**ÇUKUROVA UNIVERSITY**
**INSTITUTE OF NATURAL AND APPLIED SCIENCES**

**MSc THESIS**

**Anıl Kuşçu**

# EFFICIENT SCALING WITH MACHINE LEARNING ON CLOUD ENVIRONMENT

**DEPARTMENT OF COMPUTER ENGINEERING**

**ADANA, 2023**

## ABSTRACT

**MSc THESIS**

**EFFICIENT SCALING WITH MACHINE LEARNING ON CLOUD ENVIRONMENT**

**Anıl KUŞÇU**

**ÇUKUROVA UNIVERSITY**
**INSTITUTE OF NATURAL AND APPLIED SCIENCES DEPARTMENT OF COMPUTER ENGINEERING**

Supervisor : Prof. Dr. M. Fatih AKAY
Year: 2023, Pages: 73
Jury : Prof. Dr. M. Fatih AKAY
: Assoc. Prof. Dr. Mehmet Uğraş CUMA
: Assoc. Prof..Dr. Fatih KILIÇ

Scaling in cloud environments can significantly affect the cost and efficiency of a system. In order to properly plan capacity, calculations are often made based on scaling size. Applications typically reserve resources such as cores, memory, and network in order to maintain high quality of service (QoS). When resource limits are approached, the application is scaled by running multiple copies in order to ensure system reliability under increasing traffic. This study aims to more efficiently reserve resources in order to maintain high QoS through the use of machine learning. Traditional methods reserve resources for an application and scale the application when resource usage reaches a certain threshold. However, the method implemented in this study estimates incoming traffic and scales applications based on this estimation, resulting in more efficient performance compared to using a fixed threshold value. The results of the study showed that the scaling system created based on machine learning performs more efficiently in terms of cost and resource allocation compared to the currently used static scaling methods.



**Keywords:** Machine Learning, Cloud Computing, Scaling, Kubernetes, Resource Management, Cloud Cost

# ÖZ

## YÜKSEK LİSANS TEZİ

## BULUT ORTAMINDA MAKİNE ÖĞRENİMİ İLE VERİMLİ ÖLÇEKLENDİRME

**Anıl KUŞÇU**

**ÇUKUROVA ÜNİVERSİTESİ**
**FEN BİLİMLERİ ENSTİTÜSÜ**
**BİLGİSAYAR MÜHENDİSLİĞİ ANABİLİM DALI**

Danışman : Prof. Dr. M. Fatih AKAY
Yıl: 2023, Sayfa: 73
Jüri : Prof. Dr. M. Fatih AKAY
: Doç. Dr. Mehmet Uğraş CUMA
: Doç. Dr. Fatih KILIÇ

Bulut ortamlarında yapılan ölçeklendirme, sistemin maliyet ve verimliliğini önemli ölçüde etkileyebilir. Kapasiteyi düzgün şekilde planlamak için, ölçeklendirme boyutuna göre hesaplamalar yapılır. Uygulamalar genellikle yüksek hizmet kalitesi (QoS) sağlamak için çekirdekler, bellek ve ağ gibi kaynakları rezerve eder ve kaynak sınırlarına yaklaşıldığında, uygulamanın birden fazla kopyasını çalıştırarak sistem güvenilirliğini arttıran bir şekilde ölçeklendirilir. Bu çalışma, makine öğrenimi kullanarak yüksek QoS'u koruma amacıyla kaynakları daha verimli bir şekilde rezerve etmeyi amaçlamaktadır. Geleneksel yöntemler, uygulamanın ihtiyaç duyduğu kaynakları rezerve eder ve kaynak kullanımı belirli bir seviyeye ulaştığında uygulamayı ölçeklendirir. Ancak, bu çalışmada uygulanan yöntem, gelen trafiği tahmin eder ve bu tahmine göre uygulamaları ölçeklendirir, Bu sayede sabit eşik değerine göre ölçeklendirme yapıldığında daha verimli bir performans elde edilir. Çalışmanın sonuçları, makine öğrenimi temelinde oluşturulan ölçeklendirme sisteminin mevcut olarak kullanılan statik ölçeklendirme yöntemlerine göre maliyet ve kaynak ayrılması performansı açısından daha verimli çalıştığını göstermiştir.

**Anahtar Kelimeler:** Makine Öğrenmesi, Bulut Bilişim,Ölçeklendirme, Kubernetes, Kaynak Yönetimi, Bulut Maliyeti

## EXTENDED ABSTRACT

The use of cloud services to host applications has gained widespread popularity due to the convenience, speed, cost advantage, support options, and personalized service provided by the cloud environment. This popularity has led to the emergence of new fields, problems, solutions, and studies focused on goals such as efficient resource utilization and cost reduction. In the field of cloud computing, performance and cost dynamics are constantly evolving, and this has significant implications for companies offering software as a service (SaaS). These companies often seek the most efficient and fastest ways to provide their services, and one important tool for achieving efficiency is capacity planning.

Capacity planning involves calculating the expected load on the system and the amount of resources required to maintain a high quality of service (QoS). To ensure high QoS, companies often allocate more resources than necessary to their applications. The higher the reserved idle resources, the more resilient the system will be to future loads. However, mistakes in capacity planning can lead to significant cost issues. Therefore, accurate capacity planning is crucial for companies.

There have been several studies in the literature on efficient scaling in cloud environments, which have addressed various aspects of this topic. These studies have often used simulations on different systems, but few have used a real environment and application, and even fewer have provided comparative and detailed results with the current system. This thesis aims to simulate a real product development environment and to provide results that are as close to reality as possible. The use of machine learning to predict the future has become commonplace, and this thesis aims to increase efficiency in scaling machine learning and deep learning methods in a cloud environment.

The main objective of this thesis is to use machine learning and deep learning methods to predict incoming traffic to the system and create application scaling

accordingly. To achieve this, the Extended Gradient Boosting, ARIMA, Long Short Time Memory, Multilayer Perceptron, and Hierarchical Temporal Memory algorithms from machine learning methods were applied to a dataset, and the outputs of each algorithm were compared. The Extended Gradient Increasing method was found to give the best results, and scaling was performed using this method. The dataset used for the estimation model developed in this study combines electricity market data from 01.07.2021 - 31.12.2021 and data from the SmartPulse company for the same period. The data was compiled from the Energy Markets Operation Inc. in Turkey (EPİAŞ) website (https://seffaflik.epias.com.tr) and the traffic logs of SmartPulse (https://www.smartpulse.io). The performances of the machine learning methods were compared and evaluated based on the average error percentage (MPE), CPU usage, and runtime. The average error percentage (MPE) was chosen as the primary metric for evaluation, as the number of users is the most important element in the system. All methods were implemented in Python (version 3.9) using the sklearn library.

Algorithms and libraries developed using machine learning techniques have certain basic parameters, known as hyperparameters, that significantly influence the output values and overall performance. In this study, the hyperparameters were optimized using a method called grid search, which tests different combinations of hyperparameter values within a specified range to find the optimal results. The dataset used in this thesis was organized as a time series, with data values ordered by time.

The machine learning component of the system consists of two parts. The first part is run once a day, during which transparency data from EPİAŞ is obtained and the relevant database is updated. Then, data is retrieved from this updated database and hyperparameters are optimized using the grid search method. The resulting optimized hyperparameters are used for traffic prediction. Every 60 minutes, the system uses these current hyperparameters to make traffic predictions and scales the service accordingly. This proactive horizontal pod autoscaler (PHPA) system is

written as an operator in Kubernetes, a container orchestration tool commonly used in cloud environments, and compared to the horizontal pod autoscaler (HPA) mechanism, which is the scaling system of the Kubernetes system. As the scaling mechanism in the Kubernetes system was created using fixed threshold values, a 2-day load simulation was simultaneously sent to the proactive horizontal pod autoscaler and the Kubernetes scaling systems with thresholds of 50%, 60%, 70%, and 80%, and the results were compared. As EPİAŞ announces relevant data two days in advance, a 2-day traffic forecast can be made each time. A load test simulation was created using the compiled dataset, in which approximately 2 hours of 2-day user activity were projected and the planned load was separately sent to all systems.

Microsoft Azure was used as the cloud infrastructure in this study. A Kubernetes (version 1.18) infrastructure was set up on Azure and all applications were containerized and deployed on this infrastructure. A proxy application, nginx ingress, was used to expose the application endpoints to the external environment. The metric collection system, Prometheus (version 2.21), was used to monitor the system based on metrics in the cloud environment. Grafana (version 8.4) was used to graphically display the collected metrics and data. All of these installations in the cloud environment were implemented using the configuration as code approach discussed in this study.

Apache JMeter (version 5.4.1) was used for the load test and real user data was obtained from Smartpulse company to simulate hourly real user behavior in the load test. Since the results were measured over a period of two days, these two days of user activity were condensed into two hours and the two days' worth of user activity was sent to the system in two hours. In order to perform a comprehensive comparison, a fan out service was first implemented to distribute the incoming load and the incoming load was sent to all experimental environments simultaneously.

For comparison, the aforementioned Proactive (PHPA) scaling systems, as well as those with 80%, 70%, 60%, and 50% threshold values, were evaluated based

on various metrics. The results of this evaluation led to the following conclusions: the PHPA system outperformed the others in terms of CPU Request and Memory Request metrics, while all of the scaling systems performed approximately equally or very closely in terms of Response Time, Error Rate, CPU Usage, and Memory Usage metrics. Additionally, the Proactive Scaling system has a cost advantage over the others, as it generates fewer resource request values in the cloud.

## GENİŞLETİLMİŞ ÖZET

Uygulamaları barındırmak için bulut hizmetlerini kullanmak günümüzde çok popüler. Bunun nedeni bulut ortamının sağladığı kolaylık, hız, maliyet avantajı, destek seçenekleri ve kişiselleştirilmiş hizmet gibi avantajlardır. Bu popülerlik nedeniyle zaman içinde yeni alanlar, yeni sorunlar, yeni çözümler ve yeni çalışmalar önem kazanmıştır. Elbette bu çabalar, kaynakların verimli kullanılması ve maliyetlerin düşürülmesi gibi hedeflere dayanmaktadır. Bulut bilişim alanında da performans ve maliyet dinamikleri çok hızlı bir şekilde değişmektedir. Bu değişim hizmet olarak yazılım(Software as a Service) servisleri sunan şirketleri ve yazılımları da etkilemektedir. Bu hizmeti veren şirketler de bunu yapmanın en verimli ve hızlı yollarını kullanırlar. Verimlilik konusunda en önemli araçlardan birisi de kapasite planlamadır. Kapasite planlanırken sisteme gelecek yük , kullanılacak kaynak miktarı önceden hesaplanarak her zaman servis kalitesi (Quality of Service) en üst düzeyde olması istenir. Bunun için de uygulamaya gereğinden fazla kaynak tahsis edilerek hizmetin kalitesi arttırılır. Rezerve edilen atıl kaynak ne kadar yüksek ise sistem gelecek yüke karşı o kadar dayanıklı olur. Ayrıca bu planlamada yapılan hatalar daha büyük maliyet problemlerine neden olmaktadır. Bu nedenle doğru kapasite planlaması yapmak şirketler için hayati önem taşımaktadır.

Literatürde bulut ortamlarında verimli ölçeklendirme ile ilgili yapılmış bazı çalışmalar bulunmaktadır. Bugüne kadar yapılan çalışmalarda konuyla ilgili farklı noktalara değinilmiştir. Bu konu ile ilgili farklı çalışmalarda farklı sistemler üzerinde simülasyonlar yapılmıştır. Çalışmalarda şu anki sistem ile karşılaştırmalı ve detaylı sonuçlara yer verilmemesi dışında gerçek bir ortam ve uygulama kullanılmadan deneyler yapılmıştır. Bu tezde gerçek bir ürün geliştirme ortamı simüle edilerek gerçeğe en yakın sonuçların alınması planlanmıştır. Günümüzde geleceğe yönelik yapılan tahminlerde makine öğrenmesi kullanılması oldukça yaygındır. Bu tezde, makine öğrenimi ve derin öğrenme yöntemlerini bulut ortamında ölçeklendirme işlemlerinde verimliliği artırmak amaçlanmıştır.

Bu tezin temel amacı, makine öğrenimi ve derin öğrenme yöntemlerini kullanarak sisteme gelecek trafiği tahmin etmek ve buna uygun uygulama ölçeklendirmesi oluşturmaktır. Bu çalışmada makine öğrenmesi yöntemlerinden Genişletilmiş Gradyan Artırma(Extended Gradient Boosting) , ARIMA, Long Short Time Memory, Multilayer Perceptron,Hierarchical Temporal Memory Algoritmaları veri seti ile beraber çalıştırılmış ve her bir algoritmanın çıktıları karşılaştırılmıştır. Genişletilmiş Gradyan Artırma yöntemi daha iyi sonuç verdiği için bu yöntemi kullanarak ölçeklendirme işlemi yapılmıştır. Tez çalışması boyunca geliştirilen tahmin modelinde 01.07.2021 - 31.12.2021 dönemi elektrik piyasası veri seti ve aynı tarihlere ait SmartPulse firmasına ait veri seti birleştirilerek kullanılmıştır. Veri seti Türkiye'de Enerji Piyasaları İşletme A.Ş (EPİAŞ) tarafından verilerin yayınlandığı (https://seffaflik.epias.com.tr) web sitesinden ve Smart Pulse (https://www.smartpulse.io) firmasına ait trafik loglarından derlenmiştir. Kullanılan makine öğrenimi yöntemlerinin performansları ortalama hata yüzdesi (MPE) değeri , CPU kullanımı , Çalışma Süresi hesaplanarak karşılaştırılmış ve değerlendirilmiştir. Bu ölçümde sadece ortalama hata yüzdesi (MPE) hesaplanma nedeni sistemdeki en önemli unsurun kullanıcı sayısı olması dolayısıyla hata ölçümünde bu yöntemin daha isabetli olacağı düşünülmüştür.Tüm bahsedilen yöntemler python programlama dili (versiyon 3.9) ile sklearn kütüphanesi kullanılarak yazılmıştır.

Makine öğrenimi teknikleri kullanılarak oluşturulan algoritmalar ve kütüphaneler bazı temel parametreler içermektedir. Bu parametreler çıktı değerlerin oluşturulmasında önemli bir yer tutar ve sonuçları önemli ölçüde etkiler. Bu parametreler hiper parametre olarak da adlandırılır. Bu çalışmada bu hiper parametreleri optimize etmek için ızgara arama(grid search) olarak bilinen yöntem kullanılmıştır. Bu yöntem verilen aralıklardaki hiper parametre değerlerini tek tek deneyerek en optimum sonuçlara ulaşır. Tezde kullanılan veri seti zaman serileri halinde derlenmiştir. Veri setindeki değerler zamana göre sıralanmış ve karşılık gelen veriler kaydedilmiştir.

Sistemin makine öğrenmesi kısmı 2 parçadan oluşmaktadır. İlk parça günde 1 kere çalışarak önce EPİAŞ şeffaflık verilerini çeker ve ilgili veritabanını günceller. Ardından bu güncellenmiş veritabanından tüm veriler çekilerek ızgara arama yöntemi ile hiper parametreler güncellenir. Böylece tahmin yapılacağı zaman en optimize hiper parametreler kullanılır. Diğer kısımda ise güncel hiper parametreler kullanılarak her 60 dakikada bir trafik tahmini yapılır. Yapılan trafik tahminine göre servis ölçeklendirilir. Bu ölçeklendirilen ölçeklendirme sistemi Proaktif Yatay Kapsül Oto Ölçekleyici(Proactive Horizontal Pod Autoscaler) olarak adlandırılmıştır. Bulut ortamında konteyner orkestrasyonu için kullanılan Kubernetes içinde operator olarak yazılan bu sistem ile Kubernetes sisteminin ölçeklendirme sistemi olan Yatay Kapsül Oto Ölçekleyici(Horizontal Pod Autoscaler) mekanizması karşılaştırılmıştır. Kubernetes sistemindeki ölçeklendirme mekanizması belli eşik değerler(threshold) verilerek oluşturulduğu için %50,%60,%70,%80 lik eşik değerli Kubernetes ölçeklendirme sistemleri ile Proaktif Yatay Kapsül Oto Ölçekleyiciye aynı anda 2 günlük yük simulasyonu gönderilmiştir ve sonuçlar karşılaştırılmıştır. EPİAŞ ilgili verileri 2 gün önceden açıklandığı için her seferinde 2 günlük trafik tahmini yapılabilmektedir. Derlenmiş veri seti kullanılarak bir yük testi simulasyonu oluşturulmuştur. Bu yük testinde 2 günlük kullanıcı aktivitesi yaklaşık olarak 2 saate projeksiyon edilerek ayrı ayrı tüm sistemlere planlanan yük gönderilmiştir.

Bulut altyapısı olarak Microsoft Azure kullanılmıştır. Azure ortamında bir kubernetes(versiyon 1.18) altyapısı oluşturulmuş ve tüm uygulamalar konteyner(docker) haline getirildikten sonra bu altyapı üzerinde konumlandırılmıştır. Uygulama uç noktalarını dış dünyaya açmak için nginx uygulamasına ait bir vekil uygulama(nginx ingress) kullanılmıştır. Sistemi bulut ortamında metrikler bazında izlemek için prometheus(versiyon 2.21) adlı metrik toplama sistemi kullanılmıştır. Toplanan metrikleri ve verileri grafik olarak görüntüleme için grafana(versiyon 8.4) kullanılmıştır. Bulut ortamındaki tüm bu kurulumlar kod olarak konfigürasyon yaklaşımı ile oluşturulmuştur. Bu tez boyunca

bahsedilen

Yük testi için Apache JMeter(versiyon 5.4.1) uygulaması kullanılmıştır ve Smartpulse firmasından gerçek kullanıcı verileri alınarak saatlik olarak gerçek kullanıcı davranışlar bu yük testinde simüle edilmniştir. Sonuçlar 2 günlük olarak ölçüldüğü için bu 2 günlük kullanıcı aktivitesi 2 saate projeksiyon edilmiştir ve 2 günlük kullanıcı aktivitesi 2 saatte sisteme gönderilmiştir. Tam olarak bir karşılaştırma yapabilmek için sisteme gelen yükü önce bir çoğaltıcı(fan out) servis konumlandırılmıştır ve gelen yük aynı anda tüm deney ortamlarına gönderilmiştir.

Karşılaştırma için yukarıda bahsedilen Proaktif(PHPA) , %80 eşik, %70 eşik, %60 eşik ,%50 eşik ölçeklendirme sistemleri çeşitli metrikler bazında değerlendirilmiştir. Bu değerlendirme sonucunda aşağıdaki sonuçlara ulaşılmıştır.

İşlemci isteği(CPU Request) ve Bellek isteği(Memory Request) metrikleri bazında PHPA diğer yöntemlere göre daha iyi performans göstermiştir. Cevap zamanı(Response Time), Hata oranı(Error Rate), İşlemci kullanımı(CPU Usage) ve Bellek kullanımı(Memory Usage) metrikleri bazında bütün ölçeklendirme sistemleri yaklaşık olarak eşit veya çok yakın performans göstermiştir. Proaktif Ölçeklendirme sisteminde bulut kaynaklarındaki istek değerleri daha az olduğu için maliyet anlamında bu ölçeklendirme sistemi diğerlerine göre avantaj sağlamıştır.

**ACKNOWLEDGEMENT**

Foremost, I would like to express my sincere gratitude to my advisor Prof. Dr. M. Fatih AKAY, for his motivation, patience and enthusiasm. His guidance helped me in all the time of research and writing of this thesis.

