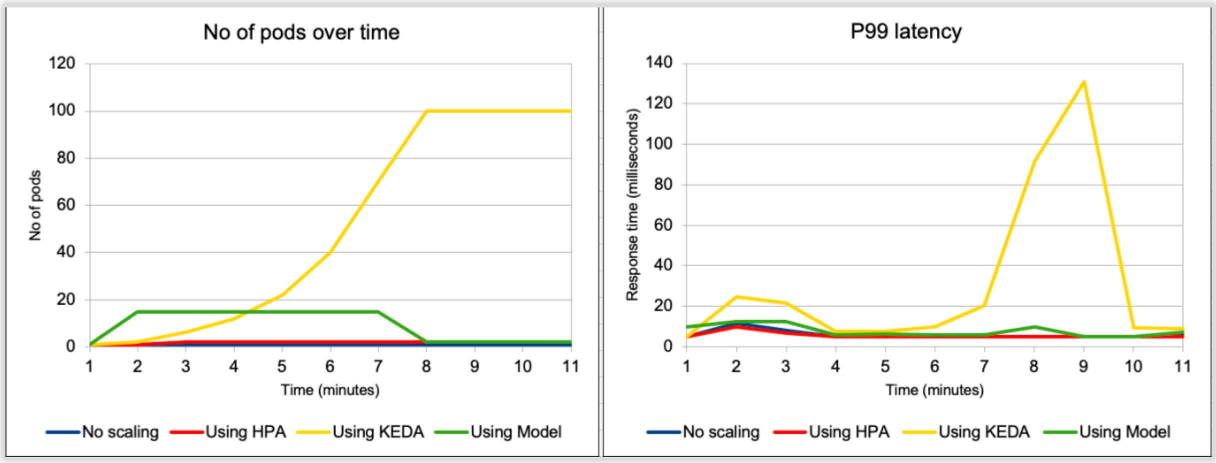

Figure 5. 2 No of pods and P99 latency over time for different configurations (low request rate)

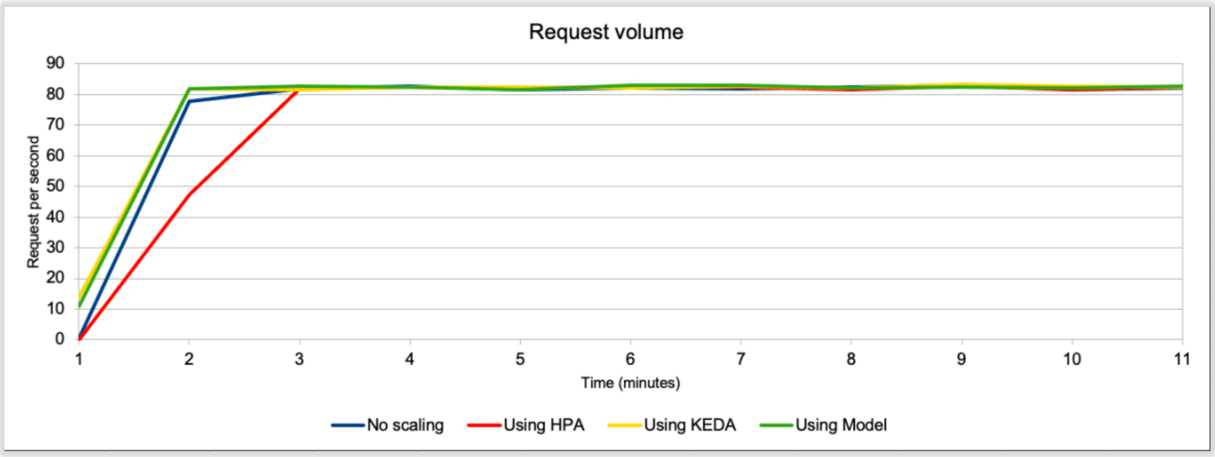

Figure 5. 3 Request volume over time (low request rate)

The graph of Figure 5. 1depicts the CPU and memory utilization trends over ten minutes period of time under the mentioned four different configurations. The number of existing pods at different time stamps along with P99 response time, which indicates the time taken for 99% of requests to be completed successfully are shown in Figure 5. 2.

