School of Computing
FACULTY OF ENGINEERING AND
PHYSICAL SCIENCES

UNIVERSITY OF LEEDS

Final Report

Using Machine Learning to Achieve Energy Aware Kubernetes Scaling

Thomas Charles Leath

Submitted in accordance with the requirements for the degree of
BSc Computer Science (Digital and Technology solutions)

2023/2024

COMP3932 Synoptic Project

The candidate confirms that the following have been submitted*:*

Items	**Format**	**Recipient(s) and Date**
Final Report	*PDF file*	*Uploaded to Minerva (16/05/24)*
Link to online code repository	*URL*	*Sent to supervisor and assessor (16/05/24)*

The candidate confirms that the work submitted is their own and the appropriate credit has been given where reference has been made to the work of others.

I understand that failure to attribute material which is obtained from another source may be considered as plagiarism.

Thomas Leath

Summary

One of the main drivers of Cloud adoption is its ability to enable a near constant availability for a business' applications. This has made cloud infrastructure, like Kubernetes, a very popular method of deployment for businesses that are looking to host their application on the Cloud. However, this prioritisation of availability has meant most deployments have too many allocated relative to the volume of demand that will be incurred.

This overallocation means that nearly 87% are running at idle at any one time – counterintuitively, during this idle time, the applications can consume almost 60% of the maximum power of the server. This, therefore, not only wastes the energy used by the servers, but also the money of the companies and corporations who have deployed the applications.

The solution to this problem is to have system in place to only allocate application resources when they are needed, this would mean that there are no resources running at idle and waiting to be used. Whilst these systems do exist in the form of Auto-Scalers, they require a threshold to be passed before they begin. This threshold means that they are, by nature, a reactive system and often too general to accurately schedule the appropriate number of resources that each individual application requires.

This paper investigated creating a new Auto- Scaler that looked ahead. It did so by using a time-series machine learning model to predict the amount of future demand an application will experience, whilst also understanding how an application consumes resources at an individual level in order to provide the optimum number of resources at all times.

Many current solutions focus on configurations based on reducing CPU usage. However, because the amount of energy consumed by an application is defined by more than just the amount of CPU it is using, these solutions do not optimise the reduction in energy consumption. This solution instead defined the optimum number of resources needed to be provided to an application (known as an application profile), in order to achieve the lowest amount of energy consumption. This was done whilst maintaining a minimum response time of 300ms.

Implementing the new proactive autoscaler, which was designed in this project, reduced energy consumption by 50% across all tests. This was when compared to the standard Horizontal Pod Autoscaler available within Kubernetes. This therefore demonstrated the significance of taking proactive auto scalers in the direction of energy consumption, rather than focussing solely on CPU usage, as literature has previously demonstrated.

Acknowledgements

I would like to acknowledge Karim Djemame, for his assistance as a supervisor throughout his project as well as his tutoring in COMP3211 and COMP5123M which helped to develop my knowledge Cloud computing and Distributed systems.

Table of Contents

Chapter 1
Introduction and Background Research

1.1 Introduction

The issue of Cloud waste is a prominent discussion point within the topics of sustainability in technology. When left idle and without purpose, resources deployed on the Cloud, can result in vast amounts of excess energy waste and unnecessary costs. This paper investigated the inefficiencies of current methods that have been designed to reduce Cloud waste, as well as the extent of their impact onto the excessive energy usage of applications on the Cloud. Developing from this, this project aimed to outline a framework which mitigated these inefficiencies, improved resource utilisation, and by proxy the amount of energy wasted through over-provisioned applications.
Cloud computing is the infrastructure and software behind on-demand access to computer systems from anywhere. It uses a collection of data centres around the world to allow any device, when connected to the internet, to access its full storage, and network, and capabilities to compute.

Over the last decade, usage of Cloud computing has continuously increased and is now used by 94% of enterprise organisations [1]. It has rapidly become an essential part of most modern technology systems – including social media, cars, banks, and even fridges – and has helped businesses around the globe migrate their own infrastructure off-site. With industry spending rising by 289% since 2014, up to $678.8 billion by the end of 2024 [2][3], it is clear this scale up will continue to rise.

The issue of Cloud waste started to become evident as this large increase in scale also lead to a large increase in consumption. With data centres predicted to consume 2967 TWh a year by 2030, and to contribute to an estimated 14% of CO_2 emissions by 2040, the effect that ever-increasing Cloud consumption will have on not only resource usage, but the future of sustainability, has become clear. [4]

In addition to the impact Cloud usage has on the carbon footprint of the runners of these data centres, it is also incredibly expensive. It has been reported by Amazon that approximately 53% of their data centre operating costs are attributed to their energy consumption. [5]
As a result of both these negative environmental and economic impacts, there have been large efforts in place to attempt to reduce energy consumption associated with Cloud usage, and Cloud waste. One of the most popular attempts has been placing data centres in extreme cold climates, as this reduces the amount of energy required to cool the site [6]. Alternatively, Google and Apple have also made steps to decrease the carbon footprint of the energy they consume by building data centres in places where exploitation of greener hydroelectric power is an option [7].

But, whilst steps have been taken on the server-side, a huge part of improving the efficiency of resource management is with the clients using these servers, more specifically how they manage them. It is the responsibility of the client, rather than the provider, to control how the resources are consumed, based on the applications they want to use, and have, on the Cloud.

1.1.1 Application Deployment in Datacentres

The Cloud, and the way it is used by clients, has changed drastically. Where it was once only possible to deploy an application by using whole machine, a strive for flexibility and elasticity has led to the development of Serverless computing. Serverless computing transfers the responsibility of the management of servers from the developer to the cloud provider, meaning the developers can now focus on writing and deploying code. Using containers, they can deploy all the libraries, code and

dependencies required as one executable unit, abstracted from the unnecessary complexity of managing an entire machine. [8] However, in a production environment, larger and more complex applications need several individual containers to operate [9], and so, a system is required to be able to manage them – this system is Kubernetes.

Kubernetes deploy collections of containers, known as pods, onto a group of machines called nodes, and together these form a Cluster [9]. Kubernetes' clusters simplify the management of large applications on the Cloud by constantly monitoring and maintaining these large collections of containers. This maintenance ensures the applications within it always have the correct number of resources [9]. All together they provide a system that, maintains a level of fault tolerance, scalability, and balance load, to effectively host production level workloads.