**CONTENT** **PAGE**

**LIST OF TABLES** **PAGE**

**LIST OF FIGURES** **PAGE**

## LIST OF ABBREVIATIONS

EPIAS : Energy Markets Operation Joint Stock Company

HPA : Horizontal Pod Autoscaler

MPE : Mean Percentage Error

PHPA : Proactive Horizontal Pod Autoscaler

# 1. INTRODUCTION

## 1.1. Overview of ML Based Scaling on Cloud

The popularity of hosting applications on cloud services has been increasing in recent times due to the various benefits it offers, including convenience, speed, cost efficiency, support options, and personalized service. The growing popularity of the cloud environment has led to the emergence of new areas of study, research, and solutions focused on optimizing resource usage and reducing costs. This study aims to design a system that can effectively achieve such goals as efficiency and cost-effectiveness. One of the advantages of hosting applications on the cloud is the ability to access resources on demand, which can help to reduce costs by only paying for what is actually used. Another benefit is the ability to scale resources up or down as needed, depending on the workload. Additionally, hosting applications on the cloud can provide improved security, as cloud service providers often have robust security measures in place to protect data and resources.

Resource management in the cloud environment is a complex issue that continues to be improved upon. Despite progress in this area, there are still challenges to be addressed. Therefore, ongoing research focuses on developing automation tools, alarm systems, virtualization, and artificial intelligence solutions to facilitate resource management and make it more efficient and cost-effective. This study aims to use the Extended Gradient Boosting (XGBoost) machine learning technique to improve resource management in the cloud environment.

Resource management in a cloud computing environment can be a complex task, and scaling is one approach that is commonly used to address this challenge. There are two main categories of scaling: scale up/down and scale in/out. Scale up involves upgrading an existing resource by replacing it with a more powerful one, while scale out involves replicating the existing infrastructure and running it in parallel, thus increasing the capacity of the infrastructure linearly. By using replicas of the application on the expanded infrastructure, this technique ensures that

resources are used efficiently and effectively. In addition to facilitating resource management, scaling can also help to optimize cost and performance in a cloud environment.

### 1.2. Motivation for ML Based Scaling on Cloud

Scale out is a critical aspect of the cloud environment because it enables applications to handle excess workload by replicating them and distributing the load across the replicas using a load balancer. This allows for dynamic handling of incoming workloads and efficient resource utilization through scale-in, which reduces the number of replicas when workload decreases to prevent unnecessary resource consumption. Proper use of scale out can improve the performance and availability of applications in the cloud, as it enables them to scale up or down as needed to meet changing demand. Additionally, it can help to reduce costs by ensuring that resources are only used when they are needed.

However, the scaling process can be costly as it requires the determination of a threshold value, often a resource type value such as CPU or memory usage, before scaling out can occur. When this threshold value is exceeded, such as when incoming traffic causes CPU usage to exceed a certain percentage, the scale out process is initiated, resulting in an increase in the number of replicas of the application to handle the increased load. This allows the system to operate effectively under heavy workloads.

While this method of scaling has some benefits, it also has trade-offs. For instance, setting a high threshold value can increase the likelihood of failure under sudden workloads, while setting a low threshold value can lead to inefficient resource usage due to idle resources waiting to meet the threshold requirement. In this study, estimates of incoming workloads will be created using a machine learning model and the scale operation will be performed based on these estimates to optimize efficiency and performance.

Implementing autoscaling in a Kubernetes system can provide numerous

benefits, including the automatic optimization of resource usage by creating only the number of replicas necessary to handle current traffic levels. Without autoscaling, a system maintains a fixed number of replicas regardless of traffic levels, which can lead to resource exhaustion and potential system saturation or failures, resulting in significant costs for businesses. Autoscaling helps to avoid these issues by dynamically adjusting the number of replicas based on incoming traffic, ensuring that the system has the resources it needs to handle the current workload and avoid saturation or failures. Overall, implementing autoscaling in a Kubernetes system can improve resource utilization, system performance and availability, and reduce costs.

AI-based auto scaling can improve system performance and availability, along with cost savings. In a traffic or load peak moment, it can scale resources to suit this traffic very quickly. This can be critical for customer-facing applications as well as systems that need to maintain a high level of uptime and responsiveness. In addition, autoscaling systems help ensure that the system can handle traffic spikes without downtime or performance degradation. A system that can scale horizontally at an increased traffic rate in response to increased traffic can maintain its performance and availability. In addition to these benefits, autoscaling also frees up the complexity of managing a Kubernetes system. With automatic scaling, the system can process changes in traffic levels without the need for manual intervention, freeing up time and resources that can be spent on other tasks. Implementing autoscaling in a Kubernetes system provides numerous benefits, including improved resource utilization, better performance and availability, and easier management.

### 1.3. Types of Scaling on Cloud

Autoscaling using machine learning in Kubernetes is a powerful feature that enables the deployment of a system based on incoming traffic or adjusts the number of replicas of services dynamically. This allows the system to manage traffic fluctuations and ensure that the necessary resources are available to service incoming

requests, and it is more efficient than threshold-based scaling methods currently in use. Implementing machine learning-based autoscaling on a Kubernetes system can bring numerous benefits, such as optimizing resource usage and reducing costs by dynamically adjusting the number of replicas based on incoming traffic rather than maintaining a fixed number of copies regardless of traffic levels. In addition, this type of scaling can improve the performance and availability of the system by ensuring that the appropriate number of replicas are available to handle incoming workloads, and it can reduce the complexity of resource management by automating the process of adjusting the number of replicas based on workloads.

### 1.4. Purpose and Contributions of the Thesis

There is a wealth of literature on both machine learning and cloud computing, as well as studies that combine the two fields. These studies have been conducted on various platforms, systems, infrastructures, and datasets, and their results may not be directly applicable to production environments. In the literature, external datasets are often used, which is understandable given the difficulty of accessing real platform data. This study will utilize a dataset compiled from multiple sources, including user data from the Smartpulse platform and various pricing and forecast values that periodically impact platform access. This aspect of the study differentiates it from others in the literature.

In terms of cloud computing and system considerations in this study, a novel Kubernetes operator will be developed that communicates directly with the API and manages the scaling process within the system. This operator will include both the trained machine learning code and the code to perform scaling operations. This approach to scaling differs from other studies in that it involves direct interaction with the system. Additionally, this operator will enable the integration of machine learning capabilities into the system's scaling process, providing the potential for improved efficiency and performance. Overall, the implementation of this Kubernetes operator represents a significant contribution to the existing literature on

cloud computing and system management.

In this study, an attempt will be made to design a system that effectively achieves goals such as efficiency and cost-effectiveness in the cloud environment by utilizing machine learning techniques. To do so, a new Kubernetes operator will be developed which can interact with the API and manage the scaling process in the system. This operator will contain both the trained machine learning algorithms and the code to perform the required operations. The use of realistic data obtained from the production environment, as well as simulated user behaviors in the same environment, will allow for the generation of results that are representative of those encountered in real-world scenarios. As a result, this study has the potential to contribute to both the fields of cloud computing and machine learning, and could serve as a valuable resource for future research endeavors.

### 1.5. Dataset Generation

The dataset was obtained by combining two different datasets. The first dataset was created by getting data from the EPİAŞ Transparency platform between dates 01.07.2021 - 31.12.2021. The second data set was obtained from the Smartpulse platform itself. This data set includes user traffic data of the platform. By combining these two data sets, a comprehensive data set was created to be used in the machine learning model. In addition, data cleaning and preprocessing processes were applied to the data set to ensure the quality and reliability of the data. In this way, a high-quality data set was created that can accurately represent the real-world conditions of the platform. As a result of this study, a machine learning model was trained using this data set to predict the traffic that the platform will receive in the next few hours.

Table 1.1. Overview of Dataset

| **Variables** | **Explanation** |
|---|---|
| Timestamp | Timestamp |
| Traffic | User Traffic |
| Weighted Average Price | A price calculated by taking into account the quantity and price of each individual trade in a specific period of time. |
| Day Ahead Market | A market where electricity is traded for the following day. |
| Primary Frequency Control | A power system control measure that aims to keep the frequency of the electricity grid within a certain range by regulating the generation and consumption of electricity. |
| Secondary Frequency Control | A power system control measure that aims to fine-tune the frequency of the electricity grid by quickly responding to changes in demand and generation. |
| Bilateral Agreements | Contracts between two parties for the exchange of electricity or related services. |
| Real Time Production | The amount of electricity that is being generated in real time. |
| Trading Volume | The total amount of electricity that is traded in a specific period of time. |
| Real Time Consumption | The amount of electricity that is being consumed in real time. |
| Finalized Daily Electricity Generation Plans | The plans for the generation of electricity for the following day have been finalized and approved. |
| Received Load Order | An instruction to increase or decrease the amount of electricity being consumed or generated. |
| Load Shedding Instruction | An instruction to reduce the amount of electricity being consumed in order to prevent a power outage. |
| Available Capacity | The amount of electricity that can be generated or consumed at a specific time. |
| Finalized Daily Production Plan | The plans for the generation of electricity for the following day have been finalized and approved. |
| Market Exchange Price | The price at which electricity is traded in a market. |

### 1.6. Roadmap of the Thesis

This dissertation consists of six chapters. Chapter 2 thoroughly reviews the previous research in relevant literature. Chapter 3 provides an overview of the machine learning and cloud computing techniques utilized in this dissertation. Chapter 4 explains the research methodology and details the system's development. Chapter 5 presents the findings and discussions. Finally, Chapter 6 concludes the study.

## 2. LITERATURE REVIEW

This study aims to contribute to the literature by combining machine learning and cloud computing in the context of resource allocation in the cloud environment. By using machine learning techniques to forecast web traffic and load, the system can dynamically adjust the number of replicas of a service in a cloud environment to optimize resource usage and reduce costs. This not only helps improve system performance and availability, it also has the potential to significantly reduce costs by avoiding the wastage of resources. In addition, this study will use real data from a production environment to create a more realistic simulation and provide a valuable source of information for future studies.

### 2.1. Kubernetes Related Studies

Tamiru et al compared configurations using two representative applications and workloads on Google Kubernetes Engine (GKE). They found that it was possible to configure Kubernetes' Cluster Autoscaler to select nodes from a single node pool or multiple node pools. They reported their results using approved monetary cost and standard auto scaling performance metrics from the SPEC Cloud Group. They demonstrated that their self-developed CA-NAP outperforms the default CA, and that auto scaling performance primarily depends on the composition of the workload.

In their study, Luciano Baresi et al propose KOSMOS, a new autoscaling solution for Kubernetes designed to address the limitations of existing solutions such as the Horizontal Pod Autoscaler (HPA) and Vertical Pod Autoscaler (VPA). These limitations include the need to control multiple metrics in HPA and VPA, the requirement to define simple threshold-based rules, and the need to stop and restart existing containers, leading to resource contention and inefficient resource utilization. To address these issues, KOSMOS is a system that is individually controlled by controllers that manage container resources in real-time. A custom component handles resource contention scenarios between containers deployed on

the same node, and a heuristic-based controller is responsible for horizontal scaling of each application at the cluster level. Overall, the KOSMOS scaling system aims to improve resource management and efficiency in Kubernetes environments.

In their study, Zhijun Ding et al. proposed a new combined scaling method called COPA, which utilizes microservice performance data, real-time workload, expected response time, and a runtime microservice replica scheme calculated using the queue network model to determine a unified scaling scheme that aims to minimize both default cost and resource cost. The method was tested on a Kubernetes cluster and compared to state-of-the-art auto scaling methods under four different workload types. The results of the experiments showed that COPA achieved a 1.22x reduction in resource cost while maintaining Quality of Service (QoS) compared to the basic method.

In their paper, Dang-Quang et al. propose a Kubernetes-based system architecture with a proactive custom autoscaler that utilizes a deep neural network model to dynamically adjust workloads during runtime. The proposed system architecture is based on the Watch-Analyze-Plan-Execute cycle, and uses a Bidirectional Long Short Term Memory (Bi-LSTM) model to predict the number of future HTTP workloads. The Bi-LSTM model achieves better accuracy than other models according to the experiments conducted, and also offers 530 to 600 times faster prediction speed than ARIMA models with different workloads. In comparison with the LSTM model, the Bi-LSTM model performs better in terms of welding accuracy and elastic acceleration. The proposed proactive custom autoscaler is also shown to outperform Kubernetes' default horizontal pod autoscaler (HPA) in terms of accuracy and speed when provisioning and deprovisioning resources. The system also includes a resource removal strategy to remove some resources when workload is reduced.

In their study, Laszlo Toka and colleagues from the MTA-BME Network Softwarization Research Group at the Budapest University of Technology used requests from the Budapest University of Technology to the Facebook application

as the dataset. They developed their own HPA+ engine as an alternative to the Horizontal Pod Autoscaler (HPA) system in Kubernetes. This engine adjusts the scale ratio of pods in the system based on a decision made by applying either the HPA method or other machine learning algorithms in HPA+. They reported successful responses to requests and indicated their intention to continue improving the model.

In their study, Tengfei Hu and Yannian Wang from Zhengzhou University propose a Kubernetes autoscaling system based on pod replica prediction. They developed a predictive Horizontal Pod Autoscaler (HPA) that takes into account environmental conditions and response times in order to perform predictive scaling. Their approach considers the impact of resource consumption in pods on response time. Through experimentation, they demonstrate that their predictive autoscaler has the advantage of faster response speed compared to other methods.

In his study, Rossi F. compared the scaling policies created using machine learning methods with Kubernetes' default Horizontal Pod Autoscaler (HPA) based on Quality of Service (QoS) metrics. The results of the experiment are presented in Table 2.1.

Table 2.1. Application performance under different scaling policies

| **Scaling Method** | **Average CPU Utilization** | **Average Number of Pods** | **Median of Response Time** |
|---|---|---|---|
| Model Based RL | 48.51 | 3.75 | 16.28 |
| Q-Learning | 76.94 | 2.90 | 20.71 |
| HPA threshold = %60 | 50.81 | 3.54 | 16.11 |
| HPA threshold = %70 | 54.43 | 3.14 | 34.61 |
| HPA threshold = %80 | 64.70 | 3.18 | 37.54 |

### 2.2. Networking Related Studies

In their study titled "Deep Learning-Based Traffic Prediction for Network Optimization", Troia S. et al. utilized deep learning techniques to predict traffic and optimize the resource management of backbone networks. They implemented the Gated Recurrent Units (GRU) method, a type of Recurrent Neural Network (RNN), for traffic prediction. Through comparison of numerical results of static and dynamic allocation based on estimates, they were able to effectively manage unexpected traffic flow and save 66.3% of available capacity in the network.

### 2.3. Cloud Environment Related Studies

In their study, Khaleq et al. present an intelligent, autonomous auto scaling system for microservices that can scale in the cloud while taking into account quality of service (QoS) constraints. They note the increasing complexity of cloud applications, which often have diverse resource needs and require a customized scaling approach to meet QoS requirements. They propose a two-module system that uses a general auto scaling algorithm to identify the resource demand of microservices and a reinforcement learning module to learn and define autoscaling thresholds based on resource demand and QoS. The proposed system demonstrates an improvement in microservice response time of up to 20% compared to the default autoscaling paradigm.

In their study, Balla et al. present Libra, an adaptive autoscaler for Kubernetes pods that can automatically detect the most suitable resources for a single pod and enable the scaling process. When the payload or environment changes, Libra revises the resource definition for the pod and scales to meet the new requirements. By testing Libra on a sample web application, the authors found that it outperforms the Horizontal Pod Autoscaler (HPA) and the autoscaling service of the open-source serverless tool OpenFaaS. The proposed system aims to improve resource management and efficiency in Kubernetes environments.

In their study, Nascimento et al. explored the use of various machine

learning techniques for scaling virtual clusters in cloud computing environments. They compared the performance of these techniques with traditional methods and found that the new techniques were more efficient, particularly in situations where workloads fluctuated over time. The authors also applied several methods, including Best-Performing Allocation, Probabilistic Best-Performing Allocation, Tunable Allocation, Adaptive Allocation, and Slice Training Data, to evaluate their effectiveness in scaling virtual clusters. Overall, the results of their study suggest that machine learning techniques can be useful for optimizing resource allocation in cloud computing environments.

In their study entitled "AI-based Resource Allocation: Reinforcement Learning for Adaptive Auto-scaling in Serverless Environments," Lucia Schuler, Somaya Jamil, and Niklas Kuh explore the use of multiple processors with various machine learning applications in architecture. Through their experiments, they found that the Q-Learning based model achieved more than 80% of the performance of the default setting in serverless systems, particularly when traffic forecasting is applied. Their research, published in 2020, demonstrates the effectiveness of this approach in optimizing resource allocation in serverless environments.

In their study entitled "An Autoscaling Framework for Cloud Computing Environments Based on Machine Learning," Gao et al. proposed a machine learning-based auto scaling framework for cloud computing environments. The proposed framework utilizes a combination of two machine learning algorithms, the gradient boosting decision tree (GBDT) and the multilayer perceptron (MLP), to accurately predict resource demand and optimize resource allocation in real-time. The authors performed experiments on a cloud computing testbed and found that the proposed framework was able to significantly improve resource utilization and reduce response time compared to traditional auto scaling approaches. In addition, the authors demonstrated that the proposed framework was able to adapt to changes in workload patterns and resource demand, making it a promising solution for dynamic and highly variable cloud computing environments.

### 2.4. Other Studies

In their study, Gong, Gu, and Wilkes propose a new method called Predictive Elastic Resource Scaling (PRESS) for cloud systems, which aims to minimize provisioning costs while meeting service level objectives (SLOs). PRESS is an automated system that adjusts resource allocations based on online estimates of dynamic application resource requirements, which are obtained using lightweight signal processing and statistical learning algorithms. The PRESS system was implemented on the Xen virtualization platform and tested using RUBiS and a Google application load monitoring service. The results of the experiments showed that PRESS was able to achieve resource forecasting accuracy with less than 5% forecasting error and that flexible resource scaling can significantly reduce both resource waste and SLO violations.

In their study, Hamid Arabnejad et al. compare two machine learning strategies based on a fuzzy logic system that learns and modifies fuzzy scaling rules at runtime: Fuzzy SARSA learning (FSL) and Fuzzy Q-learning (FQL). These strategies are designed to optimize resource allocation in cloud environments by adapting to changes in workload and ensuring that service level agreements (SLAs) are met. The FSL approach takes into account the behavior of the real agent and leads to faster learning, while the FQL approach learns independently. The experiments were conducted on the OpenStack cloud platform and the results showed that both auto scaling approaches were effective in various load traffic situations, both sudden and periodic. It was also found that both approaches had acceptable performance in terms of the number of virtual machines and SLA compliance.

# 3. OVERVIEW OF METHODS

Kubernetes was chosen as the container orchestration method for the cloud side of the project. In order to monitor and observe the system, Prometheus and grafana were also utilized. Load testing was conducted using Apache jmeter, and the xgboost method was employed for the machine learning portion of the project.

## 3.1. Kubernetes

Kubernetes is a container orchestration tool developed by Google that has gained widespread adoption with the increasing use of cloud technologies. In the past, virtual machines were the primary means of utilizing cloud resources, but containers have now become the industry standard. Container orchestration tools like Kubernetes enable easier management, debugging, versioning, and portability of applications. The basis for Kubernetes is the concept of containerization, which was introduced with Linux Containers (LXC) in 2008 and added to the Linux kernel. These container technologies provide a new approach to issues such as inter-container isolation, communication, information sharing, interruption management, and upgrading. Kubernetes simplifies the management of these containers in a cloud environment by providing a single control plane. In addition, Kubernetes offers the ability to verify the proper functioning of applications, revert to previous versions in case of failure, use and manage preferred storage systems, scale applications automatically or manually, implement self-healing, control incoming traffic to the system from a single point, and establish connections between containers. It also has the capability to terminate unhealthy running containers and restart or reschedule failed containers as needed based on the need for auto-scaling. The increasing volume of data and services in the cloud environment has made it more challenging to manage servers and has created new requirements. Kubernetes was developed to address these challenges and facilitate the automation of complex and costly operations such as scaling, rollback, auto-deployment, and service discovery through

a single control plane.