Regarding CPU utilization, the initial three setups (without autoscaling, with HPA, and with KEDA) exhibited similar patterns of increasing CPU usage with rising load, then stabilizing as the load steadies. However, they consumed less CPU than each other. When no autoscaling was configured, the system operated with only one deployed pod, which had the highest CPU usage among the four configurations but remained steady and consumed less memory. Since it was stable with one pod running and the request rate was within its capacity, the P99 latency also remained stable.

With configured Horizontal Pod Autoscaler with target utilization percentage 70, the ingress controller was scaled up with an additional pod after 2 minutes when the request rate sharply increased, as seen in Figure 5. 3 .With the scaling up process, the CPU utilization increased and stabilized after a certain period of time. Similarly, the memory utilization also became stable after a slight increase and decrease within the same time period. However, in terms of response time, it showed the same trend as the first setup without autoscaling, where the response time stabilized after the first few minutes.

While testing the system with KEDA integrated for autoscaling the graph shows this setup had the lowest CPU usage and the highest memory usage among the four setups during testing. This can be attributed to the fact that the number of pods increased gradually to maintain the correlation between the number of running pods and the scaled workload, and eventually reached the maximum number of pods set at 100. As more pods were allocated over time, memory utilization increased. Consequently, the response time was also impacted, with the graph showing a high response time that decreased approximately after 8 minutes when the number of pods had reached the maximum level.

The proposed model implemented in the system showed a different trend compared to the other three configurations. The autoscaler scaled up to 15 pods from 1 within a minute after the implemented calculator calculated the value field and configured the scaled object. In the first minute, the system showed a rising trend in CPU utilization possibly due to the transition period of no of pods increasing rapidly from 1 to 15 within a short period. This sharp rise was not seen in the other configurations specially to mention the 3rd one autoscaling with KEDA as the target relation 'value' was fixed and the pod was gradually increasing. The CPU utilization decreased after the second minute when the no of pods became stable. The number of pods again decreased to 2 after 7 minutes based on the target relation calculation, resulting in an increase in CPU utilization. The memory usage was high when there were 15 pods running and decreased after 7 minutes when the number of pods went down to 2. The response time showed a slight up and down trend as the scaled object was getting configured every minute, but it remained within an acceptable range of 16ms.

To conduct further investigation, a second round of experiment was carried out with a larger user group consisting of 1000 users with a spawn rate of 20. This time the experiment only focused on configuration with HPA and the proposed model. The simulated user request is shown in Figure 5. 4.

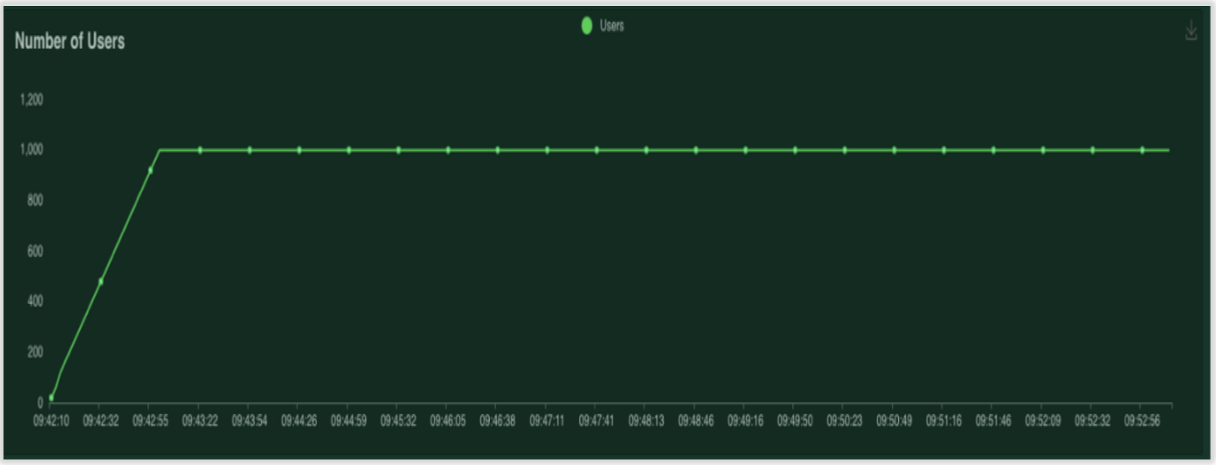

Figure 5. 4 No of users simulated by Locust

Prior to running this experiment, the Kubernetes cluster was resized, resulting in an infrastructure that was able to handle more load. As a result, the request rate increased to above 300 requests per second (Figure 5. **7**) and the maximum request rate handling capacity increased to 330 without any failure. The maximum capacity was reflected in the calculation sheet for accurate analysis.