1.1.2 Management of Resource Waste and Inefficiencies with Auto Scaling

Although one of the main benefits of, and reasons to employ, Kubernetes is their ability to scale. This scalability is only as effective as the configuration in which it is defined. In reality, the usage of Kubernetes is often very inefficient with an average of only 13% of provisioned CPU resources actually utilised (out of 4000 clusters analysed). [10] This over provisioning by companies and development teams wastes a huge amount of energy as, although they may be inactive, even idle servers can consume around 60% of peak power. [11]

Whilst the consumption of the server a system is run on cannot be controlled – there are methods, such as using an auto-scaler, to improve how efficiently a cluster utilises resources. [9]

A Kubernetes auto-scaler is responsible for controlling how the available resources in a pod increase and decrease. This can be done in one of three ways; cluster scaling to increases the number of nodes on a cluster; vertical scaling, which is done at the pod level to adjust the CPU and memory limits of pods in a node; and horizontal scaling – the method this paper focussed on – which matches demand by increasing or decreasing the number of pods available to a node. Each of these scaling techniques aims to optimise the number of resources available to the cluster to ensure it is operating as efficiently as possible. [9]

The default method for these auto-scalers is, by definition, reactive. Traditionally, rule-based scaling is employed – this is a method which uses a static metric, like CPU utilisation, defined in the Controller. This is done so that, once this metric is exceeded, more resources can be provided to that node. However, the problem with this reactive method, is that it means that the correct number of resources are only available to the node after the threshold has been released. This threshold leads to a temporary bottle neck in performance until the additional resources can be deployed. [12]

1.2 Literature Review

One solution to this problem is to conduct scaling based on how much will be utilised in the future, rather than basing this on how much of the CPU is being used at that time. A proactive approach to this is the use of machine learning in order to improve the efficiency of Kubernetes, and a multitude of research groups have investigated this. A study by Dixit, used an LSTM model to predict future resource usage for a Vertical Pod Auto-Scaler, and, in doing so, they found resources could more efficiently be allocated. The utilisation of CPUs was reduced by 2% and memory utilisation by 22%. [13]

However, studies have also shown that relying solely on basic metrics such as CPU utilisation can be inefficient and provide a misleading representation of resource usage. For example, an investigation conducted into exploring neural network accelerator bottlenecks highlights that GPU utilisation alone may not accurately reflect GPU usage. This emphasised the need for a more nuanced approach to performance metrics. [14] This was reflected by Cu & Son, who suggest that CPU utilisation alone

does not consider the diverse resource demands of different applications. Making it insufficient to be used in isolation for scaling decisions. [15]

Another previous solution was to analyse several time-series forecast models and evaluate their ability at predicting sales data [16]. The SARIMA model performed the best for 47% of the data sets and was the most consistent in its results. A key conclusion from this study being that the effectiveness of models heavily depends on the characteristics of the specific time-series data, so no one-size fits all model should be assumed. [16] This is further demonstrated in Chayama's investigation [17] of univariate and multivariate time series predictions using artificial datasets. This concluded that univariate predictions were more accurate over a short-term, and multivariate over a longer term. Again, showing the significance of not assuming a singular model will be appropriate for all applications and complexities of data sets. [17]

Whilst the selection of the model is important, the configuration can be equally as significant [18]. Taylor's study found that for very short forecasts, the more complex methods, such as ARIMA, significantly reduced the forecast error compared to methods using a simple historical average. However, when extended over 7 days these methods saw a high Mean Absolute Error (MAE), compared with more simplistic models such as exponential smoothing and dynamic harmonic regression. [18]

Most significantly to this project, Toka [19] used a combination of four different models in a horizontal pod Auto-Scaler. In doing this, the models will compete against each other, and the system will then select the best performing to predict future demands. As well as predicting web traffic, this study created another model to understand how different applications will consume resources, enabling more a precise strategy as it mapped the expected number of requests to the required number of resources. It was found that across all configurations, whilst usage increased 1.8-9%, lost requests reduced by 22-72%. This demonstrated that, despite the cluster using higher pod minutes, the resources were used more efficiently overall. [19]

Overall, the examples in literature have demonstrated the previous inefficiencies in Kubernetes' Auto-Scalers, and how time series machine learning models are capable of improving this – up to 2%. As well as this, Toka's method [19], which involved both horizontal auto-scalers and an application profile, achieved a reduction in lost requests.

This demonstrates the importance of integrating machine learning into auto-scalers for the purposes of increased efficiency, and this project aimed to utilise, and demonstrate, how this impact on efficiency, due to proactive scaling for future demand, can be taken further in order to reduce energy consumption. As these examples in literature have not previously had the topic of reduced consumption of energy at the forefront of their research aims.

Project Aim

A proactive scaler would look forward at future demand to adjust resources before they reach critical thresholds avoiding bottlenecks and delays in resource allocation. It also ensures that, once this demand reduces, the number of resources allocated are reduced and any overallocation is removed. This would ensure no more energy is consumed than necessary. To be truly effective it is important that the scaling decisions consider the unique resource consumption of an application.

The aim of this project is therefore to create and train a machine learning model to accurately predict requirements of resources based on data and trend recognition. Then to implement it into a custom Kubernetes horizontal pod auto-scaler, design to scale based on the individual resource requirements of an application to reduce the energy it consumes when hosting a running deployment.

The effectiveness of this custom auto-scaler will largely depend on how accurately future demand can be predicted. As a result, the selection of the correct model will be vital to maximise results. To forecast the future based on historical data a time series model will need to be used so that is what we will focus on using in this study.

Chapter 2
Methods

The aim of development was to achieve building three components: a dataset, an architecture, and a series of tests.

The dataset needed to contain enough data to both train the time-series model and define a model of website traffic. This was in order to be able to evaluate how both the new proactive auto scaler and the standard HPA handled load.

The architecture involved the designing of a system that utilised a time series model, as well as an application profile, to find, and scale, to the optimum number of resources available to an application at any current moment and scale.

Finally, the series of test cases needed to be defined to best cover a full scope of scenarios which an auto scaler could experience. As well as establish what metrics could be recorded to best evaluate its ability to meet its objectives.