In this study, the Kubernetes Container Orchestration Platform was utilized as the platform for system design. The application instances were deployed to this platform as deployment objects and exposed using the Ingress method, allowing the services to be directly accessible through this Ingress. Additionally, the metrics of the applications were collected and visualized from the same platform. The infrastructure used was the Azure PaaS platform. A general overview of the Kubernetes architecture can be seen in Figure 3.1.

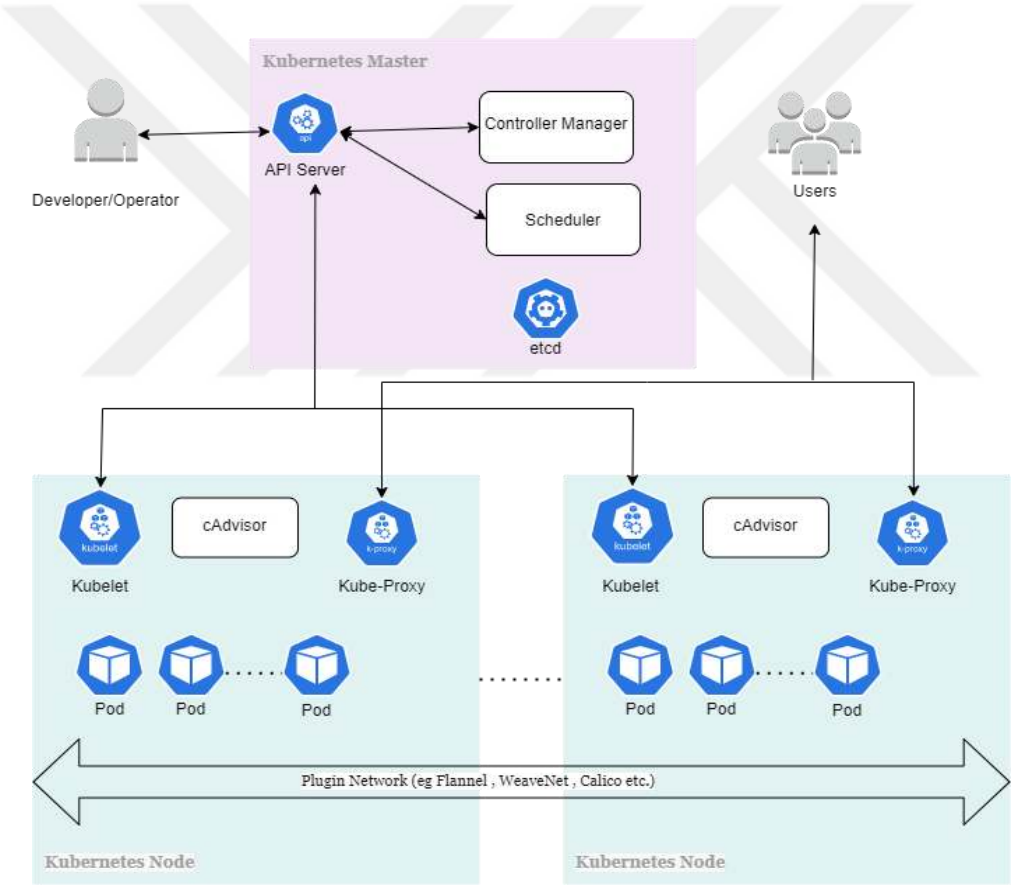


Figure 3.1. Kubernetes Architecture Overview

Upon examination of the general structure of Kubernetes, one will observe

the presence of master and worker nodes. The master node is the server responsible for managing all other nodes, while the worker nodes host the containers and manage them according to directives from the master node. Within the worker nodes are pods, which contain the containers. These containers communicate with each other via an overlay network. In addition to these features, Kubernetes offers capabilities such as increased agility in application development, the ability to easily create different environments for the same application, and easier management of microservices to implement DevOps applications such as continuous integration/continuous delivery (CI/CD). As Kubernetes is portable, it can be used in hybrid, cloud, on-premises, or multi-cloud ecosystems.

The master node in Kubernetes comprises four basic structures: the API Server, the Controller Manager, the Scheduler, and Etcd. The API Server receives all REST requests made to the master node and plans the management of the system according to these requests. These requests are usually written in the YAML configuration language and forwarded to the API Server by the client (administrator). The Controller Manager, acting as a controller, monitors the status of the entire system and ensures that system components are in the desired state using the reconciler mechanism when necessary. The Scheduler component creates a deployment plan for a workload in the system and determines on which worker nodes the applications and application replicas will run. Etcd is a distributed, consistent, and traceable key-value store (nosql database) that stores state information of system components.

The structure of worker nodes in Kubernetes consists of four basic building blocks: Kubelet, Kube-Proxy, Pod, and Container Engine. Kubelet is a Kubernetes agent running on the worker node that receives requests from the API Server and executes them on the worker node. Kube-Proxy handles network forwarding in Kubernetes and provides load-balancing for all pods of a service. The Container Engine is responsible for basic container management, including pulling container images from the registry and starting and stopping containers.

### 3.1.1. Pods

A Kubernetes pod is the smallest and most basic unit for managing containers. It typically consists of one or multiple containers running within it and shares resources such as the network, IP address, hostname, etc., and can communicate with other pods deployed on nodes. However, pods are considered temporary workloads and, if deleted or if they fail, will be lost permanently. Pods are managed by the master node, as illustrated in Figure 3.2.

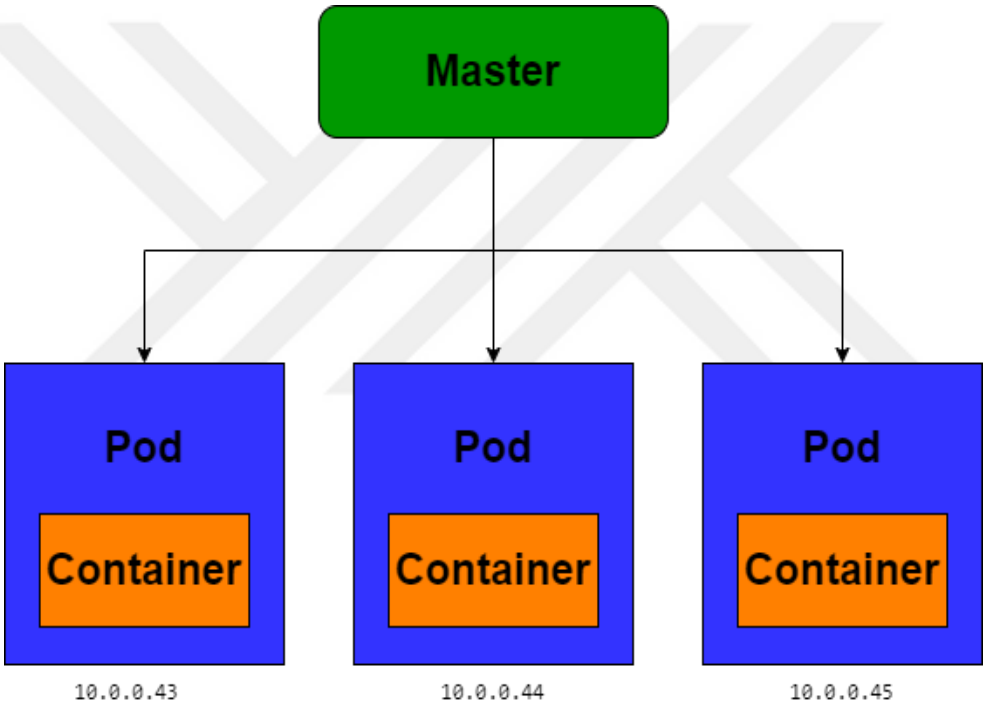


Figure 3.2. Kubernetes Pod Management

Pods can be thought of as the workspace for containers, and more than one container can run within a pod. Kubernetes manages pods as immutable, meaning that when a deployment request is made, a new version of the pod is created first and, once the new version is functioning properly, the old version is shut down. This process is illustrated in Figure 3.3. and is accomplished by applying a YAML

configuration to the pod.

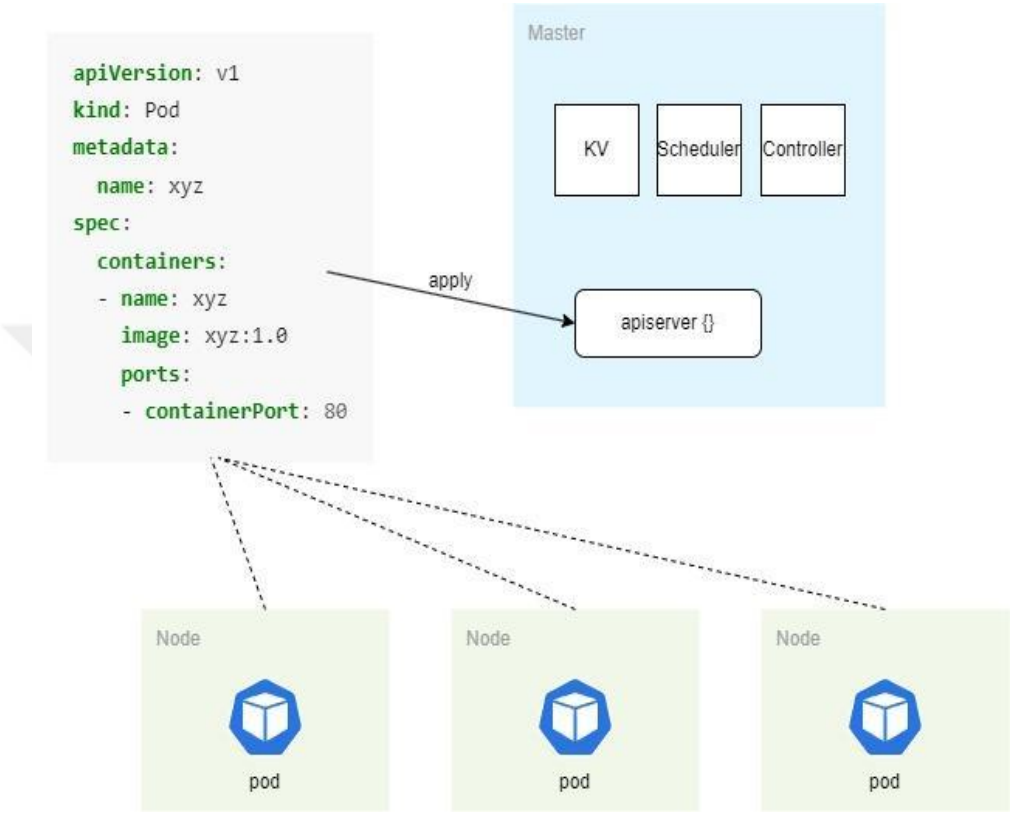


Figure 3.3. Kubernetes Change Pod Specifications

In this study, system metrics were collected based on pods and the comparison was made using these pod metrics. To do this, data was gathered from the different pods within the system and analyzed to determine any potential differences or trends. This data was then used to evaluate the performance of the system and identify any areas in need of improvement. The use of pod-based metrics allowed for a more granular analysis of the system and provided a more detailed understanding of its behavior and performance. Additionally, this approach allowed for a more comprehensive evaluation of the system, as it took into account all of the individual components within it rather than just considering the system as a whole.

Overall, the use of pod-based metrics in this study proved to be a useful and effective way to assess the performance of the system.

### 3.1.2. Cronjobs

Cronjobs are workloads that specify actions to be run at regular intervals. They are designed to run containers, and thus pods, to easily execute scheduled tasks within the Kubernetes environment. Cronjobs are a useful tool for automating tasks and processes, as they allow users to define and schedule specific actions to be taken at predetermined intervals. This can be particularly useful in cases where certain tasks or processes need to be run on a regular basis, such as data backups or system updates. Cronjobs are defined using the YAML configuration language and are managed by the master node. They can be created, modified, and deleted as needed, providing a flexible and efficient way to automate tasks within the Kubernetes environment.

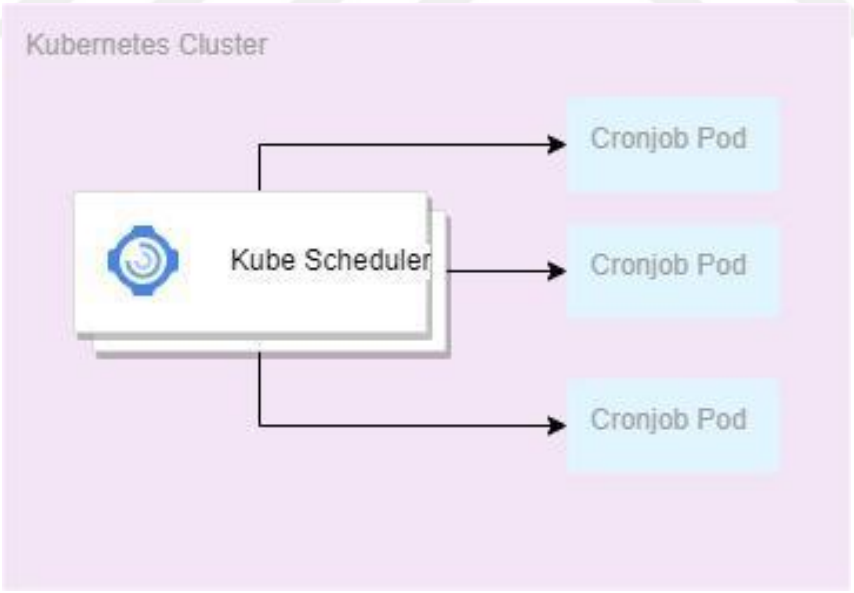


Figure 3.4. Kubernetes Cronjob Pods

Cronjobs are primarily intended to run pods on a one-time basis. As shown in Figure 3.4, the time interval at which the kube-scheduler should run is specified using configurations. When the designated time arrives, a one-time pod is run to perform the scheduled tasks. Cronjobs provide a convenient way to automate tasks and processes within the Kubernetes environment, allowing users to define and schedule specific actions to be taken at predetermined intervals. This can be particularly useful for tasks that need to be run on a regular basis, such as data backups or system updates. Cronjobs are defined using the YAML configuration language and are managed by the master node, providing flexibility and efficiency for automating tasks within the Kubernetes environment.

### 3.1.3. Ingresses

Ingress is a Kubernetes controller that allows end users to access multiple applications through a single load balancer. When an Ingress Controller is created within a cluster, it is put into service with a load balancer to provide public access. This allows multiple applications to be served using a single load balancer rather than multiple load balancers, increasing efficiency and convenience for the end user. Ingress controllers can be configured using the YAML configuration language and are managed by the master node. They provide a convenient way to access multiple applications through a single load balancer, making it easier for users to access and use these applications. Overall, Ingress controllers are a valuable tool for managing access to multiple applications in a Kubernetes environment.

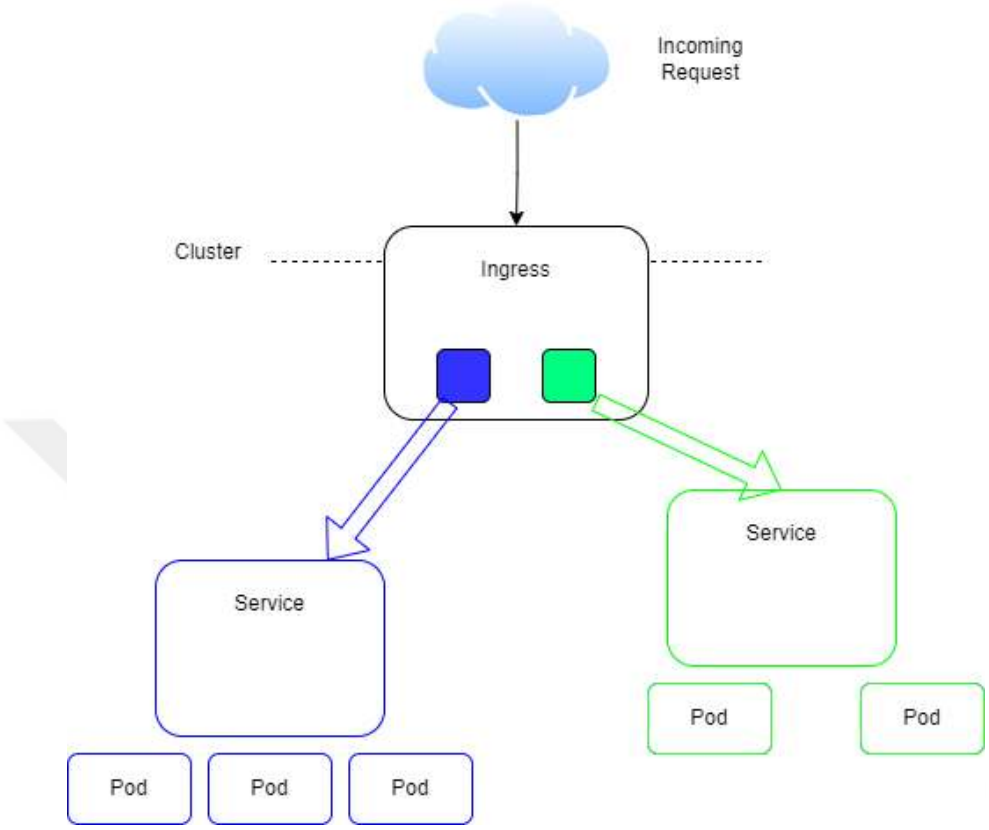


Figure 3.5. Ingress Overview

Services defined as Ingress direct incoming traffic from the external world to the appropriate pods based on rules determined within the system. This can be seen more closely in Figure 3.5. Ingress services act as a layer of abstraction, allowing users to access applications and services within the Kubernetes environment without having to know the specific details of the underlying infrastructure. They provide a convenient way to access and use applications and services within the cluster, making it easier for users to interact with the system. Ingress services can be configured using the YAML configuration language and are managed by the master node. They provide a valuable tool for managing access to applications and services within a Kubernetes environment and improving the user

experience.

### 3.2. Observability

Observability refers to the ability to gather information about the internal workings of complex systems and to use this information to identify and solve problems more quickly and accurately. It is often confused with monitoring solutions and application performance management, but observability goes beyond these approaches by transforming the data collected about applications into more comprehensive and actionable insights. This is particularly important in the cloud environment, where applications are becoming faster and more dynamic. Figure 3.6 provides a more detailed overview of the observability and metric architecture. In general, the more observable a system is, the faster and more accurately we can identify and resolve issues within it. Observability is a key component of effective system management and can significantly improve the reliability and performance of applications.