The results reflected in Figure 5. 5 showed a similar trend in terms of all metrics. The proposed model had less CPU and memory usage than the HPA initially, but after some time, it increased. However, the memory usage pattern was different, with the usage decreasing initially and then increasing, in contrast to the first experiment. The number of pods did not exhibit a significant increase within a short period exhibited in Figure 5. 6, as was observed in the first experiment where it jumped to 15 replicas within a minute. This could be attributed to the maximum capacity per pod being set lower (70) in the first experiment. In terms of latency like last time it was also within an acceptable range.

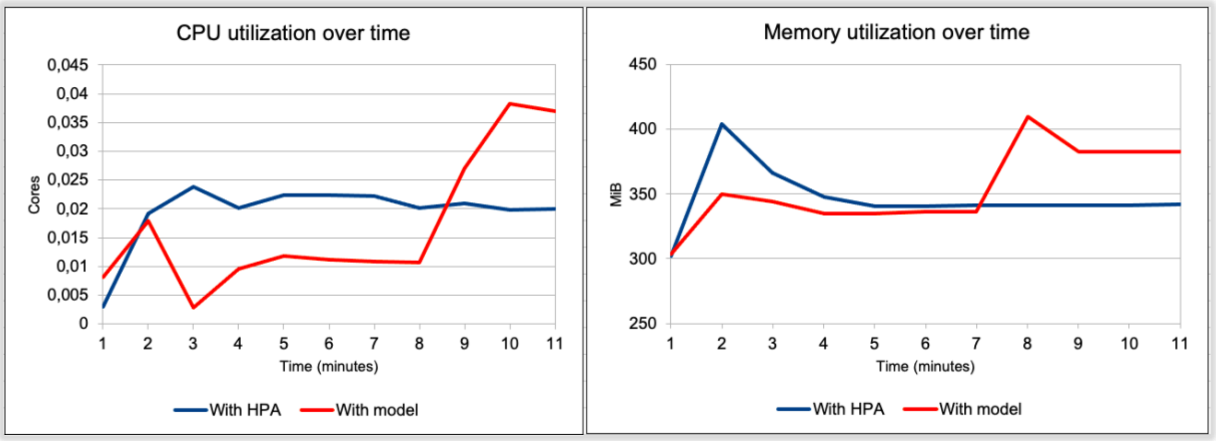

Figure 5. 5 Resource utilization over time of configuration 2 and 4

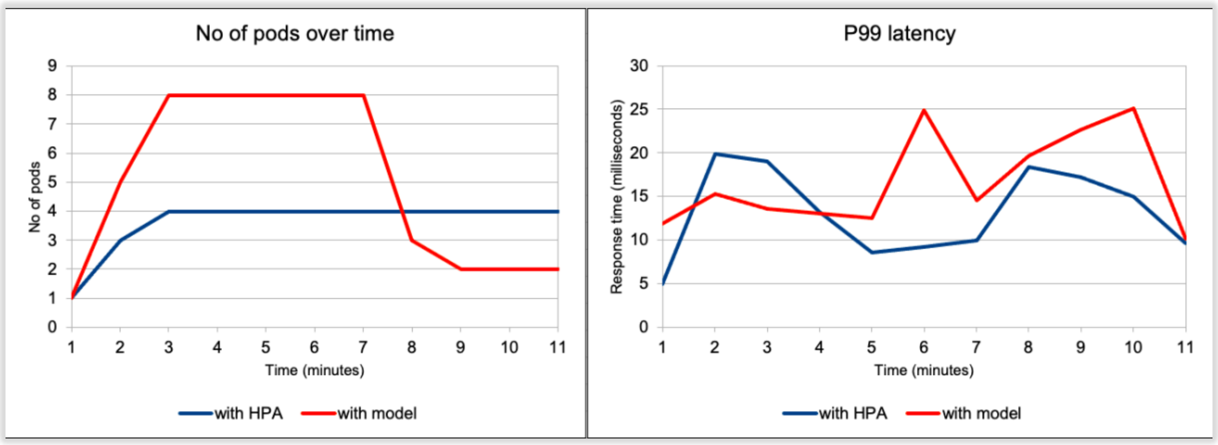

Figure 5. 6 No of pods and latency over time of configuration 2 and 4

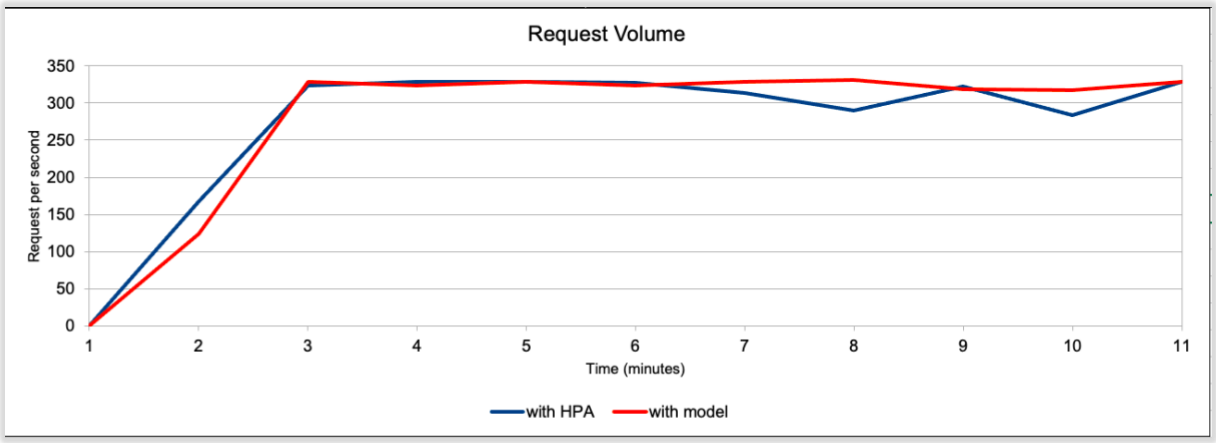

Figure 5. 7 Request volume over time (high request rate)

Based on the results found it is clear that autoscaling has a great impact on system performance. It helps to maintain a stable response time while effectively utilizing resources. Among the three autoscaling configured test scenarios (HPA, KEDA and proposed model) HPA worked the best in terms of all the metrics considered for evaluation even though KEDA showed less CPU utilization than HPA but had unstable response time which is very important to consider. On the other hand, the implemented proposed model showed promise, displaying a trend of changing utilization levels while maintaining a stable response time. Although it did not surpass HPA in any of the performance metrics, it exhibited the potential to improve resource utilization on average over an extended period of operation. However, further experimentation and refinement are necessary to optimize the proposed model and make a more comprehensive comparison with other autoscaling mechanisms.

Chapter 6: Conclusion

The recent Covid-19 pandemic has accelerated the adoption of cloud computing, as it plays a crucial role in digital transformation. The quarterly earnings report of technology company Microsoft demonstrated the significant digital transformation that occurred in a short period of time just a few months after the pandemic started(Microsoft, 2020). In this context, resource management in cloud computing has become an increasingly important research area, as it directly affects the cost, efficiency, and performance of cloud services. Machine learning models have been proposed and implemented to improve resource utilization in cloud computing.to address this issue. The goal of this project was to add to this field of study by designing and implementing an autoscaling strategy that makes use of a forecasting time series transformer model. The findings from the project shows little improvement in CPU consumption which is an important metric for cloud service providers as well as users in terms of cost but the metrics of memory usage was slightly higher compared to other configurations. These findings show that additional study in this field is worthwhile and point to the potential of machine learning models specially a transformer model for resource management in cloud computing. To address the research question, it can be concluded that the proposed model has demonstrated the potential to enhance resource utilization, but additional observation and experimentation are required to optimize the system. This study has provided a foundation or starting point for further research and development in the field of resource utilization and management in cloud computing utilizing an ingress controller and a transformer machine learning model.

6.1 Limitations and Future Works

Future works can address several limitations identified in this study.