2.1 Data

Data Used in Literature Review

Creating a time-series model that can effectively forecast, requires a dataset to train it. There are several commonly used data sets to test web applications. Gosiewska analysed the WorldCup98 dataset. [15] This commonly used data set contains the requests per second recorded for the event's website every day for 3 months producing over 1.3 billion datapoints. Toka [19] initially explored two data sets: the NASA-HTTP logs and the WorldCup98 logs, covering HTTP requests over two and three months respectively. But concluding that these were both too out date, instead choose to trace anonymised web traffic to Facebook on the university campus. Other investigations also chose to create their own dataset. Chiorescu conducted a load test on a Kubernetes cluster for 6 hours, periodically extracting metrics from this cluster every two minutes, this created a dataset of 300 data points. [20] Similar to this, Dixit Created a data set using the Kubernetes Metrics server to record metrics such as CPU and Memory utilisation for the node, number of pods each day over the course of 4 months. [13]

For the purpose of this investigation, the WorldCup98 [21] dataset was used. There are a few reasons for this, as utilising an established dataset presented a few advantages. The first being credibility – an established dataset like WorldCup98 has been widely used as a representation of traffic to load test web applications. This means it has been scrutinised and identified as a reliable representation of what real life user traffic looks like. If a new data set was used, it would have risked providing a solution without a solid foundation in comparative research.

This data set in particular provided plenty of insight into user behaviour. It was created as a large study in the patterns of load on the website throughout the months leading up to, during and following the World Cup of 1998 (of which the site was created for). Matches for this event were scheduled at fixed times and as a result people looking to find out team line-ups and match scores flocked to the website, increasing traffic dramatically whilst they took place.

The resulting data, therefore, displayed regular substantial spikes and troughs in website traffic periodically throughout the day. It presented a solid representation of seasonality that many websites observe. In turn, making it a perfect representation of predictable, event-driven web traffic dynamics. Therefore, it was an ideal choice for evaluating the performance of how a system like a Kubernetes autoscaler handles resources.

However, some filtering of the dataset was required to prepare it for this investigation. A study by Arlitt [21] conducted a research investigation to characterise the workload on the website for the three months it was active. In this, as shown in Figure 1, it was found that in the months surrounding the event the recorded traffic was significantly lower. The number of queries had a significant increase between 10th of June and 12th of July (the start and end date of the event). It was also observed that as the event is a "knock out" competition, the number of teams involved in the event decreases as it progresses. This meant that the regular spikes in traffic throughout the week became sparser as the number of games being played reduced.

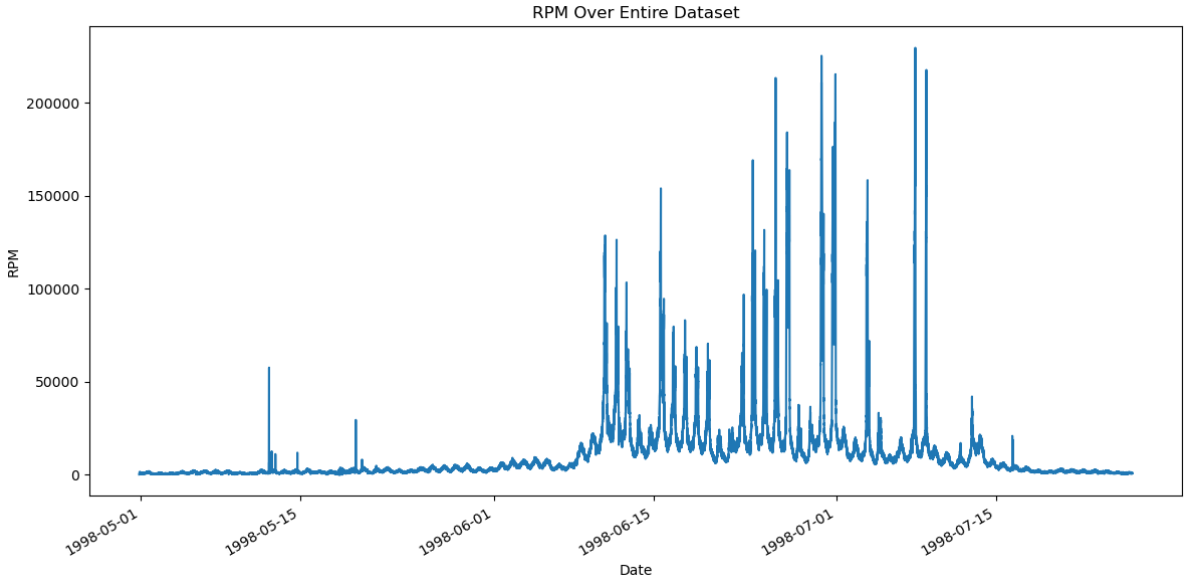

Figure 1 – Website Traffic Experienced on WorldCup98 Website between 1/05/98 and 26/07/98 (Appendix B.1)

Web traffic on weekends was also lower compared to weekdays, despite more matches being played. Arlitt attributes this to people's preference for watching the matches on television during weekends. [21]

Ideally, the time-series forecast model would be trained on the entire WorldCup98 dataset. However, as previously outlined, there are some significant irregularities that would negatively impact the effectiveness of the model at making predictions. Whilst irregularities are to be expected in any dataset, the limited proportion of regular seasonality meant it was sensible to isolate the training, and testing, to 15th -20th June, with training taking place Monday-Thursday and Friday acting as the Testing data. This produced 5760 data points to train the model on, whilst maintaining a good representation of seasonal website traffic is used fit the model.