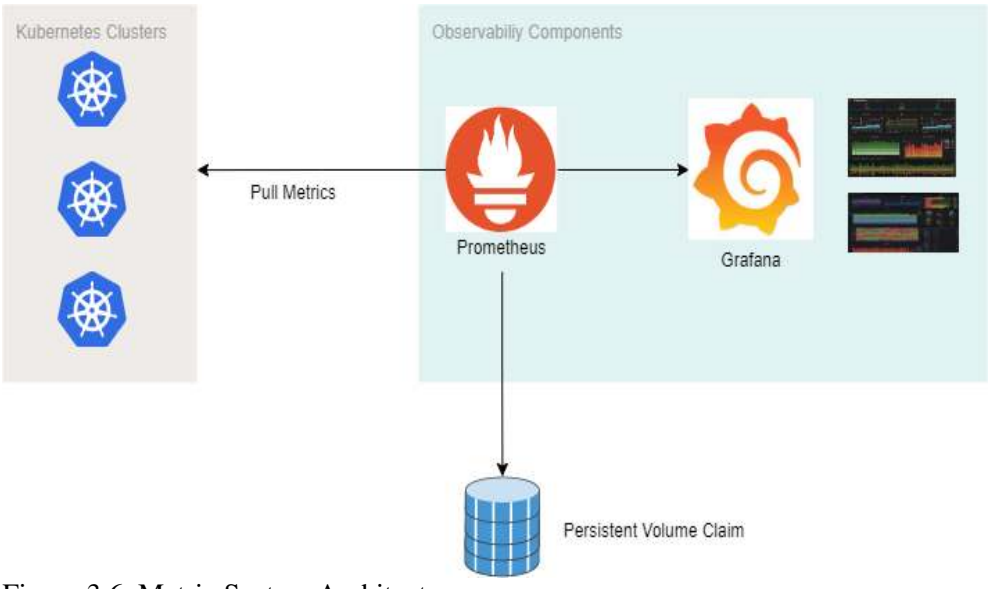


Figure 3.6. Metric System Architecture

### 3.2.1. Metric Collection

To effectively manage a Kubernetes cluster in a production environment, software to track metrics is typically used. These metrics are collected and stored in a central location, making it easier to monitor and manage the system. Based on these metrics, various actions can be taken within the cloud environment to optimize performance and ensure reliability. In this study, Prometheus, a popular metric server, was used to track and store metrics related to applications and systems in a time series. The use of a metric server such as Prometheus allows for more efficient and effective management of a Kubernetes cluster in a production environment, as it provides a centralized location for storing and analyzing metrics. This can help to identify issues and take corrective action more quickly, improving the overall performance and reliability of the system.

### 3.2.2. Visualization

Data visualization systems are applications that connect various data sources and allow users to view and interpret data in the form of charts and graphs more easily. Grafana is an open source data visualization web application that enables users to obtain data from a variety of sources and combine it in a single dashboard for visualization purposes. Grafana provides a convenient way to view and interpret data from multiple sources, making it easier to gain insights and understand trends. Data visualization is an important tool for understanding and analyzing complex data sets, and Grafana provides a user-friendly platform for visualizing data in a way that is easy to interpret. Overall, data visualization systems such as Grafana can be valuable tools for improving the interpretability and usability of complex data sets.

### 3.2.3. Load Test

Performance testing is a type of software testing that aims to measure the performance of software applications under a specified workload. The primary goal of performance testing is to identify performance bottlenecks and to evaluate the

scalability, reliability, and resource usage of a system under different conditions. In this study, performance testing was conducted using a tool called Apache JMeter, which allows users to create sample HTTP requests to simulate user behavior. By performing performance tests, it is possible to understand how a system responds to different workloads and to identify any issues that may affect the performance of the system. Overall, performance testing is a valuable tool for ensuring that software applications can meet the demands of their intended use and for identifying and addressing any potential performance issues.

### 3.2.4. Machine Learning

In this study, a literature review was conducted to identify the most commonly used regression algorithms for similar studies. Based on this review, the ARIMA, XGBoost, Long Short-Term Memory (LSTM), Multi-Layer Perceptron (MLP), and Hierarchical Temporal Memory (HTM) algorithms were selected as candidate algorithms.

ARIMA (Automatic Regressive Integrated Moving Average) is a statistical model used for time series estimation. A model class that catches errors in datasets of the time series data type by applying a regression-like equation to the data and taking into account data errors. ARIMA models are used in fields such as economics, finance, and engineering to make predictions about future trends and patterns in data. The ARIMA model consists of three components: the autoregression (AR) component, the difference component (I for "integrated"), and the moving average (MA) component. The AR component captures the autocorrelation between observations at different delays, the I component explains the non-stationary data by taking the differences of the data, and the MA component models the error term in the data. The parameters of the ARIMA model are determined by estimation techniques such as maximum likelihood estimation or Bayesian estimation. ARIMA models can be used to predict future values using predictive parameters and historical data.

Long Short Term Memory (LSTM) is a type of recurrent neural network (RNN) well suited for processing sequential data. LSTMs handle long data sequences and aim to solve one of the biggest problems in traditional RNNs, the disappearing gradients. This makes LSTMs particularly useful for tasks that require the model to preserve long-term memory, such as language modeling and sensitivity analysis. LSTMs have a slightly different structure than traditional RNNs, with additional components called gates that control the flow of information over the network. There are three types of gates in an LSTM unit: entry gate, forget gate and exit gate. These gates work together to decide what information should be kept or forgotten and which should be output, allowing the LSTM to retain information for longer. The input gate determines what new information should be added to the cell state, the forget gate determines what information should be forgotten from the previous cell state, and the output gate determines what information should be removed from the LSTM unit. The cell state is a vector that is updated at each time step based on information from the input, previous cell state, and gates. The output of the LSTM unit is based on the current cell state and is used as the input for the next time step. LSTMs are trained using supervised learning; where the model is trained on a labeled dataset to make predictions about future time steps based on previous information. The parameters of the LSTM are updated during training, so the predictive values are quite close to the actual outputs. LSTMs have been successfully applied to many different tasks, including language modeling, speech recognition, machine translation, and time series prediction. It has also been used for more complex tasks such as video analytics, captioning, and stock market forecasting. Because of their ability to process long data sequences and preserve long-term memory, LSTMs are a popular choice for solving many real-world problems.

Multi-Layer Perceptron (MLP) is a type of artificial neural network commonly used for supervised learning tasks such as classification and regression. An MLP consists of many interconnected layers of nodes or neurons, where each

layer processes input from the previous layer and transmits the results to the next layer. The last layer produces the output of the network, which is used to make predictions about the inputs. The basic building block of an MLP is the artificial neuron that takes multiple inputs, processes them using a set of weights, and produces a single output. The operation done by the neuron can be represented by the following equation:

$$y = f(b + \Sigma_{i=1}^{n} w_i * x_i)$$

where:

$x_i$ is the i-th input
$w_i$ is the weight associated with the i-th input
$b$ is the bias term
$f$ is the activation function
$y$ is the output of the neuron

The activation function is used to impart nonlinearity to the network and allows modeling complex relationships between inputs and outputs. Common activation functions include sigmoid, tanh, and corrected linear unit (ReLU). An MLP is trained using supervised learning, where the model is trained on a labeled dataset to minimize the difference between predicted outcomes and actual outcomes. This is typically done using gradient descent, where the weights and biases of the network are updated in the direction of the negative gradient of the loss function. The loss function measures the difference between the predicted outputs and the actual outputs and is used to evaluate the performance of the model. Once trained, an MLP can be used to make predictions on new, unseen data. MLPs are widely used for many different tasks, including image classification, speech recognition, and natural language processing. MLPs are a valuable tool for solving many real-world

problems.

Hierarchical Temporary Memory (HTM) is a biology-inspired machine learning algorithm. It is designed to process and make predictions about flow data such as time series data by recognizing patterns and making inferences based on these patterns. HTMs are based on the theory of the neocortex, the part of the brain responsible for sensory processing, perception and cognition. An HTM consists of a large number of nodes or neurons, each responsible for processing a different scale of temporal information. Lower levels of the network process fine-scale temporal information such as individual increases or changes in data, while higher levels process coarser-scale temporal information such as patterns and trends. HTMs use unsupervised learning, where the model is trained on the data without labeled examples to discover and learn the underlying structure of the data. One of the key components of HTMs is the spatial pooler, which is responsible for choosing a small set of active neurons to represent each input. The spatial pooler uses a combination of inhibition and amplification to ensure that only the most active neurons are selected while suppressing less active neurons. This allows HTM to represent each input with a compact and robust set of active neurons while reducing the dimensionality of the data. Another important component of HTMs is volatile memory, which is responsible for making predictions about the future based on current and past inputs. Temporal memory uses a series of lateral connections between neurons to store and retrieve strings of information, allowing data to maintain a persistent memory over time. Predictions made by temporal memory are used to update the spatial pooler and allow HTM to continuously improve its data representation. HTMs have been applied to a variety of tasks, including anomaly detection, prediction, and classification. It has been used for applications such as stock price prediction, sensor data analysis, and speech recognition. Because of their ability to process streaming data and make predictions, HTMs are a valuable tool for solving many real-world problems.

XGBoost (eXtreme Gradient Boost) is a machine learning algorithm that has

been optimized through the incorporation of various modifications to the Gradient Boost method. Among the most notable features of XGBoost are its ability to enhance prediction accuracy, prevent overfitting, handle missing or incomplete data, and operate efficiently. The initial step in the XGBoost process involves setting a base score, which can also be generated randomly as the algorithm endeavors to identify the correct outcome for the transactions in subsequent steps. By default, this base score is typically set to 0.5. The model's predictive accuracy is then evaluated in relation to the number of incorrect predictions made, with error values calculated as the difference between observed and predicted values. A decision tree is subsequently constructed to predict errors and a similarity score is computed for the tree's branches, with the similarity score indicating the degree to which the data was effectively divided into these branches. Subsequent to the calculation of similarity scores, trees encompassing all possible probabilities are created in order to determine whether a more accurate prediction can be made, with similarity scores calculated for all of these trees. To determine which of these trees is optimal, it is also necessary to calculate the gain, which is derived from the evaluation of the entire tree using the similarity score for its branches.

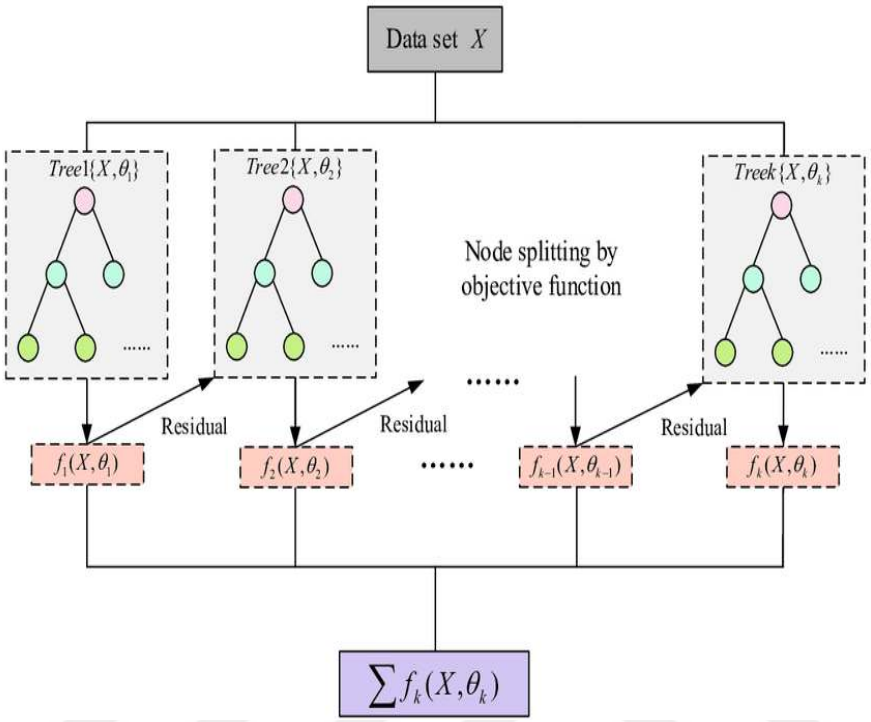


Figure 3.7. XGBoost Decision Trees

In Figure 3.7, it is possible to examine in greater detail the manner in which XgBoost functions and the formula underlying this process. This figure illustrates that a large number of trees will be generated, and the best tree will be selected. Once the tree to be used has been determined based on the gain value, the pruning process begins. During pruning, a "gamma" value, which represents an evaluation of the gain score, is chosen, and branches with a gain score lower than the gamma score are trimmed. Pruning proceeds from the bottom up. The default gamma value is 0. A gamma value of 0 indicates that if the gain score is negative, the corresponding branches will be pruned. If all branches are pruned, the initial default guess of 0.5 is used

The selection process took into account not only the margin of error of each algorithm, but also its resource usage and completion time. Since the aim of the study

was to optimize resource efficiency, it was important to consider the resource utilization of each machine learning algorithm. Additionally, the completion time of each algorithm was also important, as it would be operating within certain time periods. As shown in Table 3.7, the ARIMA algorithm had a 17% mean percentage error (MPE), a RAM usage of 122 MB, a CPU usage of 32%, and a completion time of 17 minutes. The XGBoost algorithm had a 22% MPE, a RAM usage of 60 MB, a CPU usage of 23%, and a completion time of 11 minutes. The LSTM algorithm had a 16% MPE, a RAM usage of 101 MB, a CPU usage of 29%, and a completion time of 15 minutes. The MLP algorithm had a 15% MPE, a RAM usage of 95 MB, a CPU usage of 37%, and a completion time of 22 minutes. The HTM algorithm had a 20% MPE, a RAM usage of 82 MB, a CPU usage of 30%, and a completion time of 19 minutes. Based on these results, the XGBoost algorithm was determined to be the most suitable and efficient algorithm for this study.

Table 3.1. Machine Learning Methods Comparison

| **Machine Learning Algorithm** | **Mean Percentage Error(%)** | **Memory Usage(Mb)** | **CPU Usage Percentage(%)** | **Completion Time(min)** |
|---|---|---|---|---|
| ARIMA | 17 | 122 | 32 | 17 |
| XGBOOST | 22 | 60 | 23 | 11 |
| Long Short Time Memory | 16 | 101 | 29 | 15 |
| Multilayer Perceptron | 15 | 95 | 37 | 22 |
| Hierarchical Temporal Memory | 20 | 82 | 30 | 19 |

# 4. DEVELOPMENT

This section comprises four subsections. The first subsection elaborates on the development of Kubernetes operators. The second subsection introduces the deployment of a fanout proxy. The third subsection presents a sample application. The fourth subsection includes screenshots. Additionally, the purpose of this section is to explicate the process of developing Kubernetes operators and deploying applications using them. This information can be useful during the deployment of Kubernetes-based applications and, therefore, the content of this section is of significance. It is worth noting that the deployment of a fanout proxy is an integral part of this process, as it enables the efficient distribution of incoming requests to multiple replicas of an application. Similarly, the inclusion of a sample application in this section allows for a practical understanding of the concepts discussed. Overall, this section aims to provide a comprehensive overview of the development and deployment of Kubernetes operators.

## 4.1. Kubernetes Operator

Kubernetes is designed for the automation and orchestration of containers. Through the extension of Kubernetes, it is possible to enhance the functionality beyond the core features of Kubernetes. The operator approach utilized in Kubernetes enables the extension of the behavior of the cluster without modifying the Kubernetes code. This is achieved by allowing Kubernetes structures to be managed and controlled through the implementation of written operators. Operators are Kubernetes API clients that serve as controllers and monitors for a Custom Resource Definition (CRD) that they create. In this way, the operator method allows for the customization and management of Kubernetes functionality through the creation of CRDs and the implementation of operators to handle them. The general operator software structure can be viewed in Figure 4.1.

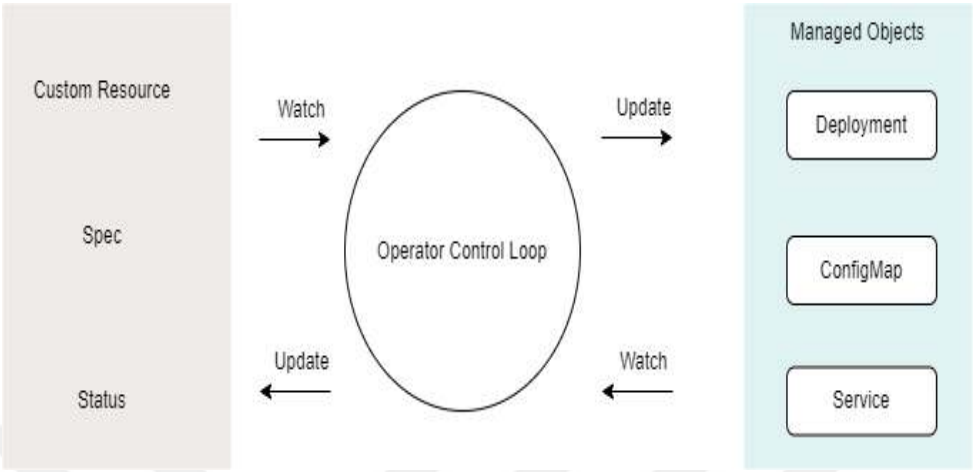


Figure 4.1. Kubernetes Operator Structure

In the course of this study, a Kubernetes operator written in the Python programming language was developed. This operator consists of two parts. The first part is implemented as a cronjob that runs on a daily basis. The cronjob performs a parameter optimization technique known as Grid Search. This cronjob helps PHPA to consistently operate with the most optimal parameters by organizing the hyperparameters in a database shared with PHPA, based on the data it retrieves from EPIAS. The second part of the operator consists of a set of functions that handle the management and execution of the cronjob. These functions ensure that the cronjob runs smoothly and can be easily modified or terminated if necessary. Overall, the development of this Kubernetes operator has allowed for the automated optimization of PHPA's parameters, leading to improved performance and efficiency.The general PHPA software structure can be viewed in Figure 4.2.

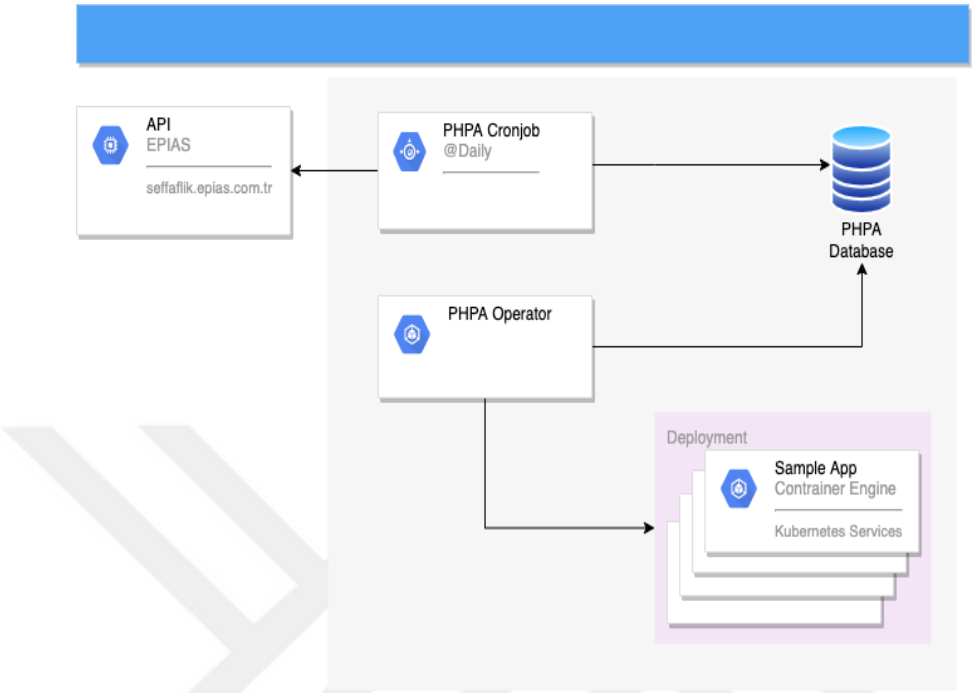


Figure 4.2. Kubernetes PHPA Structure

### 4.2. Fan Out Proxy

The fanout proxy is a tool that evenly and simultaneously distributes HTTP requests from a load test to all environments, enabling more robust testing. Essentially, it is a proxy written in the Go programming language that duplicates all incoming requests and forwards them to all environments where the experiment is being conducted. By using a fanout proxy, it is possible to ensure that the load test is being carried out in a balanced and controlled manner, without overwhelming any specific environment. This is particularly useful in situations where the load test involves a large number of requests or where the environments being tested have varying levels of capacity. Overall, the use of a fanout proxy can significantly improve the reliability and accuracy of load testing results.