The current study utilized a prediction model with default parameters, and the potential for further improvement is evident through hyperparameter tuning. Future studies could focus on optimizing the model parameters to achieve more accurate predictions and enhance system performance. It is worth noting that the absence of workload trends in the dataset could be seen as a positive aspect for the model, as it was still able to make accurate predictions without relying on trend information.

Another drawback of the study is that the proposed model was only evaluated on a simple microservice app. While this was adequate to show the model's efficiency, further research might examine how it can be used to more complex applications, including multi-tier applications, in a bigger production environment. This might make it easier to spot any model flaws and gauge how it behaves in more complicated settings. Further improvements to the existing strategy might also come from investigating the incorporation of different machine learning models and creating a hybrid approach for scaling policy.

References:

Abdullah, M., Iqbal, W., Berral, J. L., Polo, J., & Carrera, D. (2020). Burst-aware predictive autoscaling for containerized microservices. *IEEE Transactions on Services Computing*, *15*(3), 1448-1460.

Alibaba. (2021). *Overview of Microservices Traces*. Alibaba. Retrieved March 02 2023 from https://github.com/alibaba/clusterdata/tree/master/cluster-trace-microservices-v2021

Apostu, A., Puican, F., Ularu, G., Suciu, G., & Todoran, G. (2013). Study on advantages and disadvantages of Cloud Computing–the advantages of Telemetry Applications in the Cloud. *Recent advances in applied computer science and digital services*, *2103*.

Authors, K. (2014-2023). *Kubernetes Event-driven Autoscaling*

. Retrieved 27 March from https://keda.sh/

Authors, K. (March 30, 2023). *Horizontal Pod Autoscaling*. Retrieved May 5 from https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/

Bi, Q., Goodman, K. E., Kaminsky, J., & Lessler, J. (2019). What is machine learning? A primer for the epidemiologist. *American journal of epidemiology*, *188*(12), 2222-2239.

Densify. *Kubernetes Service Load Balancer*

. Densify. Retrieved 30 April 2023 from https://www.densify.com/kubernetes-autoscaling/kubernetes-service-load-balancer/

Docker, I. (2020). Docker. *lınea].[Junio de 2017]. Disponible en: https://www. docker. com/what-docker*.

Eshete, G. A. (2020). *Autonomous Global Distribution of Container Workload using K-means Clustering Algorithm*

Face, H. (2023). *How do Transformers work?* Hugging Face. https://huggingface.co/learn/nlp-course/chapter1/4?fw=pt

Giacaglia, G. (2019). *How Transformers Work*. Medium, Towards Data Science. Retrieved April 5, 2023 from https://towardsdatascience.com/transformers-141e32e69591

Goli, A., Mahmoudi, N., Khazaei, H., & Ardakanian, O. (2021). A Holistic Machine Learning-based Autoscaling Approach for Microservice Applications. CLOSER,

Khaleq, A. A., & Ra, I. (2021). Intelligent autoscaling of microservices in the cloud for real-time applications. *IEEE Access*, *9*, 35464-35476.

Kubernetes, T. (2019). Kubernetes. *Kubernetes. Retrieved May*, *24*, 2019.

Marie-Magdelaine, N., & Ahmed, T. (2020). Proactive autoscaling for cloud-native applications using machine learning. GLOBECOM 2020-2020 IEEE Global Communications Conference,

Marinescu, D. C. (2022). *Cloud computing: theory and practice*. Morgan Kaufmann.

Merrit, R. (2022). *What is a Transformer Model?* Retrieved 27 March from https://blogs.nvidia.com/blog/2022/03/25/what-is-a-transformer-model/

Microsoft. (2020). *2 years of digital transformation in 2 months*. Microsoft. Retrieved Mar 22, 2023 from https://www.microsoft.com/en-us/microsoft-365/blog/2020/04/30/2-years-digital-transformation-2-months/

Phung, H.-D., & Kim, Y. (2022). A Prediction based Autoscaling in Serverless Computing. 2022 13th International Conference on Information and Communication Technology Convergence (ICTC),

Rashid, A., & Chaturvedi, A. (2019). Cloud computing characteristics and services: a brief review. *International Journal of Computer Sciences and Engineering*, *7*(2), 421-426.