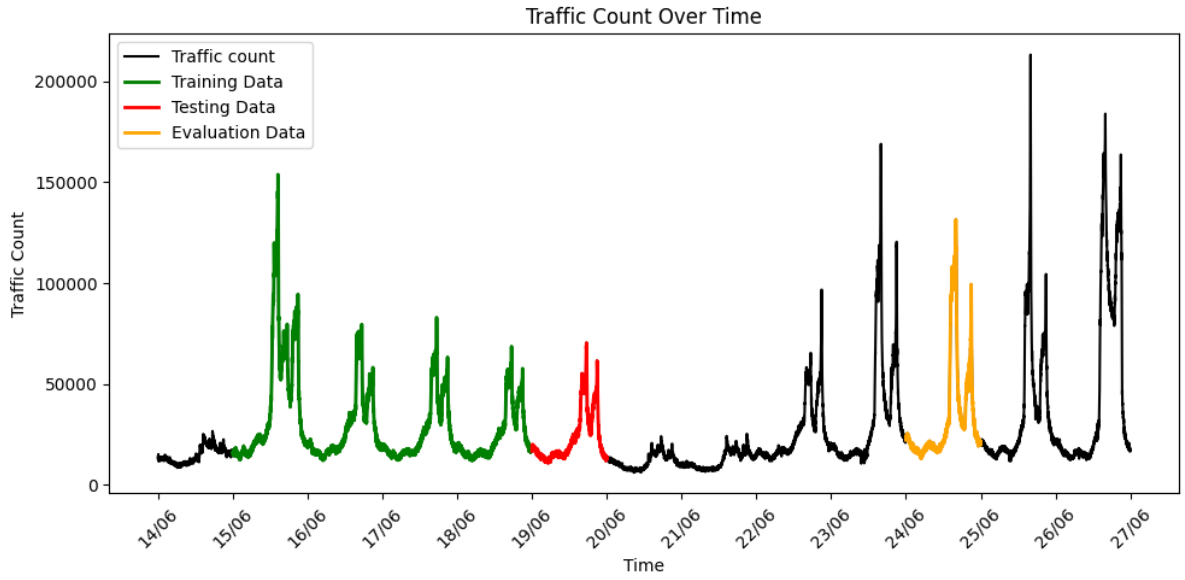

Figure 2 -Visualisation of Data that made up the train, test and evaluation data sets. Black – Data set as a whole. Green – Data to train the model. Red – data to test the model. Yellow – data to evaluate model. (Appendix B.1)

2.2 Design

The solution itself was broken down into 3 tasks. The first was to create a model to be able to predict the amount of traffic coming into the system. This allowed the cluster to make decisions on how to scale pods proactively. The second was a model that provided a mapping between the amount of traffic on the webapp and the optimum number of resources required to handle that load. This was where the focus on energy consumption was. The third was to create an architecture to bring all three of these together, which would be able to use both models to regularly produce a optimum number of pods for the web-app deployment to scale to.

Because of the complexity that such a solution could incur, ensuring that the system was designed in as a modular architecture is crucial to ensure its viability as solution in a variety of different applications.

2.2.1 Model

A time series model is any model that uses historical data to provide a future forecast. The model to be used in this investigation was a Long-Short-Term Memory model (LSTM).

An LSTM is a type of Recurrent Neural Network (RNN)[22]. This is a type of artificial neural network designed to recognise patterns and trends in timeseries of sequential data. They process one input at a time and use it, together with a history of previous input to make a prediction n-steps into the future.

The strength of an RNN comes from its ability to use both the most recent input as well as past information to produce its output. However, they are limited by the vanishing gradient problem [22]. This was where the model struggles to remember long term trends in the data - the further away the information was from the present the less impact it has on the output.

The LSTM model was designed to fix this issue. As shown in Figure 2, an LSTM was made up of three structures called gates. The Forget gate: decides what information was no longer important to

keep in the memory. The input gate: decides what part of the input was valuable and should be stored. The Output gate: decides what the output should be based on the actions taken by the input and forget gate. This made them ideal for handling large amounts of time-series data.

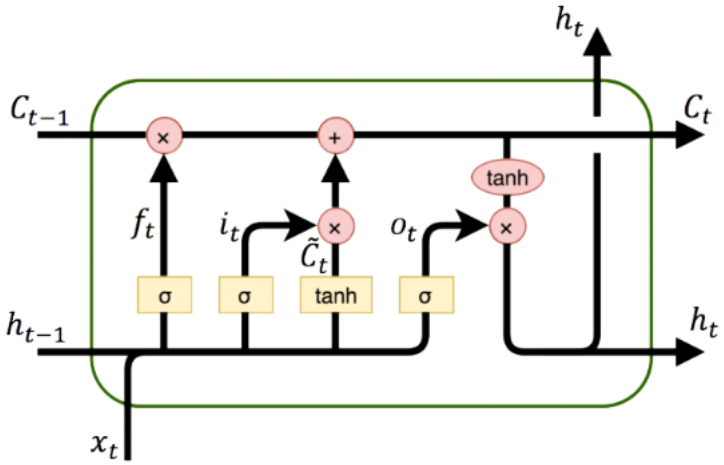

Figure 3 - Basic architecture of an LSTM model. f_t is the forget gate, i_t is input gate, o_t is output gate. [23]

To get the most out of any model a level of optimisation is required to make sure the model is configured to work best with your specific dataset. The LSTM model was implemented using TensorFlow's [24] LSTM library. For this library, the LSTM model had a number of attributes can be configured: LSTM units, batch size, epochs, look-back window, and learning rate. To optimise these hyperparameters, grid search was implemented [25] . The model was trained and tested at different configurations, which ultimately chose the configuration with the lowest RMSE [26] value.

2.2.2 Application Profile

Where one model was used to predict future data, this did not directly inform the optimum number of pods to scale to. In the same way a Machine learning model must be configured for specific data, there is no one size fits all for model for how all web apps consume resources. For this, a separate model needed to be developed to map between the amount of traffic the web app received and the number of pods that should have be deployed.

An application profile is a representation of how an application consumes resources. Wen et al. ran a study exploring cloud-based computations to optimise the energy use of mobile devices [27]. As part of this exploration, they developed application profiles for different apps. These profiles took key characteristics such as input data size and deadline for task completion to create energy consumption models for executing locally and on the cloud. They found that creating these models by understanding the specific demands of their application they could accurately decide when to offload the app to the Cloud.

This investigation aimed to produce a similar model to precisely define at what load to change the amount of resources available to a web application. This approach was closer to that conducted by Toka [19]. The way the application profile is developed in this study differs slightly. Toka [19] mapped the relationship between traffic and number of pods by setting up a web app with a standard HPA. They then ran benchmark tests at varying levels of load (requests per second) for a target number of pods. The test would slowly increase the load until the HPA scaled the number of pods past

the target quantity. This created a profile to map the number of resources the application requires at different load.