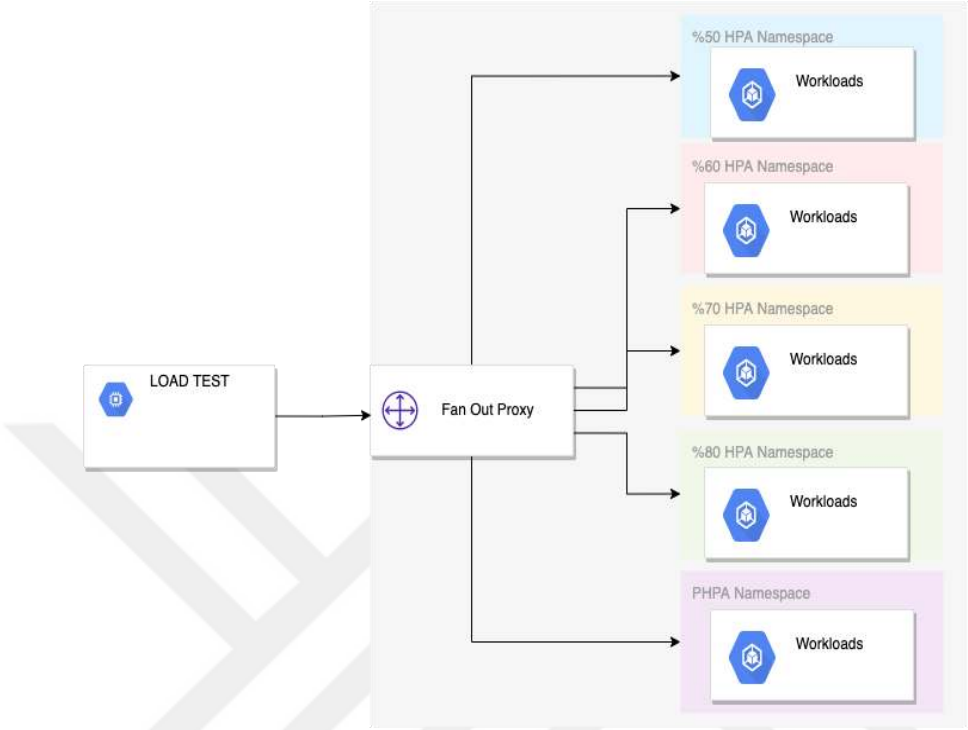


Figure 4.3. Fan Out Operation

### 4.3. Sample Application

The following is an example of a simple Go application that serves as a sample application with a few basic endpoints based on one of the microservices in the smartpulse ecosystem. This application has been written in the Go programming language and is well-suited for horizontal scaling, as it does not maintain any state. Additionally, it is worth noting that this sample application serves as a useful illustration of how Go can be used to implement microservices in a distributed system.

The application consists of a set of functions that handle HTTP requests and responses. It includes an endpoint for retrieving a list of items, an endpoint. One of the key advantages of this sample application is its simplicity and ease of use. It provides a clear and straightforward illustration of how Go can be used to implement

microservices, making it an excellent resource for learning and experimentation. Additionally, the lack of state in the application allows it to scale horizontally with minimal overhead, making it an efficient and effective solution for distributed systems.

### 4.4. Screenshots

This section contains examples of screenshots of the application and system. These screenshots demonstrate how the application and system operate and can be used. These screenshots support the concepts discussed in previous sections with practical examples and facilitate a better understanding of them. In addition, these screenshots show how the application and system can be used in real-life scenarios, making the content of this section highly relevant. This section includes examples that will help to better understand the operating principles of the application and system and facilitate the learning of how to use the application and system.

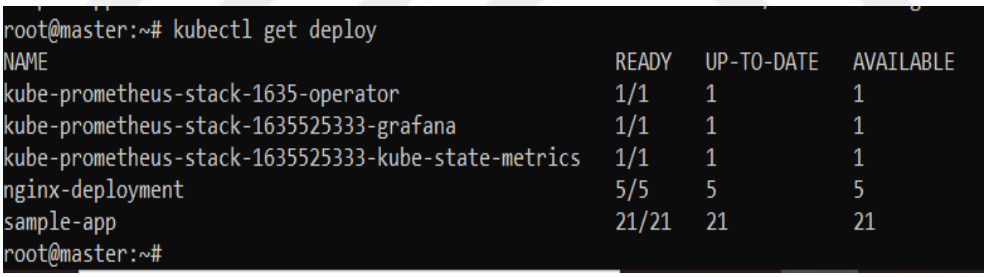


Figure 4.4. PHPA Deployment

```
root@master:~# kubectl get po
NAME                                                     READY   STATUS      RESTARTS
alertmanager-kube-prometheus-stack-1635-alertmanager-0   2/2     Running     20 (7d20h ago)
kube-prometheus-stack-1635-operator-8497df9fbc-8qgrf     1/1     Running     10 (7d20h ago)
kube-prometheus-stack-1635525333-grafana-6d4454dd65-b5268   2/2     Running     20 (74m ago)
kube-prometheus-stack-1635525333-kube-state-metrics-595484fklg4   1/1     Running     10 (7d20h ago)
kube-prometheus-stack-1635525333-prometheus-node-exporter-8qr7x   1/1     Running     10 (74m ago)
minio-0                                                  1/1     Running     9 (7d20h ago)
mlhpa-27523295--1-pckk2                                  0/1     Completed   0
mlhpa-man--1-d6wk4                                       0/1     Completed   0
nginx-deployment-66b6c48dd5-sznw2                        1/1     Running     0
prometheus-kube-prometheus-stack-1635-prometheus-0       2/2     Running     20 (74m ago)
sample-app-85694fc5c6-47ws8                              1/1     Running     3 (7d20h ago)
sample-app-85694fc5c6-7b2xb                              1/1     Running     0
sample-app-85694fc5c6-7mh44                              1/1     Running     0
sample-app-85694fc5c6-bqdkf                              1/1     Running     0
sample-app-85694fc5c6-cf86d                              1/1     Running     0
sample-app-85694fc5c6-crphz                              1/1     Running     0
sample-app-85694fc5c6-fbjbc                              1/1     Running     0
sample-app-85694fc5c6-ffktt                              1/1     Running     0
sample-app-85694fc5c6-gx2r7                              1/1     Running     0
sample-app-85694fc5c6-hplwv                              1/1     Running     0
sample-app-85694fc5c6-mht2m                              1/1     Running     0
sample-app-85694fc5c6-mqp6c                              1/1     Running     0
sample-app-85694fc5c6-n468v                              1/1     Running     0
sample-app-85694fc5c6-nmscr                              1/1     Running     0
sample-app-85694fc5c6-nrhwn                              1/1     Running     0
sample-app-85694fc5c6-sgftp                              1/1     Running     0
sample-app-85694fc5c6-x7j7p                              1/1     Running     0
sample-app-85694fc5c6-xwz69                              1/1     Running     0
```

Figure 4.5. PHPA Pods

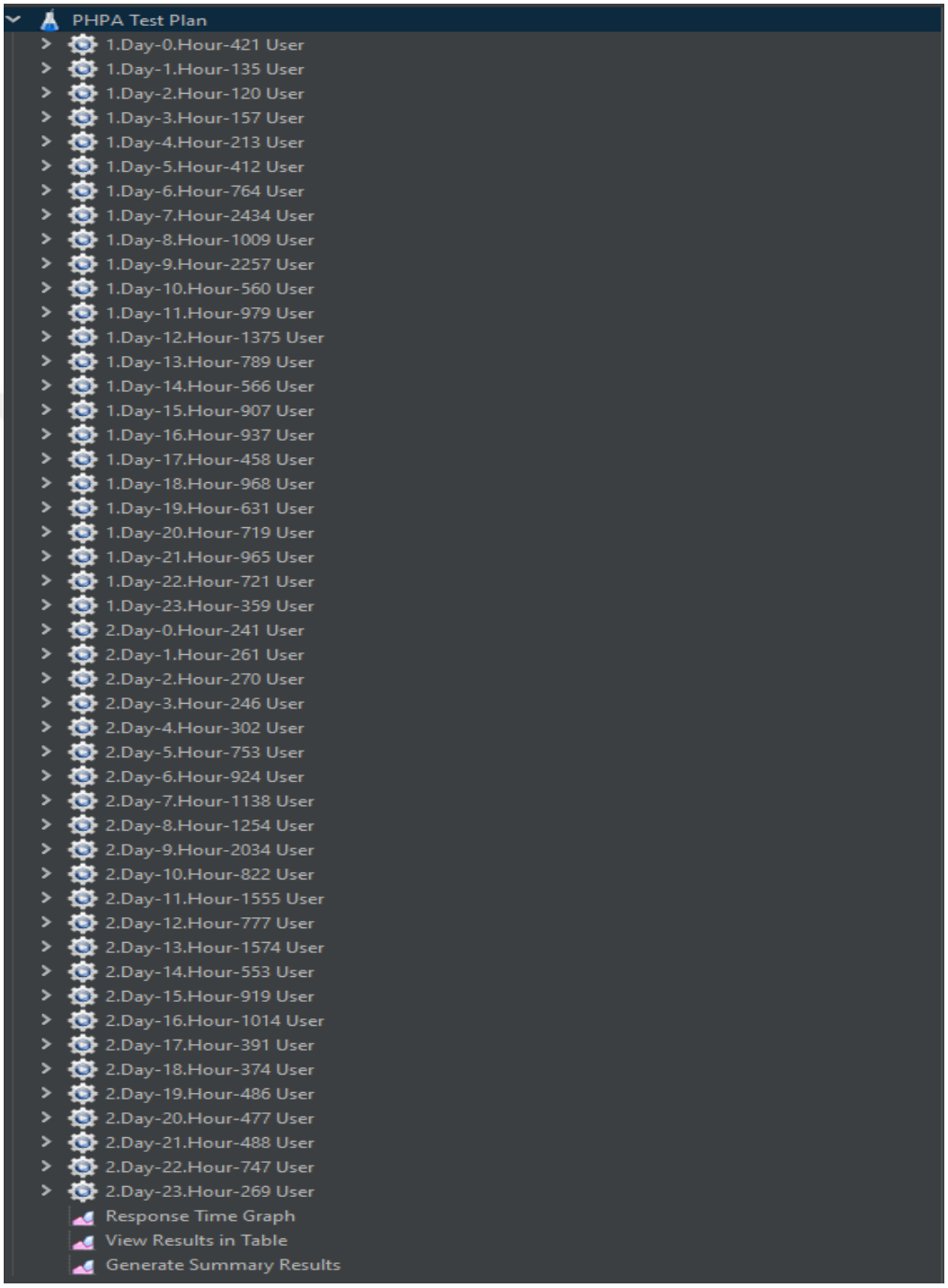

Figure 4.6. PHPA Test Plan

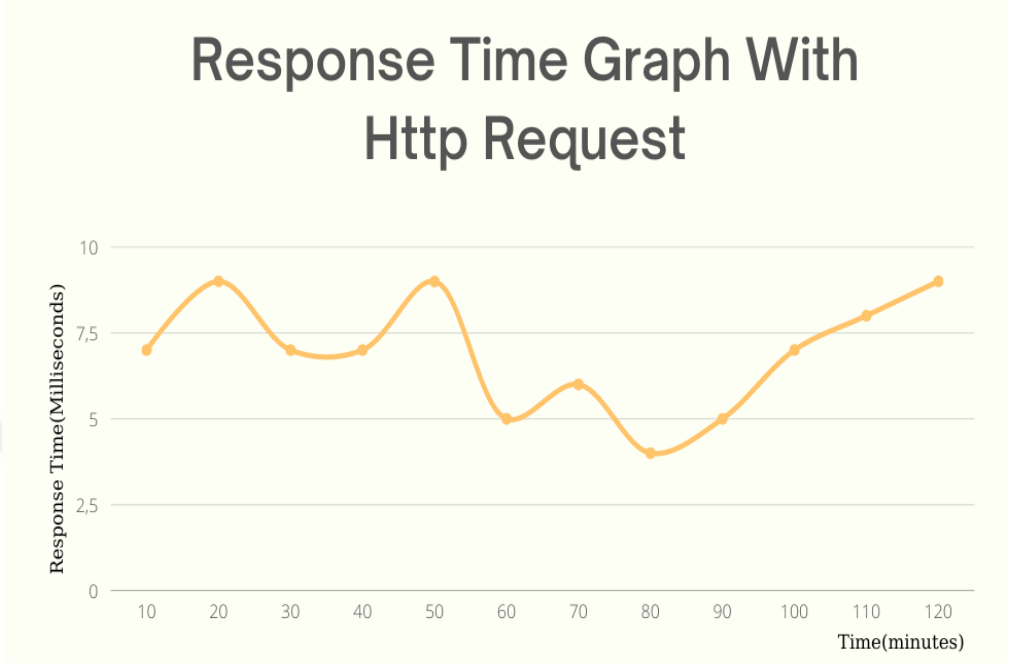


Figure 4.7. PHPA Response Time Graph

| Sample # | Start Time | Thread Name | Label | Sample Time(ms) | Status |
|---|---|---|---|---|---|
| 367205 | 15:54:35.526 | 2.Day-23.Hour-269 User 48-265 | HTTP Request | 1 | |
| 367206 | 15:54:35.527 | 2.Day-23.Hour-269 User 48-265 | HTTP Request | 0 | |
| 367207 | 15:54:35.527 | 2.Day-23.Hour-269 User 48-265 | HTTP Request | 1 | |
| 367208 | 15:54:35.528 | 2.Day-23.Hour-269 User 48-265 | HTTP Request | 0 | |
| 367209 | 15:54:35.528 | 2.Day-23.Hour-269 User 48-265 | HTTP Request | 0 | |
| 367210 | 15:54:35.528 | 2.Day-23.Hour-269 User 48-265 | HTTP Request | 1 | |
| 367211 | 15:54:35.970 | 2.Day-23.Hour-269 User 48-266 | HTTP Request | 0 | |
| 367212 | 15:54:35.971 | 2.Day-23.Hour-269 User 48-266 | HTTP Request | 0 | |
| 367213 | 15:54:35.971 | 2.Day-23.Hour-269 User 48-266 | HTTP Request | 0 | |
| 367214 | 15:54:35.971 | 2.Day-23.Hour-269 User 48-266 | HTTP Request | 0 | |
| 367215 | 15:54:35.971 | 2.Day-23.Hour-269 User 48-266 | HTTP Request | 1 | |
| 367216 | 15:54:35.972 | 2.Day-23.Hour-269 User 48-266 | HTTP Request | 0 | |
| 367217 | 15:54:35.972 | 2.Day-23.Hour-269 User 48-266 | HTTP Request | 0 | |
| 367218 | 15:54:35.972 | 2.Day-23.Hour-269 User 48-266 | HTTP Request | 0 | |
| 367219 | 15:54:35.972 | 2.Day-23.Hour-269 User 48-266 | HTTP Request | 1 | |
| 367220 | 15:54:35.973 | 2.Day-23.Hour-269 User 48-266 | HTTP Request | 0 | |
| 367221 | 15:54:36.416 | 2.Day-23.Hour-269 User 48-267 | HTTP Request | 1 | |
| 367222 | 15:54:36.417 | 2.Day-23.Hour-269 User 48-267 | HTTP Request | 0 | |
| 367223 | 15:54:36.417 | 2.Day-23.Hour-269 User 48-267 | HTTP Request | 1 | |
| 367224 | 15:54:36.418 | 2.Day-23.Hour-269 User 48-267 | HTTP Request | 0 | |
| 367225 | 15:54:36.418 | 2.Day-23.Hour-269 User 48-267 | HTTP Request | 0 | |
| 367226 | 15:54:36.418 | 2.Day-23.Hour-269 User 48-267 | HTTP Request | 0 | |
| 367227 | 15:54:36.418 | 2.Day-23.Hour-269 User 48-267 | HTTP Request | 1 | |
| 367228 | 15:54:36.419 | 2.Day-23.Hour-269 User 48-267 | HTTP Request | 0 | |
| 367229 | 15:54:36.419 | 2.Day-23.Hour-269 User 48-267 | HTTP Request | 0 | |
| 367230 | 15:54:36.419 | 2.Day-23.Hour-269 User 48-267 | HTTP Request | 0 | |
| 367231 | 15:54:36.862 | 2.Day-23.Hour-269 User 48-268 | HTTP Request | 1 | |
| 367232 | 15:54:36.863 | 2.Day-23.Hour-269 User 48-268 | HTTP Request | 0 | |
| 367233 | 15:54:36.863 | 2.Day-23.Hour-269 User 48-268 | HTTP Request | 1 | |
| 367234 | 15:54:36.864 | 2.Day-23.Hour-269 User 48-268 | HTTP Request | 0 | |
| 367235 | 15:54:36.864 | 2.Day-23.Hour-269 User 48-268 | HTTP Request | 0 | |
| 367236 | 15:54:36.864 | 2.Day-23.Hour-269 User 48-268 | HTTP Request | 0 | |
| 367237 | 15:54:36.864 | 2.Day-23.Hour-269 User 48-268 | HTTP Request | 1 | |
| 367238 | 15:54:36.865 | 2.Day-23.Hour-269 User 48-268 | HTTP Request | 0 | |
| 367239 | 15:54:36.865 | 2.Day-23.Hour-269 User 48-268 | HTTP Request | 0 | |
| 367240 | 15:54:36.865 | 2.Day-23.Hour-269 User 48-268 | HTTP Request | 0 | |
| 367241 | 15:54:37.308 | 2.Day-23.Hour-269 User 48-269 | HTTP Request | 0 | |
| 367242 | 15:54:37.308 | 2.Day-23.Hour-269 User 48-269 | HTTP Request | 1 | |
| 367243 | 15:54:37.309 | 2.Day-23.Hour-269 User 48-269 | HTTP Request | 0 | |
| 367244 | 15:54:37.309 | 2.Day-23.Hour-269 User 48-269 | HTTP Request | 0 | |
| 367245 | 15:54:37.309 | 2.Day-23.Hour-269 User 48-269 | HTTP Request | 1 | |
| 367246 | 15:54:37.310 | 2.Day-23.Hour-269 User 48-269 | HTTP Request | 0 | |
| 367247 | 15:54:37.310 | 2.Day-23.Hour-269 User 48-269 | HTTP Request | 0 | |
| 367248 | 15:54:37.310 | 2.Day-23.Hour-269 User 48-269 | HTTP Request | 0 | |
| 367249 | 15:54:37.310 | 2.Day-23.Hour-269 User 48-269 | HTTP Request | 1 | |
| 367250 | 15:54:37.311 | 2.Day-23.Hour-269 User 48-269 | HTTP Request | 0 | |