Schuler, L., Jamil, S., & Kühl, N. (2021). AI-based resource allocation: Reinforcement learning for adaptive auto-scaling in serverless environments. 2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid),

stefanprodan. (2021). *podinfo*. Retrieved May 5 from https://github.com/stefanprodan/podinfo

TrainDataHub. (2022). *Interpretation of Evaluation Metrics For Regression Analysis (MAE, MSE, RMSE, MAPE, R-Squared, And Adjusted R-Squared)*. Retrieved May 5 from https://medium.com/@ooemma83/interpretation-of-evaluation-metrics-for-regression-analysis-mae-mse-rmse-mape-r-squared-and-5693b61a9833

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., & Polosukhin, I. (2017). Attention is all you need. *Advances in neural information processing systems*, *30*.

Wang, Z., Zhu, S., Li, J., Jiang, W., Ramakrishnan, K., Zheng, Y., Yan, M., Zhang, X., & Liu, A. X. (2022). DeepScaling: microservices autoscaling for stable CPU utilization in large scale cloud systems. Proceedings of the 13th Symposium on Cloud Computing,

Zhang, Z., Wang, T., Li, A., & Zhang, W. (2022). Adaptive Auto-Scaling of Delay-Sensitive Serverless Services with Reinforcement Learning. 2022 IEEE 46th Annual Computers, Software, and Applications Conference (COMPSAC),

Zhong, Z., Xu, M., Rodriguez, M. A., Xu, C., & Buyya, R. (2022). Machine learning-based orchestration of containers: A taxonomy and future directions. *ACM Computing Surveys (CSUR)*, *54*(10s), 1-35.

Appendices:

Appendix A: Configuration files

- Sample App: The deployment file of the sample application is as follows.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: podinfo
spec:
  selector:
    matchLabels:
      app: podinfo
  template:
    metadata:
      labels:
        app: podinfo
    spec:
      containers:
      - name: podinfo
        image: stefanprodan/podinfo
        ports:
        - containerPort: 9898
---
apiVersion: v1
kind: Service
metadata:
  name: podinfo
  labels:
    app: podinfo
spec:
  type: ClusterIP
  selector:
    app: podinfo
  ports:
    - protocol: TCP
      name: web
      port: 8080
      targetPort: 9898
```

- Ingress resource: As mentioned in section (Deploying and integrating different components) an ingress resource was created to integrate the application with the ingress controller. The yaml file was written as follows:

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: podinfo
spec:
  ingressClassName: nginx
  rules:
    - host: "sampleapp.com"
      http:
        paths:
          - backend:
              service:
                name: podinfo
                port:
                  number: 8080
```

- HPA configuration: For evaluating the performance of the implemented model and comparing it with state-of-art methods. HPA was configured. The yaml file consists of the following code.

```
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  name: hpanginx
  namespace: ingress-nginx
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: ingress-nginx-controller
  minReplicas: 1
  maxReplicas: 100
        targetCPUUtilizationPercentage: 70
```

- KEDA Scaledobject: As per section (Deploying and integrating different components) the KEDA scaledobject yaml file was written as follows:

```
apiVersion: keda.sh/v1alpha1
kind: ScaledObject
metadata:
  name: ingress-nginx-workload
  namespace: ingress-nginx
spec:
  scaleTargetRef:
    name: ingress-nginx-controller
  triggers:
  - type: kubernetes-workload
    metadata:
      podSelector: 'app.kubernetes.io/name=ingress-nginx'
      value: "0.877"
```