The goal of this study was to reduce the amount of power consumed by a web app. As a result the application profile was modelled with that in mind. Similarly to Toka's method, the web app was isolated to its own node with end points to the application exposed by a Load Balancer. [19]

However, a HPA was not used. Instead, for each test the deployment was scaled to a fixed number of pods (1-10). Using k6 (Appendix B.3), a series of load tests were then conducted where for different stages of load ranging from 100 QPS to 2250. Each stage lasted 4 minutes followed by a 6 minute cool down to ensure the behaviour of the pods remains consistent across each stage of load. Once the 4 minutes of load are complete the metric scraping tool Prometheus [28] was used to collect the average CPU utilisation of all of the pods in the deployment, and k6 was used to measure number of failed requests. Once a round of tests for a specific number of pods was completed, the script increased the pod count by one and repeated the process,. This, thereby, systematically explored how the system handled increasing loads with different numbers of pods.

To create the profile itself, the data collected in the tests was analysed to discover which number of pods consumes the least energy at each stage of load. Using equation [29].

$$\text{Consumption (kWh)} = \left(\frac{\text{vCPUs} \times \text{TDP} \times \text{CPU utilisation} \times \text{hours}}{1000}\right)$$

Equation 1- Equation used to calculate energy consumption of a virtual machine (node) [29]

2.2.3 Proactive Autoscaler

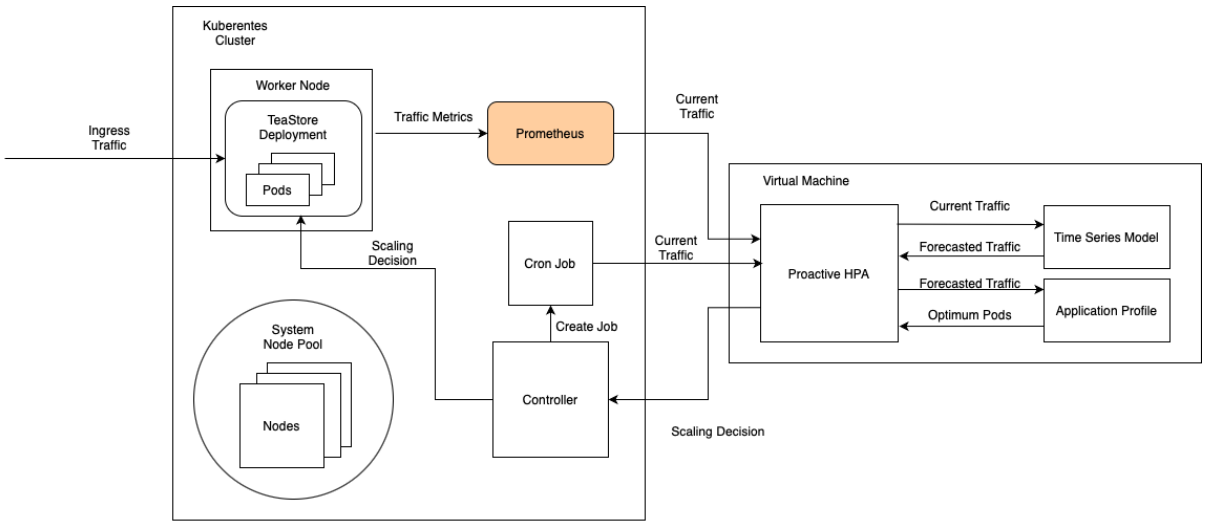

Figure 4 – The Architecture of the System

Figure 4 above shows the architecture of the system. The time series model and the application profile are created and built into a framework that utilises them both to proactively scale the deployment.

The system was made up of a cluster containing two node pools. One node pool was be used as a worker node. It was here that the web app, and the web app exclusively, was be deployed. As this was where the node was being monitored in the experiments, its resources needed to be exclusively utilised by the web app deployment being tested. The other was a system pool. This contained all of the system and monitoring pods, and abstracted any background computation away from the worker node.

The cluster also contains a Con Job [9]. This repeatedly created Kubernetes Jobs on a pre-defined schedule. In this case, calling an endpoint inside of an external virtual machine (VM).

This VM contained 3 docker images: the LSTM, the application profile and the proactive autoscaler (PA). Where a normal autoscaler would be a deployment on the cluster through the HPA Controller [9], this system had an external HPA Controller (called Proactive HPA) contained in the VM. This external Controller manually forced the deployment to scale depending on the recommendations from the two models. This controller was called every minute by the cron job and in turn, first extracted the total amount of queries the web app has received over the last minute from Prometheus. It then communicated this rate of traffic with the LSTM, receiving back a forecast for the rate of traffic over the next minute. This forecasted number was then provided to the image containing the application profile which returned the optimum number of pods that should be run for that level of traffic.

The external Controller image then instructed the true cluster Controller to scale to that number of pods, simulating the role of a true HPA Controller.

2.3 Experimental Set up

This experiment was conducted to compare the energy consumption of the worker node in the cluster using the Proactive Horizontal Pod Autoscaler (PHPA) against the standard Kubernetes HPA. To do these 4 individual tests on both systems were be conducted to examine their capabilities at a variety of traffic conditions. This, ultimately, evaluated the effectiveness of using a proactive autoscaler instead on a reactive technique.

The default HPA works by periodically monitoring a predefined metric (CPU utilisation, Memory Utilisation etc) and scaling pods to align that metric to a predefined target metric. To best reflect capability of our PHPA, the HPA was configured to scale on CPU Utilisation. In other solutions that compare against a regular HPA, the exact limit that was scaled to varies from 60-80% utilisation. For this project, configuration conducted by Phuc [30] was followed and set the scale target to 80% utilisation.

The 4 tests were taken from 4 hour long periods, from Wednesday 24th June in the WorldCup98 dataset. This date in particular was chosen as it still follows the intraday seasonality that was found in the training-test week, but is also still completely unique outside of that dataset. This ensured that the model was not over-fitted and there was no contamination in the training and validation data.