Figure 4.8. PHPA Result Table

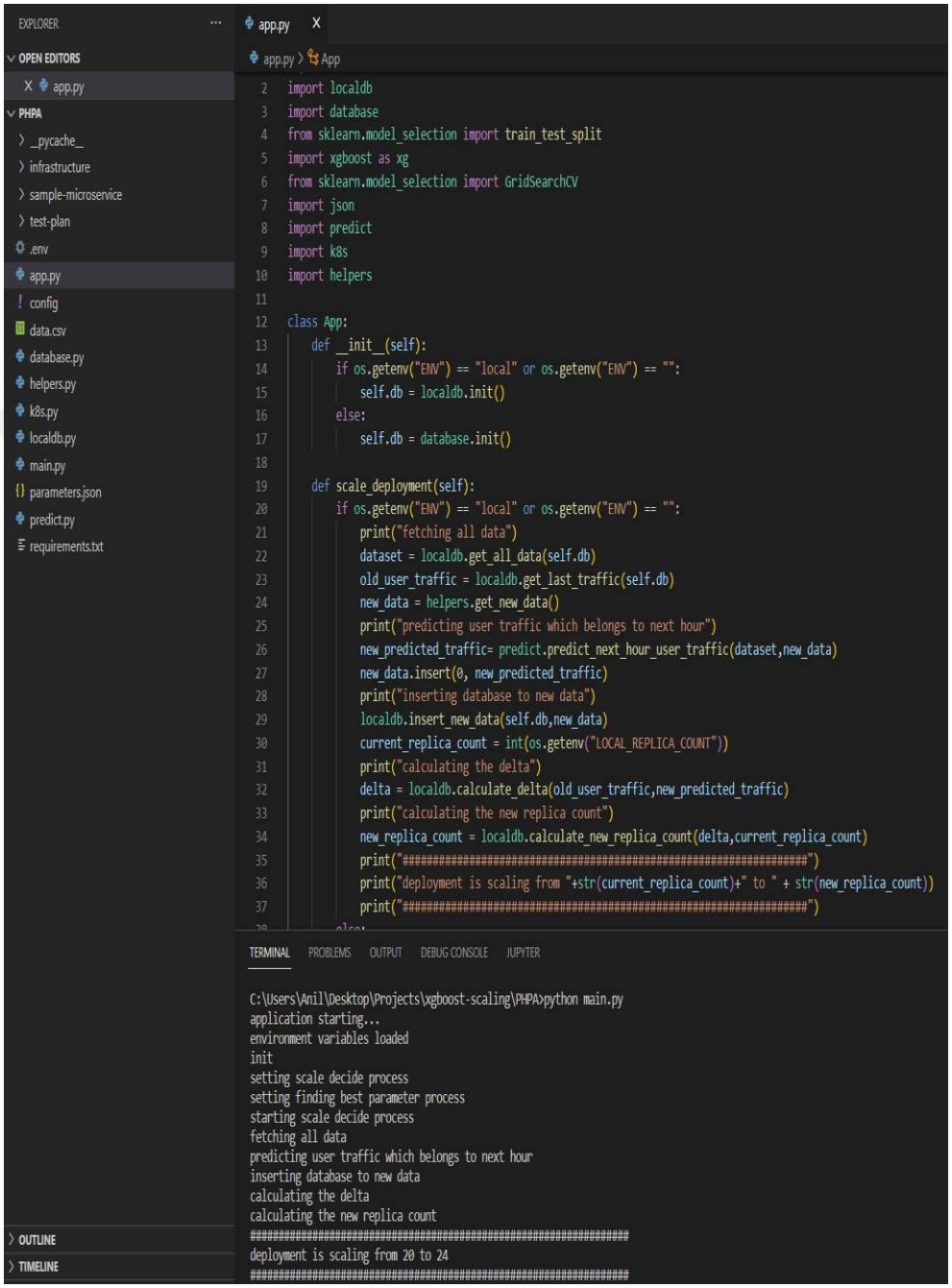

Figure 4.9. PHPA Application Code

# 5. RESULTS AND DISCUSSION

In this thesis, the efficiency between classical scaling systems in the cloud environment and the predictive scaling system called PHPA is compared. A load test, compiled from real user data and representing real users, was applied to the system, and then measurements were made. These measurement results are available below. Predictive scaling systems are found to be more efficient than classical scaling systems. This will be important information for how to scale applications in the cloud environment. Predictive scaling systems automatically perform the scaling process by predicting the future loads of the application. This eliminates the need for manual scaling and increases the speed of the process. In this thesis, the efficiency of the predictive scaling system called PHPA compared to classical scaling systems is examined.The metrics are based on the most followed and most demanded metrics in the cloud environment. These metrics are directly related to cost.

## 5.1. Pod Count

The number of pods that can be scheduled to each node in a Kubernetes environment is limited due to the availability of resources such as CPU and memory. As a result, the number of pods can have an impact on the cost of highly loaded systems. To ensure that this cost is minimized and to optimize the performance of the system, it is important to monitor the number of pods and the resource utilization of each node. This can help to identify any potential bottlenecks and allow for adjustments to be made to the number of pods or the resources allocated to them. During the load test in all environments, the number of pods was carefully measured in order to understand the impact on the system. By monitoring and adjusting the number of pods as needed, it is possible to optimize the performance and cost of a Kubernetes environment.

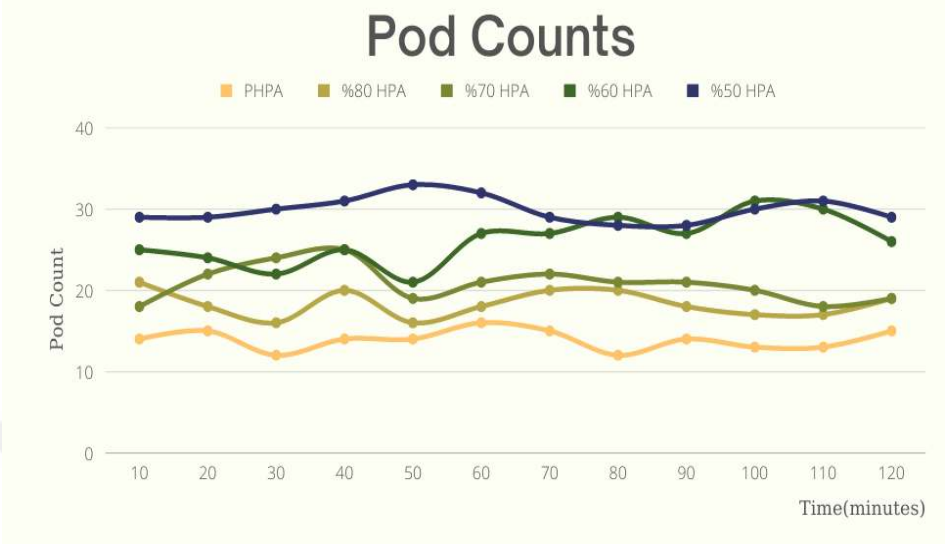


Figure 5.1. Pod Counts

As depicted in Figure 5.1, when a general graph is generated, it is observed that the method with the highest number of pods is 50% HPA. Upon further examination, it is apparent that 60% HPA follows closely behind, with 70% and 80% HPA values ranking lower. When considering the number of pods, the PHPA system appears to be the least efficient in terms of scaling, as it has the lowest number of pods observed. This suggests that, among the various HPA methods, 50% and 60% HPA may be more effective at scaling the system and utilizing resources efficiently, while the PHPA system may be less effective in this regard. It is important to carefully consider the efficiency and effectiveness of different scaling methods in order to optimize the performance and cost of a system.

Table 5.1. Pod Count Comparison

| Scaling Method | Minimum | Maximum | Average |
|---|---|---|---|
| PHPA | 12 | 16 | 14.5 |
| %80 HPA | 16 | 21 | 18 |
| %70 HPA | 18 | 25 | 21 |
| %60 HPA | 21 | 31 | 24 |
| %50 HPA | 28 | 31 | 30.5 |

The exact values for the different HPA methods can be found in Table 5.1. Upon examining this table, it becomes clear that PHPA performed the best, with a minimum pod count of 13, a maximum pod count of 16, and an average of 14.5 pods over the course of the experiment. HPA with a 80% threshold, on the other hand, had a minimum of 15 pods, a maximum of 21, and an average of 18 pods. The 70% HPA method had a minimum of 19 pods, a maximum of 25, and an average of 21 pods. At the 60% HPA threshold, the minimum number of pods was 21, the maximum was 29, and the average was 24. Finally, the 50% HPA method had the lowest performance, with a minimum of 28, a maximum of 33, and an average of 30.5 pods. These results suggest that, among the various HPA methods, PHPA may be the most effective at scaling the system efficiently and utilizing resources effectively.

### 5.2. Requested CPU

In a Kubernetes environment, CPU request values enable us to determine the amount of resources that pods will need to operate optimally. By establishing these resource requirements, we can guarantee that adequate resources are allocated to the pods, enabling them to run smoothly. If the usage of these allocated resources surpasses the predetermined threshold value, the system will automatically scale the

services and augment the amount of resources being reserved. This process facilitates the availability of the necessary resources for the pods, enabling them to run efficiently and effectively. The judicious utilization of CPU request values can facilitate the optimization of performance and cost in a Kubernetes environment.

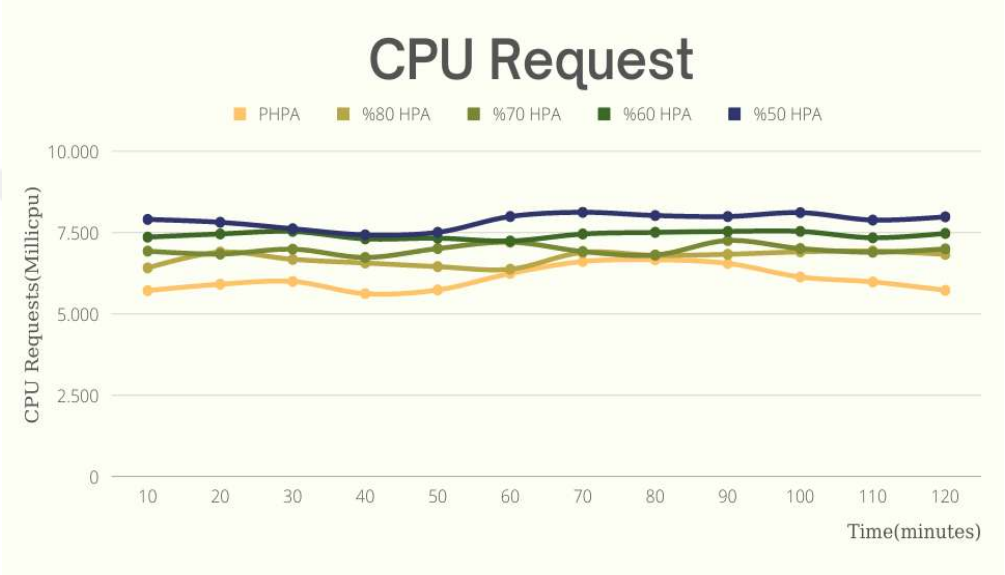


Figure 5.2. CPU Requests

Figure 5.2 provides a comparison of the CPU Request values for the different HPA methods. From this graph, it is clear that the method with the highest CPU Request values is 50% HPA. Upon closer examination, it becomes apparent that the 60%, 70%, and 80% HPA values have the next highest CPU Request values, in that order. The PHPA system, on the other hand, has the lowest CPU Request value of all the methods. This suggests that the 50% HPA method may require the most resources in order to scale the system, while the PHPA method may be the most efficient in terms of resource utilization. It is important to consider the resource requirements of different scaling methods in order to optimize the performance and cost of a system.

Table 5.2. CPU Request Comparison

| Scaling Method | Minimum | Maximum | Average |
|---|---|---|---|
| PHPA | 5606 | 6655 | 6110 |
| %80 HPA | 6363 | 6924 | 6633 |
| %70 HPA | 6725 | 7249 | 6980 |
| %60 HPA | 7234 | 7531 | 7384 |
| %50 HPA | 7423 | 8121 | 7794 |

Table 5.2 provides values on the basis of milliCPU for the different HPA methods. According to this table, PHPA had the best performance, with a minimum of 5606 milliCPU, a maximum of 6655 milliCPU, and an average of 6110 milliCPU over the course of the experiment. The HPA method with a 80% threshold had a minimum of 6363 milliCPU, a maximum of 6924 milliCPU, and an average of 6633 milliCPU. The 70% HPA method had a minimum of 6725 milliCPU, a maximum of 7249 milliCPU, and an average of 6980 milliCPU. At the 60% HPA threshold, the minimum value was 7234 milliCPU, the maximum was 7531 milliCPU, and the average was 7384 milliCPU. Finally, the 50% HPA method had the lowest performance, with a minimum of 7423 milliCPU, a maximum of 8121 milliCPU, and an average of 7794 milliCPU. These results suggest that PHPA may be the most effective at scaling the system efficiently and utilizing resources effectively, while the 50% HPA method may be the least efficient in this regard.

### 5.3. CPU Usage

The percentage of time that a computer program or operating system spends on processing its instructions and completing the tasks assigned to the central processing unit (CPU) is known as CPU usage. This behavior is not specific to Kubernetes or cloud computing environments. It's important to note that there is a

distinction between the CPU usage of an application and the CPU request usage. The CPU request refers to the amount of resources allocated to an application, while the CPU usage indicates how much of the allocated resources the application actually uses. As shown in Figure 5.3, PHPA did not have a significant impact on the resource usage of the applications being analyzed. It's worth noting that monitoring and optimizing CPU usage is an important aspect of ensuring the efficient and effective functioning of a computer system.

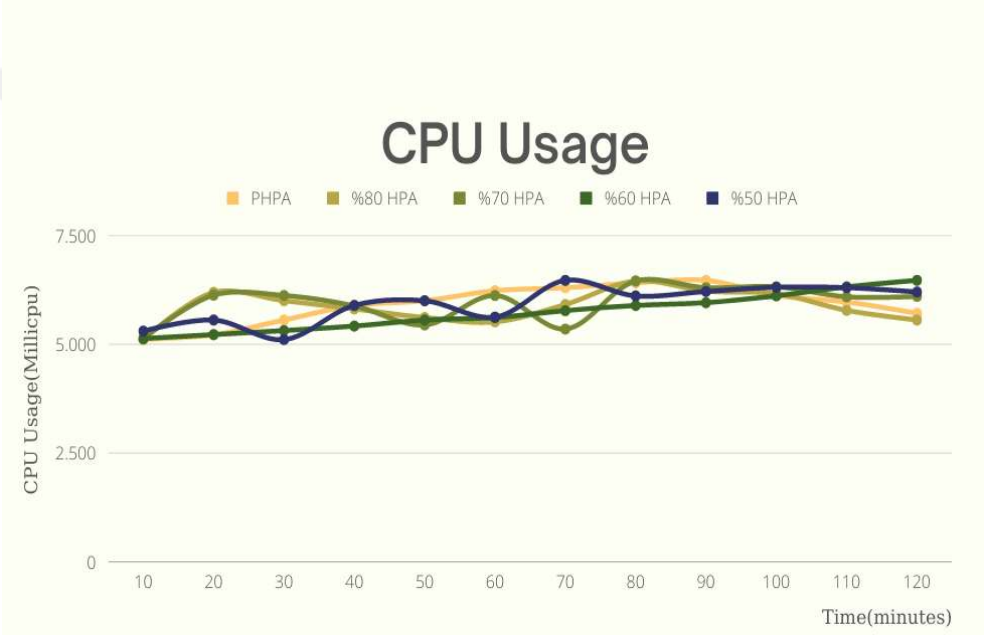


Figure 5.3. CPU Usage

As can be seen in Table 5.3, there was not much variation in the CPU usage during the experiment. The CPU usage of the PHPA system was recorded as a minimum of 5113 milliCPU, a maximum of 6469 milliCPU, and an average of 5802 milliCPU. For the 80% HPA system, the minimum, maximum, and average values were recorded as 5106 milliCPU, 6455 milliCPU, and 5784 milliCPU, respectively. Similarly, the minimum, maximum, and average values for the 70% HPA system were recorded as 5125 milliCPU, 6468 milliCPU, and 5814 milliCPU, respectively.

The values for the 60% HPA system were recorded as a minimum of 5127 milliCPU, a maximum of 6470 milliCPU, and an average of 5817 milliCPU. Finally, the minimum, maximum, and average values for the 50% HPA system were recorded as 5103 milliCPU, 6470 milliCPU, and 5769 milliCPU, respectively. It is worth noting that these values are important indicators of the performance and efficiency of the systems being analyzed.

Table 5.3. CPU Usage Comparison

| **Scaling Method** | **Minimum** | **Maximum** | **Average** |
|---|---|---|---|
| PHPA | 5113 | 6469 | 5802 |
| %80 HPA | 5106 | 6455 | 5784 |
| %70 HPA | 5125 | 6468 | 5814 |
| %60 HPA | 5127 | 6470 | 5817 |
| %50 HPA | 5103 | 6470 | 5769 |

### 5.4. Requested Memory

In the Kubernetes environment, Memory Request values allow us to specify the amount of memory that pods will require to function properly. Once this value has been defined, it helps to ensure that the number of pods needed for a particular service is optimal, and that the necessary system resources are reserved for that service. This process of resource allocation is repeated whenever the memory usage reaches the defined threshold. By carefully setting Memory Request values, we can help to optimize the performance of our Kubernetes pods and ensure that they have the resources they need to function effectively. It's worth noting that managing and optimizing memory usage is an important aspect of any cloud computing or Kubernetes environment, as it can help to ensure the smooth operation of our applications and services.

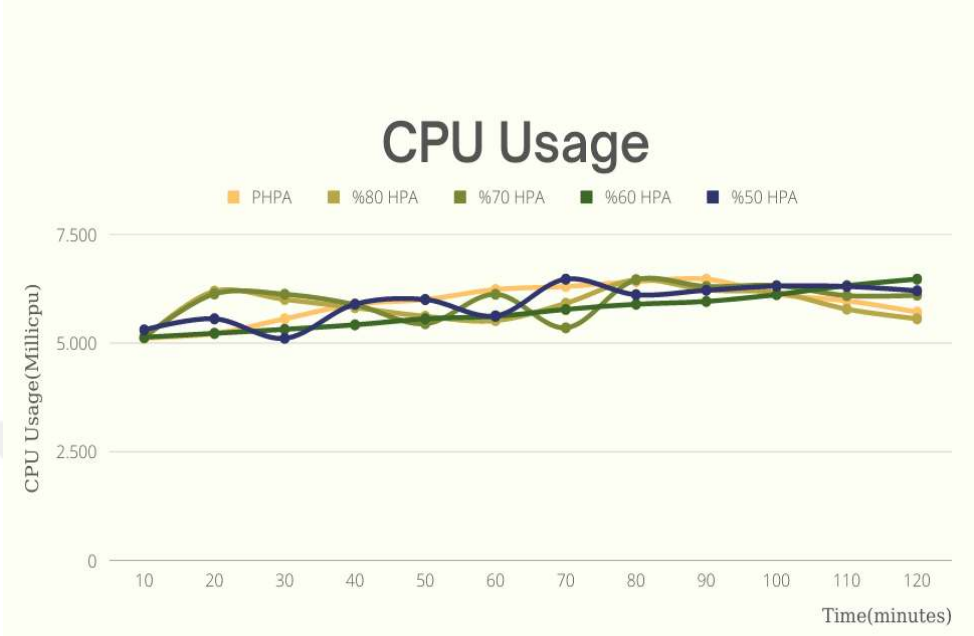


Figure 5.4. Requested Memory

In Figure 5.4, a comparison is presented of the Memory Request data for various systems under examination. The results indicate that the 50% HPA method exhibits the highest Memory Request value. The Memory Request values for the 60%, 70%, and 80% HPA systems are also depicted in the graph. It is noteworthy that the PHPA system exhibits the lowest Memory Request value among the methods being compared. This data is valuable in understanding the resource utilization patterns of these systems and can aid in optimizing the performance of our applications and services. It is of great importance to carefully manage and monitor memory usage, as inadequate memory resources can lead to impaired performance or even system failure. Conversely, the over-allocation of memory resources can result in inefficiency and increased costs.