Appendix B: Training and predicting

- Training the model: The code for training the transformer model for predicting future workload:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from darts import TimeSeries
from darts.dataprocessing.transformers import Scaler
from darts.models import TransformerModel, ExponentialSmoothing
from darts.metrics import mape
from darts.utils.statistics import check_seasonality, plot_acf
# Load the Alibaba Cluster Traces 2021 dataset
df_res_0 = pd.read_csv('MSRTQps_0.csv')
df_res_1 = pd.read_csv('MSRTQps_1.csv')
df_res_2 = pd.read_csv('MSRTQps_2.csv')
df_res_3 = pd.read_csv('MSRTQps_3.csv')
df_res_4 = pd.read_csv('MSRTQps_4.csv')
df_res_5 = pd.read_csv('MSRTQps_5.csv')
df_res_6 = pd.read_csv('MSRTQps_6.csv')
df_res_7 = pd.read_csv('MSRTQps_7.csv')
df_res_8 = pd.read_csv('MSRTQps_8.csv')
df_res_9 = pd.read_csv('MSRTQps_9.csv')
df_res_10 = pd.read_csv('MSRTQps_10.csv')
df_res_11 = pd.read_csv('MSRTQps_11.csv')
df_res_12 = pd.read_csv('MSRTQps_12.csv')
df_res_13 = pd.read_csv('MSRTQps_13.csv')
df_res_14 = pd.read_csv('MSRTQps_14.csv')
df_res_15 = pd.read_csv('MSRTQps_15.csv')
df_res_16 = pd.read_csv('MSRTQps_16.csv')
df_res_17 = pd.read_csv('MSRTQps_17.csv')
df_res_18 = pd.read_csv('MSRTQps_18.csv')
df_res_19 = pd.read_csv('MSRTQps_19.csv')
df_res_20 = pd.read_csv('MSRTQps_20.csv')
df_res_21 = pd.read_csv('MSRTQps_21.csv')
df_res_22 = pd.read_csv('MSRTQps_22.csv')
df_res_23 = pd.read_csv('MSRTQps_23.csv')
# Concatenate the dataframes
df_res = pd.concat([df_res_0, df_res_1, df_res_2, df_res_3, df_res_4,
df_res_5, df_res_6, df_res_7, df_res_8, df_res_9, df_res_10, df_res_11,
df_res_12, df_res_13, df_res_14, df_res_15, df_res_16, df_res_17,
df_res_18, df_res_19, df_res_20, df_res_21, df_res_22, df_res_23])
# Select relevant columns
df = df_res[['msname','timestamp', 'metric', 'value']]
df = df.sort_values(by='timestamp')
df['timestamp'] = pd.to_datetime(df['timestamp'], unit='ms')
#select only one microservice
```

```
msname =
'6190227e8489cba622c6675f26dbf94a0407dc21594d0dcd6776c46975b7e225'
df = df.loc[df['msname'] == msname]
df = df.set_index('timestamp').groupby('metric').resample('15s').mean()
df=df.reset_index()
df.dropna(inplace= True)
metric = 'providerRPC_MCR'
df = df.loc[df['metric'] == metric]
#Keep only the target column
df = df[['timestamp', 'value']]
# Split the data into training and validation sets
train, val = train_test_split(df, test_size=0.30, shuffle=False)
# Create the time series objects
train_series = TimeSeries.from_dataframe(train, 'timestamp', ['value'])
val_series = TimeSeries.from_dataframe(val, 'timestamp',['value'])
df_series = TimeSeries.from_dataframe(df, 'timestamp',['value'])

# Scale the data
scaler = Scaler()
train_series_scaled = scaler.fit_transform(train_series)
val_series_scaled = scaler.fit_transform(val_series)
df_series_scaled = scaler.transform(df_series)

# Create and train the model

model = TransformerModel(
    input_chunk_length=12,
    output_chunk_length=4,
    batch_size=32,
    n_epochs=200,
    model_name="resource_transformer",
    nr_epochs_val_period=10,
    d_model=16,
    nhead=8,
    num_encoder_layers=2,
    num_decoder_layers=2,
    dim_feedforward=128,
    dropout=0.1,
    activation="relu",
    random_state=42,
    save_checkpoints=True,
    force_reset=True,
)
model.fit(series=train_series_scaled, val_series=val_series_scaled,
verbose=True)
# Predict
prediction = model.predict(n=len(val_series_scaled))
prediction_unscaled = scaler.inverse_transform(prediction)
```

```
# Evaluate the performance using MAPE
mape_error = mape(val_series, prediction_unscaled)
print(f"MAPE error: {mape_error:.2f}")
from darts.metrics import mae
# Calculate MAE on the test set
val_mae_error = mae(val_series, prediction_unscaled)
print(f"MAE error on validation set: {val_mae_error:.2f}")

from darts.metrics import rmse

# Calculate RMSE on the validation set
rmse_error = rmse(df_series, prediction_unscaled)
print(f"RMSE error: {rmse_error:.2f}")
from darts.metrics import smape

# Calculate SMAPE on the validation set
smape_error = smape(prediction_unscaled, df_series)
print(f"SMAPE error: {smape_error:.2f}")
```

- Prediction of workload: The model was used to predict data based on previous one hour of data scraped from prometheus. The script is as follows.