K6 was again used to induce load on the dataset to reflect the patter of traffic from the dataset. To gain the best insight into the ability of the PHPA, the 4-hour periods were chosen specifically reflect different extremities of load that an autoscaler would need to be able to handle. The test cases, and the reasonings for each, were as below:

- Test case 1: High Variance (Total variance of 986,889 between 20:11-21:11) - Tests how well the auto-scaler reacts to a change in trend.

- Test case 2: Largest decrease (Decrease of 92008.0 between 1998-06-24 16:02:00) - to test how well the auto-scaler scales down.
- Test case 3: Largest increase (Increase of 75447.0 between 1998-06-24 13:49:00) - test how well the auto-scaler can scale up.
- Test case 4: Steady traffic (Total variance of 422,676 between 10:24-11:24) - tests how well the autoscaler handles steady traffic.

These test cases can be seen in the context of traffic count changing over time in Figure 5 below.

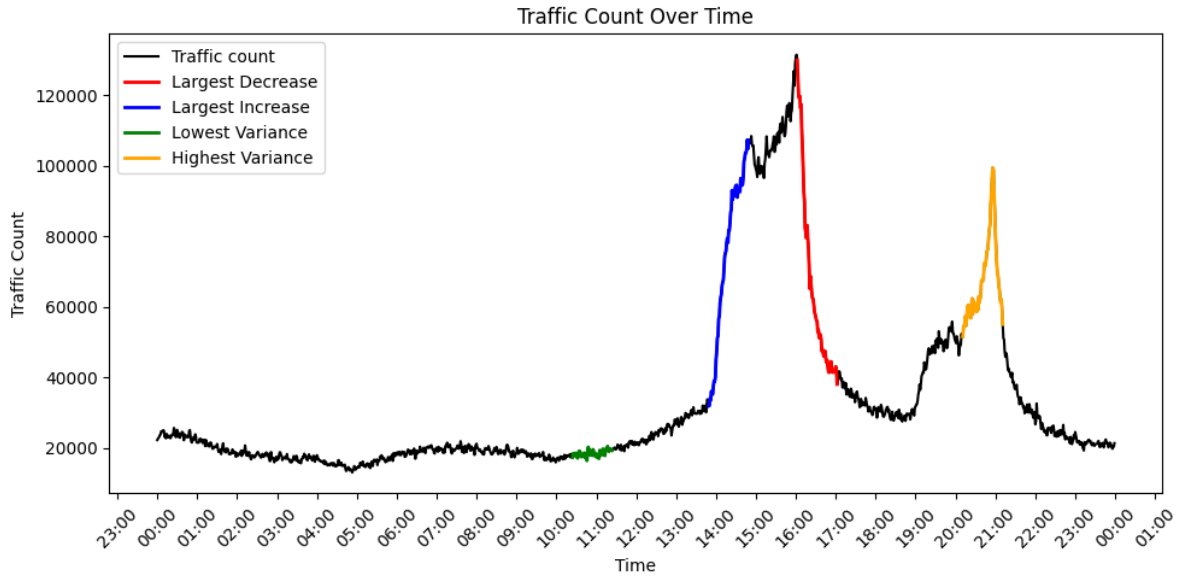

Figure 5 – Visualisation of the data each test case will evaluate – demonstrates how traffic count changes over time, with focus on the time periods selected for the test cases. Green – Test Case 4, Blue – Test Case 2, Red – Test Case 2, Yellow – Test Case 1.

Load was induced for the 20 minutes before each test case to ensure the final results were not affected by any "cold start" effects.

Throughout each test, Prometheus gathered the following minute-by-minute: the CPU utilisation at node level, the number of failed requests and, the average response time.

The CPU utilisation used Equation 1 to derive the energy consumption. This was totalled to derive how much power the node/deployment consumed in the test period. It describes how well the autoscaler managed physical resources, particularly in relation to server load and utilization. Maximizing the utility of each node and pod without over-provisioning.

Lost requests and average response time were combined with total requests to evaluate the effectiveness of the deployment at the replica count (number of pods), that each Auto scaler has configured it in. This followed the RED method (Rate, Errors, Duration) [31] and was an adaptation of the USE method [32] popularised by Brendan Gregg. But, where use is used to access the performance of larger infrastructure, RED is focussed at the microservice level. The methodology behind recording these metrics, specifically, was to expose how healthy the microservices within an architecture are.

Each of these metrics directly affected the quality of service and the user experience. By measuring how many requests were lost during peak times and how long these requests take; it could be

evaluated how reliable the autoscaler was in maintaining service availability and responsiveness, and assessed whether the scaling strategy was effectively preventing bottlenecks and system overloads.

By measuring both energy consumption and the number of lost requests, it can be ensured that the autoscaler not only optimised operational costs, but also maintained high service quality and reliability. Verifying whether the autoscaler could be used in a production environment.

2.4 Project Management

2.4.1 Version Control

Git (Appendix B.2) was used as a repository for the codebase for the project. Commits were clear and concise and outlined what was changed or added, and why this had taken place. Extra descriptions were provided in cases where the commit message by itself was not enough to indicate what actions were undertaken.

2.4.2 Continuous Integration

The philosophy for the development of the project followed small continuous improvements. With constant integration into the minimum viable product. The Process for development went as follows:

1. First, a Kubernetes Cluster was created in Azure AKS and successfully TeaStore deployed on it.
2. A K6 load testing framework was set up and a success test was conducted at constant load on the deployment.
3. Testing scripts were set up to create the application profile data set.
4. The standard HPA was applied to the TeaStore deployment and a test script created to run the 4 different test cases on it.
5. An application profile was developed.
6. The LSTM machine learning model was trained and tested.
7. The application profile and the machine learning model were deployed to azure.
8. The Cron Job was developed to utilise both the application profile and the LSTM model.
9. The Cron Job was deployed onto the Kubernetes cluster.