Table 5.4. Memory Request Comparison

| Scaling Method | Minimum | Maximum | Average |
|---|---|---|---|
| PHPA | 1502 | 1531 | 1515 |
| %80 HPA | 1552 | 1592 | 1570 |
| %70 HPA | 1600 | 1671 | 1635 |
| %60 HPA | 1652 | 1701 | 1677 |
| %50 HPA | 1780 | 1842 | 1812 |

Table 5.4 presents Memory Request values in megabytes. As depicted in this table, PHPA exhibited the best performance with a minimum value of 1502 megabytes, a maximum value of 1531 megabytes, and an average value of 1515 megabytes. The 80% threshold for HPA reached a minimum value of 1552 megabytes, a maximum value of 1592 megabytes, and an average value of 1570 megabytes. At 70% HPA, a minimum value of 1600 megabytes, a maximum value of 1671 megabytes, and an average value of 1635 megabytes were achieved. The values for 60% HPA were recorded as a minimum of 1650 megabytes, a maximum of 1700 megabytes, and an average of 1677 megabytes. Finally, 50% HPA reached a minimum value of 1780 megabytes, a maximum value of 1840 megabytes, and an average value of 1812 megabytes. These results provide insight into the Memory Request patterns of the systems being analyzed and can assist in optimizing the performance and efficiency of our applications and services. It is important to carefully manage and monitor Memory Request values in order to ensure that our systems have the necessary resources to function optimally.

### 5.5. Memory Usage

Memory is the designated area within a computer or other electronic device where programs are executed and where temporary or permanent data is stored. As demonstrated in Figure 5.5, there is no correlation between Kubernetes Memory

request and Memory usage. Therefore, PHPA had no impact on memory usage. It is essential to understand the role that memory plays in the operation of our systems and to carefully manage and optimize its usage in order to ensure the smooth functioning of our applications and services. It is worth noting that the efficient utilization of memory resources can help to improve the performance and cost-effectiveness of our systems. In a Kubernetes environment, monitoring and managing memory usage is an important aspect of ensuring the optimal operation of our pods and services.

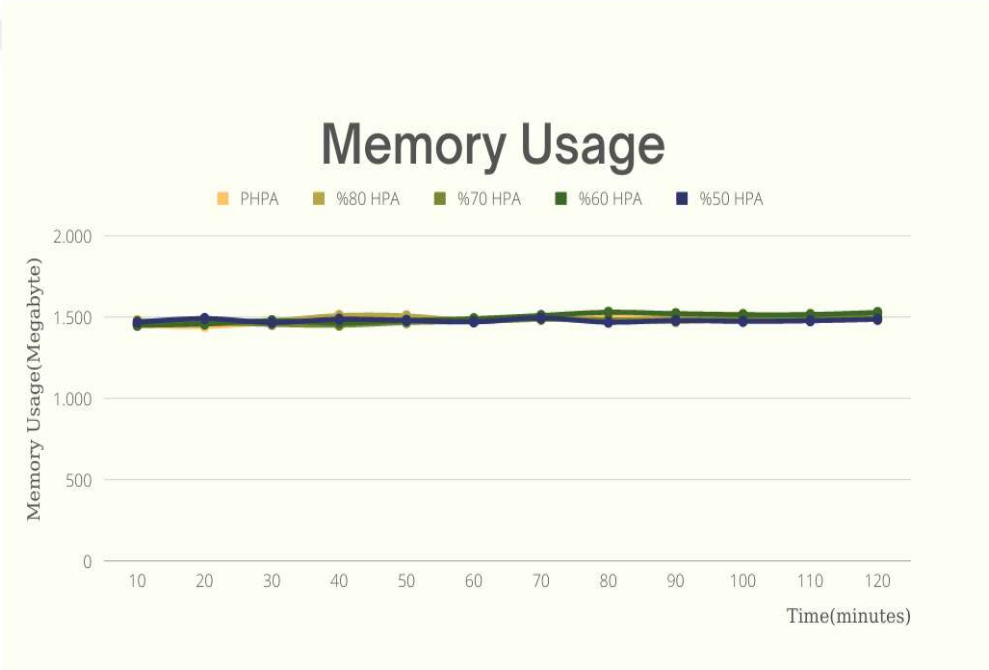


Figure 5.5. Memory Usage

As depicted in Table 5.5, there is little variation in CPU usage. The memory usage values for the PHPA system were recorded as a minimum of 1444 megabytes, a maximum of 1521 megabytes, and an average of 1474 megabytes. The minimum, maximum, and average values for the 80% HPA system were recorded as 1462 megabytes, 1510 megabytes, and 1479 megabytes, respectively. Similarly, the

minimum, maximum, and average values for the 70% HPA system were recorded as 1452 megabytes, 1499 megabytes, and 1482 megabytes, respectively. The values for the 60% HPA system were recorded as a minimum of 1449 megabytes, a maximum of 1530 megabytes, and an average of 1462 megabytes. Finally, the minimum, maximum, and average values for the 50% HPA system were recorded as 1468 megabytes, 1495 megabytes, and 1490 megabytes, respectively. It is important to understand the memory usage patterns of our systems in order to optimize their performance and efficiency. By carefully managing and monitoring memory usage, we can help to ensure the smooth operation of our applications and services.

Table 5.5. Memory Usage Comparison

| **Scaling Method** | **Minimum** | **Maximum** | **Average** |
|---|---|---|---|
| PHPA | 1444 | 1521 | 1474 |
| %80 HPA | 1462 | 1510 | 1479 |
| %70 HPA | 1452 | 1499 | 1482 |
| %60 HPA | 1449 | 1530 | 1462 |
| %50 HPA | 1468 | 1495 | 1490 |

### 5.6. Error Rate

The error rate metric quantifies the number of errors that occur within a sample service over a specific period of time. As demonstrated in Figure 5.6, there is no correlation between PHPA and the error rate metric. It is important to monitor and manage the error rate of our systems in order to ensure their reliability and performance. By carefully tracking the error rate, it can identify potential issues and take the necessary steps to address them in a timely manner. It is worth noting that the error rate can be influenced by a variety of factors, including system design, resource allocation, and environmental conditions. By understanding the underlying causes of errors, we can implement strategies to minimize their occurrence and

improve the overall performance of our systems.


Figure 5.6. Error Rate

As reflected in the values presented in Table 5.6, the error rate values are roughly equivalent. The minimum values for all systems are approximately 0 percent, while the maximum values are around 2 percent. The average values are approximately 1 percent for all systems.

Table 5.6. Error Rate Comparison

| Scaling Method | Minimum | Maximum | Average |
|---|---|---|---|
| PHPA | 0 | 2 | 0.99 |
| %80 HPA | 0 | 2.1 | 1.01 |
| %70 HPA | 0 | 1.9 | 1.01 |
| %60 HPA | 0 | 1.9 | 0.96 |
| %50 HPA | 0 | 1.8 | 0.99 |

### 5.7. Response Time

Response time refers to the average reaction time of a sample application. It is used to measure the elapsed time on the server side. Generally, an acceptable response time is around 4 seconds. If the response time is less than 1.2 seconds, it can be inferred that the application is performing well overall. It is important to monitor and optimize the response time of our systems in order to ensure their performance and user satisfaction. By minimizing the response time, we can improve the overall efficiency and effectiveness of our applications. Factors that can impact response time include system design, resource allocation, and environmental conditions. By understanding the factors that contribute to response time, we can implement strategies to optimize the performance of our systems and enhance the user experience.


Figure 5.7. Response Time

As can be seen in Table 5.7, the scaling system did not cause a noticeable change in the response time.

Table 5.7. Response Time Comparison

| Scaling Method | Minimum | Maximum | Average |
|---|---|---|---|
| PHPA | 10 | 20 | 14.5 |
| %80 HPA | 9 | 22 | 16.5 |
| %70 HPA | 11 | 19 | 13.6 |
| %60 HPA | 10 | 21 | 15 |
| %50 HPA | 9 | 19 | 13 |

### 5.8. Cost Results

The present research aims to achieve cost reduction and efficient resource utilization in the context of cloud computing. To this end, an analysis of costs was conducted, taking into consideration various relevant data that was obtained. The ultimate objective of the study is to optimize the utilization of resources in order to minimize expenses related to cloud infrastructure. In order to accomplish this goal, it is essential to accurately assess the costs associated with various resource utilization scenarios, and to identify opportunities for optimization. By carefully examining the data obtained, it is possible to identify trends and patterns that can inform decision-making processes related to resource allocation and cost management. Through this approach, it is hoped that the results of the study will provide valuable insights for organizations seeking to optimize their cloud-based operations and reduce their overall costs.

As demonstrated in Figure 5.8.a, the total cost of utilizing 50% HPA (Horizontal Pod Autoscaler) is approximately $412 per month. This cost is accompanied by a memory efficiency of 61% and a CPU efficiency of 65%. These efficiency metrics are important to consider when evaluating the cost-effectiveness of different resource utilization strategies. By comparing the costs and efficiencies of different scenarios, it is possible to identify options that offer the best balance

between cost and performance. Ultimately, the goal is to maximize efficiency while minimizing costs, and the data presented in Figure 5.8.a provides valuable insights that can inform these decision-making processes.

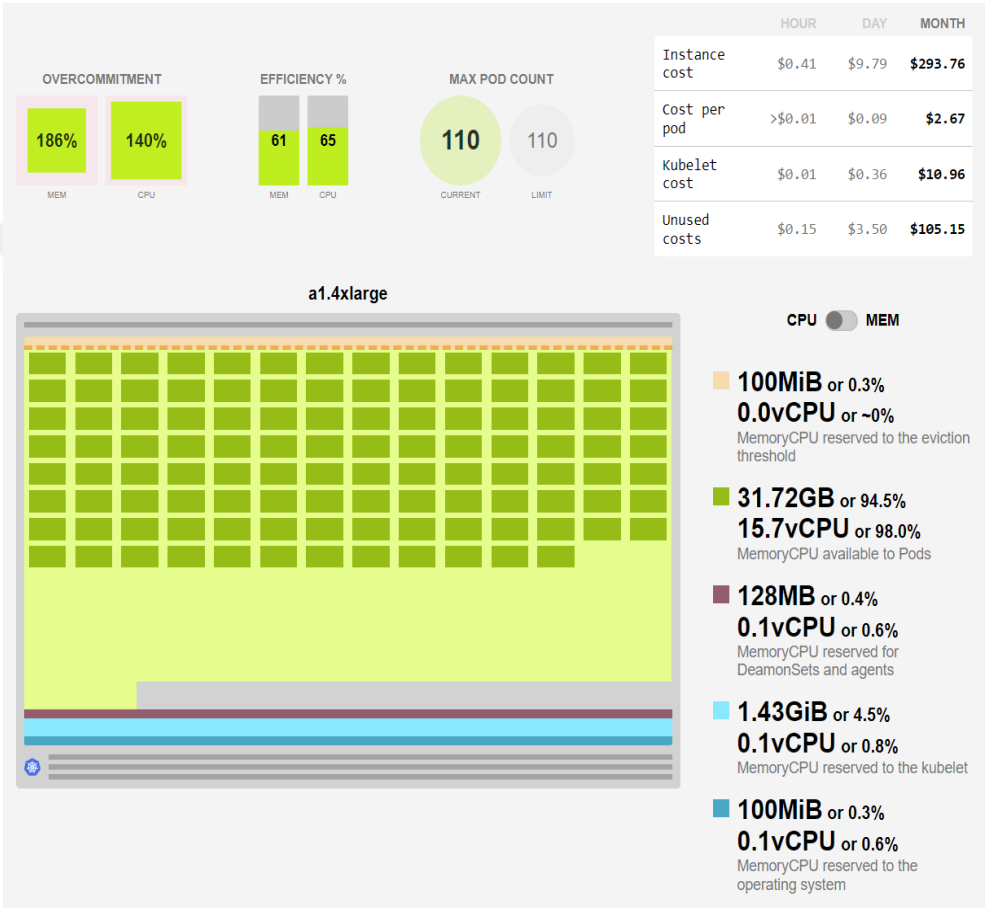


Figure 5.8. %50 HPA Cost

As depicted in Figure 5.9. the cost of utilizing 60% HPA (Horizontal Pod Autoscaler) is approximately $393 per month. Alongside this cost, the associated memory efficiency is 68% and the CPU efficiency is 72%. These metrics are important indicators of the effectiveness of different resource utilization strategies,

as they provide insights into the balance between cost and performance.

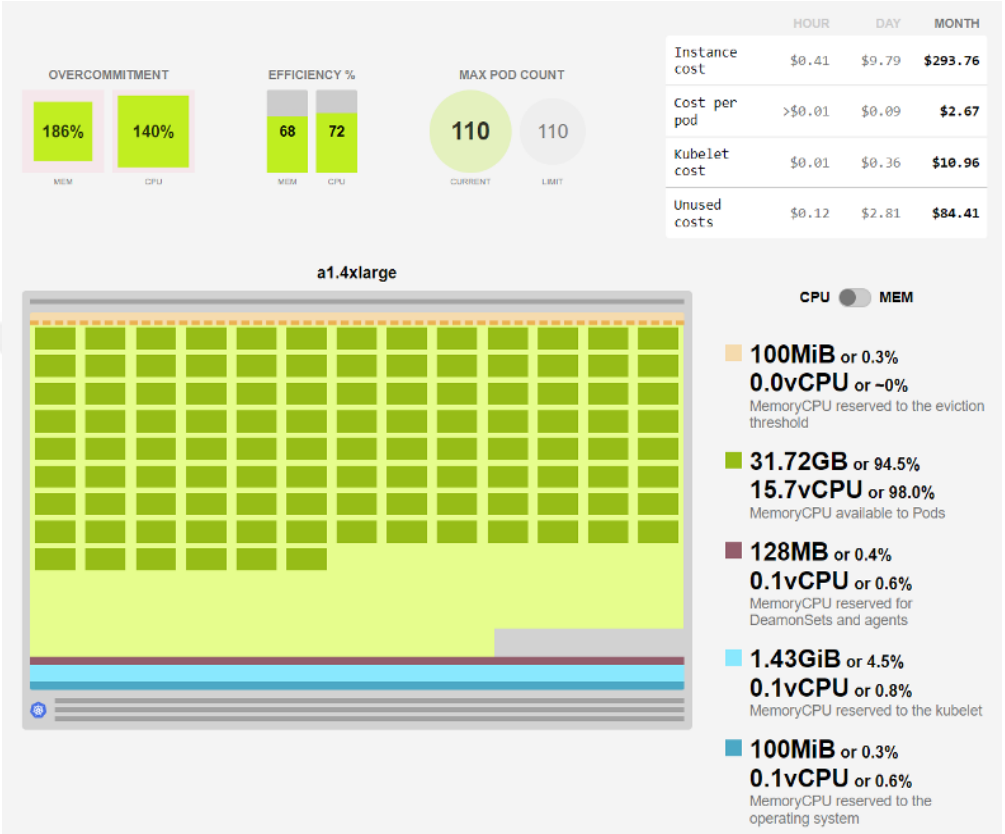


Figure 5.9. %60 HPA Cost

As illustrated in Figure 5.10, the use of 70% HPA (Horizontal Pod Autoscaler) results in a total cost of approximately $380 per month. In addition to this cost, the corresponding memory efficiency is 72% and the CPU efficiency is 77%.

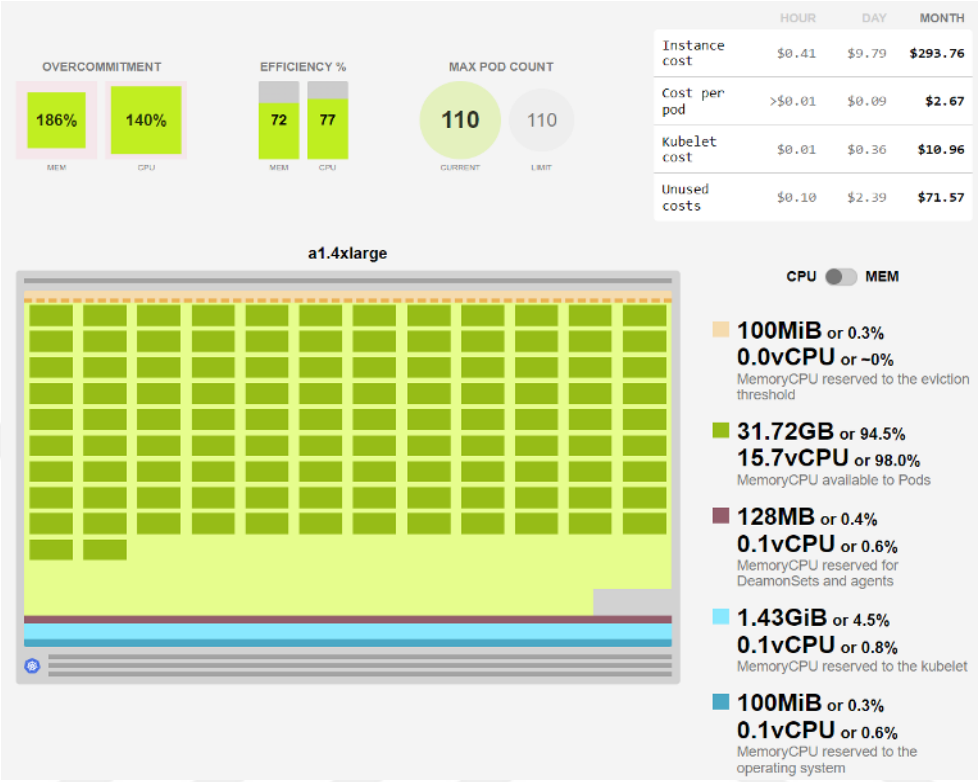


Figure 5.10. %70 HPA Cost

According to Figure 5.11, the total cost of utilizing 80% HPA (Horizontal Pod Autoscaler) is approximately $367 per month, with accompanying memory efficiency of 76% and CPU efficiency of 82%.