```
import requests
from datetime import datetime, timedelta
import pandas as pd
import numpy as np

from darts import TimeSeries
from darts.dataprocessing.transformers import Scaler
from darts.models import TransformerModel, ExponentialSmoothing

PROMETHEUS_URL = 'http://10.108.86.201:9090/api/v1/query_range'

# Set the start time and end time for the query
end_time = datetime.now()
start_time = end_time - timedelta(hours=1)

# Construct the PromQL query
query =
'sum(irate(nginx_ingress_controller_requests{controller_pod=~"ingress-
nginx-controller-.*",controller_namespace=~"ingress-
nginx",ingress=~"podinfo-ingress”}-[1m]))'

# Make the query request to Prometheus
url =
f'{PROMETHEUS_URL}?query={query}&start={int(start_time.timestamp())}&en
d={int(end_time.timestamp())}&step=15s'
#print(f'Request URL: {url}')
response = requests.get(url)
#print(f'Response JSON: {response.json()}')

# Parse the response data into a tabular format
data = response.json()['data']['result']
```

```
rows = []
for result in data:
    values = result['values']
    for value in values:
        time = datetime.fromtimestamp(value[0]).strftime('%m/%d/%Y
%H:%M')
        rps = value[1]
        rows.append((time, rps))

# Create a pandas DataFrame
df = pd.DataFrame(rows, columns=['Time', 'RPS'])

df["RPS"] = pd.to_numeric(df["RPS"], downcast="float")

df = df.groupby(['Time']).mean()
df.dropna(inplace= True)
df=df.reset_index()
# Print the DataFrame
print(df)


# Create the time series objects
df_series = TimeSeries.from_dataframe(df, 'Time',['RPS'])

# Scale the data
scaler = Scaler()
df_series_scaled = scaler.transform(df_series)

import joblib
# Load the trained model from the joblib file
model = joblib.load('transformer_model.joblib')

prediction= model.predict(n=4, df_series)
prediction_unscaled = scaler.inverse_transform(prediction)
print(prediction_unscaled)

#save the predicted data
df_pred=prediction_unscaled.pd_dataframe(copy=True,
suppress_warnings=False)
df_pred.to_csv('predicted_values.csv')
```

Appendix C: Target Relation Calculation

- Calculation Script: The following script calculates the parameter for the "value" field of the scaledobject based on the predicted value and current pod status.

```
import pandas as pd
import requests
from datetime import datetime, timedelta
import yaml
import subprocess

# Load the YAML file
with open("scaledobject-podinfo.yaml", "r") as f:
    scaledobject = yaml.safe_load(f)

# Load the CSV file
df = pd.read_csv('predicted_values.csv')

# The maximum RPS value in the 'RPS' column of the CSV file
max_rps = df['RPS'].max()
# Max RPS per pod based on load testing
max_rps_per_pod = 330

# Calculate the number of pods required to handle the predicted request
rate
num_pods_required = max_rps / max_rps_per_pod

# Set the desired utilization percentage based on workload and cluster
capacity
desired_utilization_percentage = 70

# Get the current number of pods running
PROMETHEUS_URL = 'http://10.108.86.201:9090/api/v1/query'

# Construct the PromQL query to get the number of pods running nginx
container
query = 'sum(kube_pod_info{pod=~"ingress-nginx-controller-.*"})'

# Make the query request to Prometheus
url = f'{PROMETHEUS_URL}?query={query}'
response = requests.get(url)

# Extract the current number of pods running the container from the
response
data = response.json()['data']['result']
num_matching_pods = int(data[0]['value'][1])

# Calculate the value for the KEDA 'value' field
scaled_workload_pods = num_pods_required
value = (num_matching_pods / scaled_workload_pods) *
desired_utilization_percentage / 100

# Set the new value in the scaledobject YAML
scaledobject['spec']['triggers'][0]['metadata']['value'] = str(value)

# Write the updated scaledobject YAML file
```

```
with open("scaledobject-podinfo.yaml", "w") as f:
    yaml.dump(scaledobject, f)

# Apply the updated scaledobject YAML file
subprocess.run(["kubectl", "apply", "-f", "scaledobject-podinfo.yaml"])
```