Chapter 3
Results

3.1 Implementation

Load testing was implemented for two different purposes for this project. The application profile, and to performance test the cluster. The application profile required load levels that were sequentially increased from the minimum to maximum Queries Per Second (QPS) observed in the evaluation dataset. Conversely, the evaluation tests induced load based on the Queries Per Minute (QPM) taken directly from the evaluation dataset. To facilitate both of these different requirements, instead of building individual K6 jobs for each level of load to be tested

This solution was initial configured to define a number of Virtual users who would each make a single http in a looping structure for a predefined duration of time. To simulate more realistic user traffic patterns the http address was taken at random from a list of 5 pages on the TeaStore web app. To attempt to create a constant rate of load sleep intervals ("sleep(1)") were used. This however, introduced unnecessary complexity into the load generation, creating timing errors that produced an inconsistent varying load. The increased complexity also meant that the k6 testing object was hitting a maximum QPS of only 560, almost 4 times below the maximum amount defined for the test (2250). This inefficiency was later reduced upon discovering 'Scenario' feature in k6 [30].

```
export const options = {
  scenarios: {
    constant_request_rate: {
      executor: 'constant-arrival-rate',
      rate: 1000,
      timeUnit: '1s', // 1000 iterations per second, i.e. 1000 RPS
      duration: '30s',
      preAllocatedVUs: 100, // how large the initial pool of VUs would be
      maxVUs: 200, // if the preAllocatedVUs are not enough, can initialize more
    },
  },
};
```

Figure 6 - Example solution of K6 "scenario" framework (Appendix B.3)

This allowed for precise control over load consistency, over a predefined duration. This vastly increased the consistency of the traffic the web app received but still only increased the maximum QPS to 750.

Establishing that the limit of the K6 framework had been reached required a scaling down of the data set by a factor of 3, this was to align the maximum rate of traffic enabled by the K6 system with the maximum load recorded in the dataset.

A second problem faced was with the medium by which tests were run on the cluster. Initially the testing scripts were hosted on a personal machine. This meant that the load generated was first transmitted through whatever WIFI the machine was connected to. It was evident that the strength of this connection had a large influence on the results received. Analysis on the failure rate produced by K6 showed a correlation with the length of the test rather than the amount of load being induced.

To mitigate these issues the testing environment was migrated to a dedicated Virtual machine. This allowed for a consistent connection and isolated the test environment from any influence on the results caused by congestion of the local network as well as any external requests the personal machine conducted outside of the test. This shift in strategy improved the consistency and reliability of the test results.

The modifications to the testing framework, as well as the testing environment, ensured that the experimental setup not only made the testing more effective at conducting a full range of load tests but also maintained a high standard of reliability and reproducibility.

3.2 AI Model

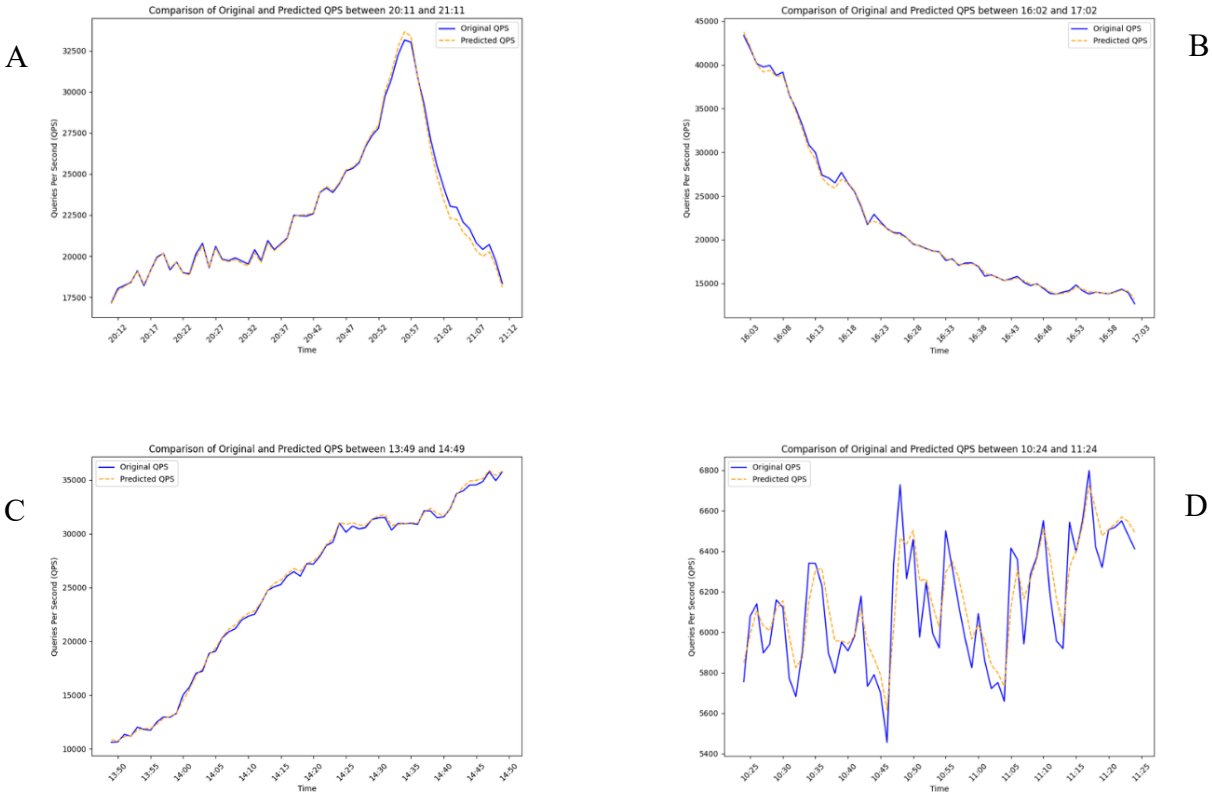

Figure 7 – Demonstrates the performance of LSTM model for each test case. A – Test Case 1, B – Test Case 2, C – Test Case 3, D – Test case 4

The model was optimised using the grid search methodology and the hyperparameters—look_back, units, batch_size, epochs, and learning_rate – were iteratively adjusted to identify the configuration with the lowest error. The optimal configuration was found to be:

- look_back: 15
- units: 30
- batch_size: 16
- epochs: 25
- learning_rate: 0.007

When tested against the Validation dataset the configuration showed a significant improvement in performance, reducing the RMSE against the set from 0.155 to 0.120, an accuracy increase of approximately 23%.