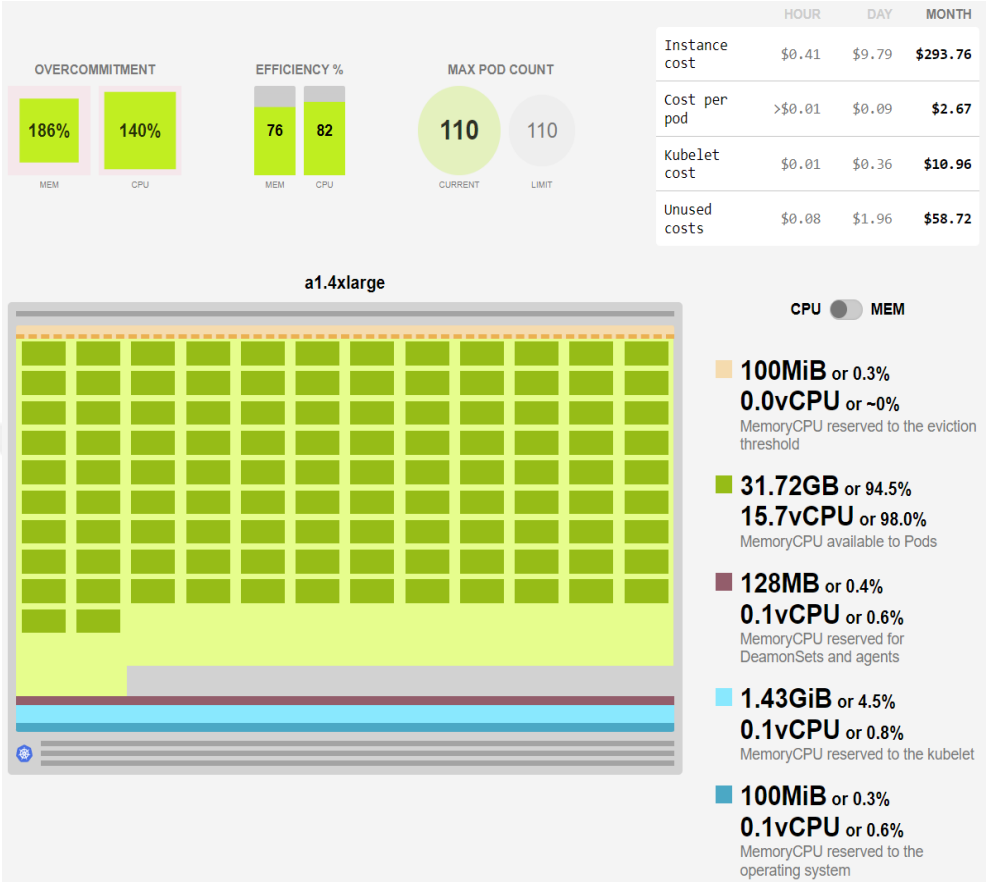


Figure 5.11. %80 HPA Cost

Figure 5.12 indicates that the total cost of utilizing PHPA (Proportional Horizontal Pod Autoscaler) is approximately $355 per month, accompanied by memory efficiency of 80% and CPU efficiency of 87%

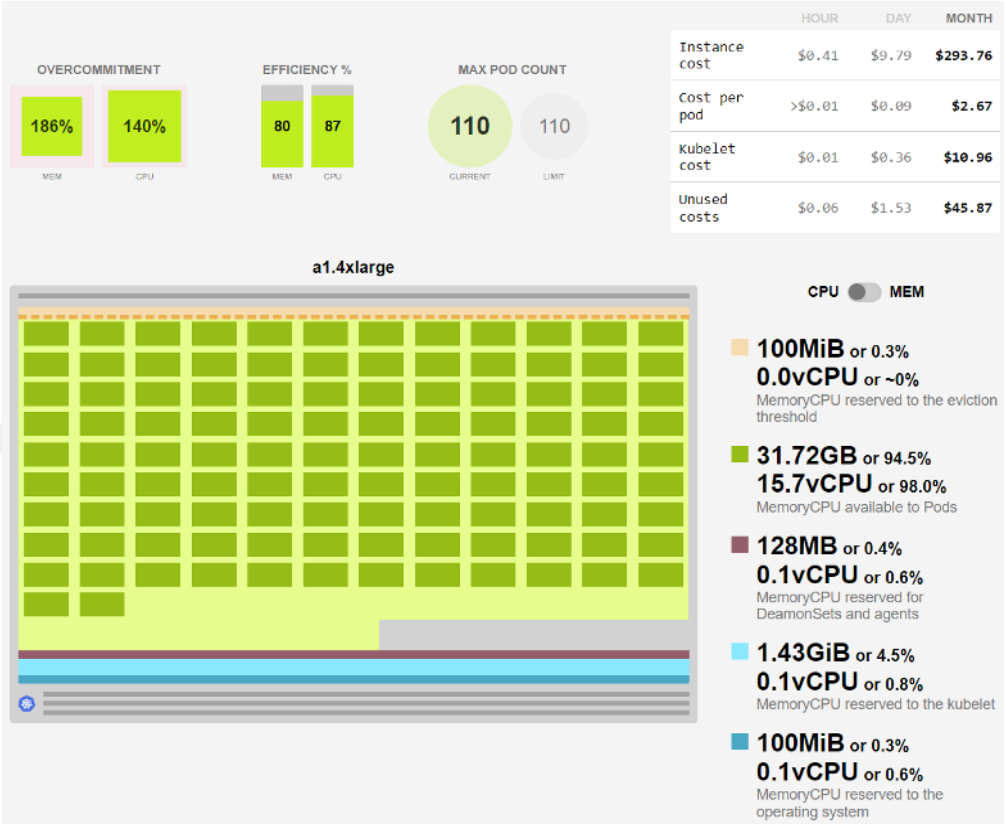


Figure 5.12. PHPA Cost

Detailed information on the various cost calculations is presented in Table 5.8. As can be seen, all of the scaling systems incur both a mandatory fixed cost and a redundant cost. Among these systems, PHPA (Proportional Horizontal Pod Autoscaler) has proven to be the most effective in minimizing the redundant cost. Specifically, PHPA has the lowest unused cost, at $45, while 50% HPA (Horizontal Pod Autoscaler) has the highest unused cost, at $105. These costs are reflected in the total cost field, and contribute to the overall cost-effectiveness of different resource utilization strategies.

It is important to consider the trade-off between cost and performance when evaluating different resource utilization options. While it is generally desirable to minimize costs, it is also important to ensure that sufficient resources are available to meet the demands of the system. By carefully analyzing the costs and efficiencies of different scenarios, it is possible to identify options that offer the best balance between cost-effectiveness and performance. Through this approach, it is hoped that the results of the study will provide valuable insights for organizations seeking to optimize their resource utilization and reduce their overall costs.

Table 5.8. Cost Comparison

| **Scaling Method** | **Fixed Cost** | **Unused Cost** | **Total Cost** |
|---|---|---|---|
| PHPA | 315$ | 45$ | 360$ |
| %80 HPA | 315$ | 58$ | 373$ |
| %70 HPA | 315$ | 71$ | 386$ |
| %60 HPA | 315$ | 84$ | 399$ |
| %50 HPA | 315$ | 105$ | 420$ |

### 5.9. General Discussions of Results

The findings of this study indicate that the machine learning-based scaling model called PHPA (Proportional Horizontal Pod Autoscaler) outperforms the default scaling system used by Kubernetes. This comparison is based not only on resource use, but also on the resources that are kept as redundant in order to enhance service quality and reliability. By utilizing PHPA, it is possible to achieve a more efficient utilization of resources, resulting in less occupied resources and an improved quality of service.

To further clarify these results, it is important to examine the details of the comparison. The effectiveness of PHPA can be demonstrated by comparing it to the default scaling system in terms of various performance metrics, such as cost, efficiency, and reliability. By analyzing these metrics, it is possible to identify the strengths and weaknesses of different resource utilization strategies, and to identify opportunities for optimization. Ultimately, the goal is to select the scaling model that offers the best balance between cost-effectiveness and performance.

- In this study, the Extended Gradient Boosting method, a machine learning approach, was used to forecast traffic for a 2-day period and to inform resource allocation decisions. The results of the study are based on data sets compiled from publicly available data sources, including EPİAŞ and smartpulse traffic data. The results of the methods used in the study were presented in the form of tables and graphs, with a total of 9 tables and 10 graphs being created and visualized. In order to determine the most appropriate method, MPE (Mean Percent Error) , Memory Usage , CPU Usage Percentage , Completion Time values were analyzed and the method with the most fit appropriate one was selected and applied. It is crucial for this method to outperform traditional approaches in order to produce the most accurate estimates.
- In this study, resource usage and resource allocation are determined based on traffic patterns, using a linear incremental sample microservice. Because there is a relationship between user traffic and resource usage, it is possible to estimate resource allocation directly from traffic data. With these estimates, it is possible to determine the number of copies and replicas that the microservice needs to run, and to optimize the utilization of resources to meet these requirements. By taking this approach, it is possible to ensure that resources are used

efficiently, in order to support the smooth operation of the microservice.

- In this study, a range of machine learning algorithms were considered, including ARIMA, XGBoost, Long Short Time Memory, Multilayer Perceptron, and Hierarchical Temporal Memory. All of these algorithms were run under the same conditions in order to determine the most suitable option for the study. The algorithm selection was based on various metrics related to resource usage, MPE (Mean Percent Error), and time, which were collected and analyzed as part of the study. Based on the results of this analysis, it was decided to use the XGBoost algorithm throughout the study.The decision to use XGBoost was based on the performance of the algorithm in terms of resource usage, MPE, and time. These metrics are important indicators of the cost-effectiveness and efficiency of different machine learning approaches, and by considering them, it is possible to identify the algorithm that offers the best balance between performance and cost. Through this approach, it is hoped that the results of the study will provide valuable insights for organizations seeking to optimize their resource utilization and improve their machine learning capabilities.
- The XGBoost algorithm was chosen as the most suitable option for PHPA (Proportional Horizontal Pod Autoscaler) in this study due to its ability to efficiently utilize resources and produce fast results. When comparing different algorithms, it is appropriate to consider the MPE (Mean Percent Error) value, as this metric provides a reliable indication of the overall impact on the system. This is particularly useful in this case, since all of the values in the dataset are positive and MPE is a suitable measure for assessing the overall effect on the system. The XGBoost algorithm is used by the PHPA system to predict future load and to determine the appropriate number of pods to scale in the system.

This helps to prevent the waste of system resources and to ensure that resources are used efficiently. Selecting the most suitable machine learning algorithm, it is possible to optimize resource utilization and improve the performance of the PHPA system.

- In this study, the PHPA (Proportional Horizontal Pod Autoscaler) and HPA (Horizontal Pod Autoscaler) scaling systems were compared in order to assess their relative efficiency. The comparison results showed that PHPA is on average 8.5% more efficient than 80% HPA in terms of CPU request results. In addition, PHPA was found to be 14.2%, 20.8%, and 27.5% more efficient on average compared to 70%, 60%, and 50% HPA systems, respectively. These findings demonstrate that PHPA is more efficient in terms of CPU resource utilization, which is an important consideration in the context of this study. Given the recommendation to use 70% or 60% threshold as Best Practice, the use of PHPA would result in 20.8% and 27.5% less idle and redundant resources being kept in the cloud environment for these values. This helps to optimize resource utilization and to minimize costs, while still ensuring that sufficient resources are available to meet the demands of the system. Analyzing the efficiency of different resource utilization strategies, it is possible to identify options that offer the best balance between cost-effectiveness and performance.
- An analysis of Memory Request values reveals that the PHPA (Proportional Horizontal Pod Autoscaler) system is more efficient than the HPA (Horizontal Pod Autoscaler) system. According to the benchmark results, PHPA is on average 3.5% more efficient than 80% HPA in terms of Memory Request results. In addition, PHPA was found to be 7.9%, 10.6%, and 19.6% more efficient on average compared to 70%, 60%, and 50% HPA systems, respectively. These findings

demonstrate that PHPA is more efficient in terms of Memory Request, which is an important consideration in the context of this study. Given the recommendation to use 70% or 60% threshold as best practice, the use of PHPA would result in 10.6% and 19.6% fewer idle and redundant resources being kept in the cloud environment for these values. This helps to optimize resource utilization and to minimize costs, while still ensuring that sufficient resources are available to meet the demands of the system.

- In addition to the metrics already discussed, this study also examined Error Rate, Response Time, Memory Usage, and CPU Usage. On average, Error Rate values were around 1%, Response Time values were around 15 milliseconds, Memory Usage values were around 1450 Megabytes, and CPU Usage values were around 5800 milliSeconds. Based on these values, it appears that the PHPA (Proportional Horizontal Pod Autoscaler) system does not have a significant impact on these metrics, and it is not possible to achieve a noticeable level of efficiency in these areas. While the PHPA system may not offer significant improvements in terms of these particular metrics, it is important to consider the overall balance of performance and cost-effectiveness when selecting a resource utilization strategy.

# 6. CONCLUSION

Machine learning-based prediction has become an increasingly important tool across a wide range of fields. In this study, a cloud environment was simulated and a machine learning-based automatic scaling system called PHPA was developed using the XGBoost algorithm. The performance of PHPA was compared with that of the default Kubernetes scaling system, HPA, using threshold values of 50%, 60%, 70%, and 80%. In order to make predictions with PHPA, a dataset was created by combining data from EPİAŞ and Smartpulse between 01.07.2021 and 31.12.2021. During the comparison, a 2-day load was simulated and sent to the system in 2-hour increments, and then distributed to other systems using PHPA and a Fanout Proxy. All system metrics were collected and compared in a single metric system.

The results of this comparison show that PHPA is approximately 20% more efficient than other systems in the CPU Request metric, and around 10% more efficient in the Memory Request metric. These values do not represent the resources actually used by the software, but rather the excess resources allocated for the software to run. By using a prediction model to scale the system, PHPA is able to allocate fewer resources to the same software, which also has the added benefit of lower costs. In terms of pricing, PHPA was found to be more cost-effective than HPA. These methods can be applied to systems operating in the cloud environment and can provide a noticeable cost advantage, especially for high-traffic applications.

## REFERENCES


Al-Dhuraibi, Y., Paraiso, F., Djarallah, N. and Merle, P., "Autonomic Vertical Elasticity of Docker Containers with ELASTICDOCKER," 2017 IEEE 10th International Conference on Cloud Computing (CLOUD), 2017, pp. 472-479, doi: 10.1109/CLOUD.2017.67.

Arabnejad, H., Pahl, C., Jamshidi, P., and Estrada, G., "A Comparison of Reinforcement Learning Techniques for Fuzzy Cloud Auto-Scaling," 2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID), 2017, pp. 64-73, doi: 10.1109/CCGRID.2017.15.

Arabnejad, H., Pahl, C., Jamshidi, P., and Estrada, G., "A Comparison of Reinforcement Learning Techniques for Fuzzy Cloud Auto-Scaling," 2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID), 2017, pp. 64-73, doi: 10.1109/CCGRID.2017.15.

Balla, D., Simon, C. and Maliosz, M., "Adaptive scaling of Kubernetes pods," NOMS 2020 - 2020 IEEE/IFIP Network Operations and Management Symposium, 2020, pp. 1-5, doi: 10.1109/NOMS47738.2020.9110428.

Baresi, L., Hu, D.Y.X., Quattrocchi, G., Terracciano, L. (2021). KOSMOS: Vertical and Horizontal Resource Autoscaling for Kubernetes. In: Hacid, H., Kao, O., Mecella, M., Moha, N., Paik, Hy. (eds) Service-Oriented Computing. ICSOC 2021. Lecture Notes in Computer Science(), vol 13121. Springer, Cham. https://doi.org/10.1007/978-3-030-91431-8_59

Chen, Tianqi,, and Carlos, Guestrin. "Xgboost: A scalable tree boosting system." Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining. 2016.

Dang-Quang, N.-M.; Yoo, M. Deep Learning-Based Autoscaling Using Bidirectional Long Short-Term Memory for Kubernetes. Appl. Sci. 2021, 11, 3835. https://doi.org/10.3390/app11093835

Ding, Z. and Huang, Q., "COPA: A Combined Autoscaling Method for Kubernetes," 2021 IEEE International Conference on Web Services (ICWS), 2021, pp. 416-425, doi: 10.1109/ICWS53863.2021.00061.

Hu, T. and Wang, Y., "A Kubernetes Autoscaler Based on Pod Replicas Prediction," 2021 Asia-Pacific Conference on Communications Technology and Computer Science (ACCTCS), 2021, pp. 238-241, doi: 10.1109/ACCTCS52002.2021.00053.

Jain, G. and Prasad, R. R., "Machine learning, Prophet and XGBoost algorithm: Analysis of Traffic Forecasting in Telecom Networks with time series data," 2020 8th International Conference on Reliability, Infocom Technologies and Optimization (Trends and Future Directions) (ICRITO), 2020, pp. 893-897, doi: 10.1109/ICRITO48877.2020.9197864.

Khaleq, A. A. and Ra, I., "Intelligent Autoscaling of Microservices in the Cloud for Real-Time Applications," in IEEE Access, vol. 9, pp. 35464-35476, 2021, doi: 10.1109/ACCESS.2021.3061890.

Kubernetes, Cost, Calculator. https://learnk8s.io/kubernetes-instance-calculator, accessed 2022-10-16.

Mart, O., Negru, C., Pop, F. and Castiglione, A., "Observability in Kubernetes Cluster: Automatic Anomalies Detection using Prometheus," 2020 IEEE 22nd International Conference on High Performance Computing and Communications; IEEE 18th International Conference on Smart City; IEEE 6th International Conference on Data Science and Systems (HPCC/SmartCity/DSS), 2020, pp. 565-570, doi: 10.1109/HPCC-SmartCity-DSS50907.2020.00071.

Nascimento, D.C., Pires, C.E. & Mestre, D.G. Applying machine learning techniques for scaling out data quality algorithms in cloud computing environments. Appl Intell 45, 530–548 (2016). https://doi.org/10.1007/s10489-016-0774-2

Nguyen, T.-T.; Yeom, Y.-J.; Kim, T.; Park, D.-H.; Kim, S. Horizontal Pod Autoscaling in Kubernetes for Elastic Container Orchestration. Sensors 2020, 20, 4621. https://doi.org/10.3390/s20164621

Rossi, F., Nardelli, M. and Cardellini, V., "Horizontal and Vertical Scaling of Container-Based Applications Using Reinforcement Learning," 2019 IEEE 12th International Conference on Cloud Computing (CLOUD), 2019, pp. 329-338, doi: 10.1109/CLOUD.2019.00061.

Rossi, Fabiana. "Auto-scaling Policies to Adapt the Application Deployment in Kubernetes." ZEUS. 2020.

Schuler, L., Jamil, S. and Kühl, N., "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), 2021, pp. 804-811, doi: 10.1109/CCGrid51090.2021.00098.

Sukhija, N. and Bautista, E., "Towards a Framework for Monitoring and Analyzing High Performance Computing Environments Using Kubernetes and Prometheus," 2019 IEEE SmartWorld, Ubiquitous Intelligence & Computing, Advanced & Trusted Computing, Scalable Computing & Communications, Cloud & Big Data Computing, Internet of People and Smart City Innovation (SmartWorld/SCALCOM/UIC/ATC/CBDCom/IOP/SCI), 2019, pp. 257-262, doi: 10.1109/SmartWorld-UIC-ATC-SCALCOM-IOP-SCI.2019.00087.

Tamiru, M. A., Tordsson, J., Elmroth, E., and Pierre, G. "An Experimental Evaluation of the Kubernetes Cluster Autoscaler in the Cloud," 2020 IEEE International Conference on Cloud Computing Technology and Science (CloudCom), 2020, pp. 17-24, doi: 10.1109/CloudCom49646.2020.00002.

Toka, L., Dobreff, G., Fodor, B. and Sonkoly, B., "Adaptive AI-based auto-scaling

for Kubernetes," 2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID), 2020, pp. 599-608, doi: 10.1109/CCGrid49817.2020.00-33.

Troia, S., Alvizu, R., Zhou, Y., Maier, G. and Pattavina A., "Deep Learning-Based Traffic Prediction for Network Optimization," 2018 20th International Conference on Transparent Optical Networks (ICTON), 2018, pp. 1-4, doi: 10.1109/ICTON.2018.8473978.

Wang, M., Zhang, D. and Wu, B., "A Cluster Autoscaler Based on Multiple Node Types in Kubernetes," 2020 IEEE 4th Information Technology, Networking, Electronic and Automation Control Conference (ITNEC), 2020, pp. 575-579, doi: 10.1109/ITNEC48623.2020.9084706.

Zhenhuan, Gong, Xiaohui, Gu,. and Wilkes, J., "PRESS: PRedictive Elastic ReSource Scaling for cloud systems," 2010 International Conference on Network and Service Management, 2010, pp. 9-16, doi: 10.1109/CNSM.2010.5691343.

**CURRICULUM VITAE**

Anıl KUŞÇU. After completing elementary and high school at Adana, he graduated from the Computer Engineering Department of Çukurova University in 2019. He started the MSc program at the Department of Computer Engineering, Çukurova University, Adana, in 2020 under the supervision of Prof. Dr. M.Fatih Akay, and graduated in 2020