Similar to the PHPA itself, the model was evaluated against the four test periods shown in Figure 7. The RMSE for these four test cases were:

- Largest Decrease (Test case 2): 0.215
- Largest Increase (Test case 3): 0.184
- Lowest Variance (Test case 4): 0.090
- Highest Variance (test case 1): 0.120

Expectedly, the model was most accurate under test period 10:24 and 11:24, the hour with the lowest variance across the validation data set. This highlighted the model was particularly effective at capturing stable trends.

Conversely, the model performed worse under test period 16:02 and 17:02, the hour with the largest decrease across the validation data set. This suggests, that while the model capturesd the trend of the decrease, it might not precisely model the magnitude and could often over estimating the rate at which the QPS declined.

After rigorous hyperparameter tuning, the optimised LSTM model demonstrated substantial predictive accuracy. Showcasing its capability to effectively forecast univariate time series data. This was shown by the consistent low RMSE values across all test periods. Notably, the second-best performance was during the period of highest variance, underscoring the model's robustness at handling fluctuating traffic.

On the other hand, the model's underperformance in periods of large change highlights potential areas for further refinement. This could involve further hyperparameter tuning or adjustments in how the model is trained, which would improve its accuracy in predicting the magnitude of rapid decreases in traffic.

Nevertheless, the overall performance at accurately forecasting future website traffic proved that the more accurately the forecasted traffic was, the more consistently the PHPA selected the optimum amount of resources to deploy.

Once the model was trained and optimised it was deployed via a python flask app. This app was built locally before being containerised using Docker (Appendix B.4) onto an Azure VM. The app exposes the model to the following http endpoint:

http://<Virtual Machine IP address>:5000/

Predicting where each incoming traffic figure is formatting in a JSON packet containing:

```
'current_traffic': requests_per_minute
```

An important part of the web apps deployment was managing this input data to make accurate predictions. As the LSTM was configured to use a 15 minute look back period, a rolling history was required to be maintained by the flask app, saving the last 15 Queries per Minute datapoints. This was essential to make sure the LSTM could predict future demand effectively. As a result, it was also important to ensure that for each test of the Proactive HPA, a 20 minute “warm-up” period was included to allow the traffic history to fully populate and ensure the model could make predictions using a complete set of data.

3.3 Application Profile

The application profile, as outlined in 2.2, was created to show how the TeaStore application utilised resources at varying loads. Multiple tests were run, and each time the number of pods available to the deployment were increased. Despite each of the nine tests taking more than an hour to run, the result provided a complete, and comprehensive, depiction of the resource consumption profile of TeaStore. Furthermore, to help mitigate any anomalous error test to test the model was run 3 times with the final data set used to create the model being the average of all three.

Unfortunately, due to limitations to the amount of load that could to be placed on the deployment by K6, the deployment did not experience a high enough load for failed requests to be a usable quantifier of quality of service (Appendix B.3). As a result, the average duration the HTTP took from request to response was recorded to measure how much speed deteriorated as load was increased. The slowest 5% were excluded from Figure 8, in order to remove the impact any lost requests may have.

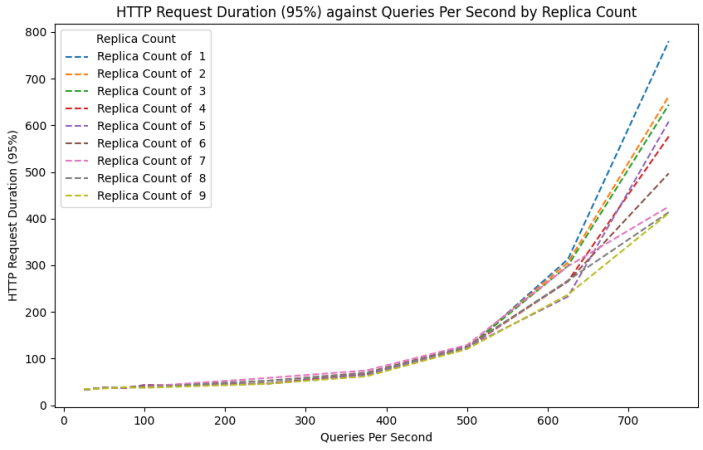

Figure 8 – Relationship between HTTP request duration and the quantity of HTTP requests per second for each Replica count. Demonstrates the rate of change for HTTP request duration in respect of the QPS rate

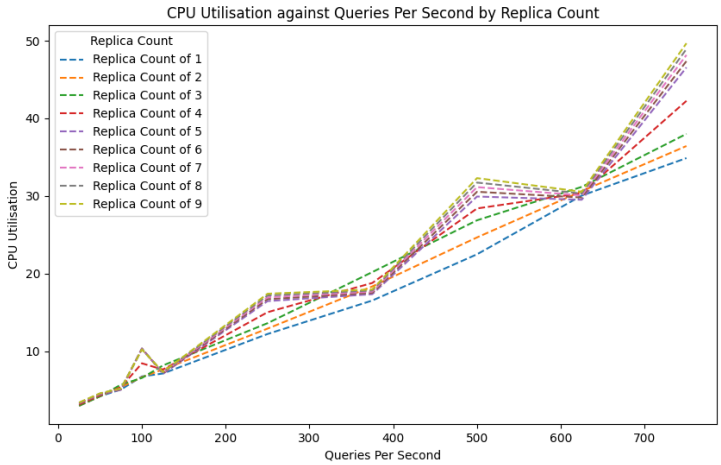

Figure 9 – CPU utilisation change with increased quantity of HTTP requests for each Replica count. Demonstrates the rate of change for HTTP request duration in respect of the QPS rate

Figures 8 and 9 show the correlation between the rate of load (queries per second) and CPU utilisation and Request duration respectively. With Figure 9 showing that CPU utilisation remained mostly linear for replica counts 1-4 as load increases. However, when the deployment had more than five, the CPU utilisation increased in volatility across the range of loads. This suggests there was an approximate minimum level of QPS present before the CPU of pods began to be overutilized. The relationship between HTTP request duration and Queries per second remained much more consistent across all resource ranges up until a load of 500 queries per second.

Once this request rate was surpassed the amount of replicas had a clear impact on the speed in which HTTP requests are completed. The difference in response time deviated heavily as load increased up to 750 QPS. Where the request time of the deployment with nine pods available to it was 90% faster than when only a single pod was available.

Once these tests were complete the optimal pod at each rate of load. As fail rate was not a usable metric, instead of using a minimum success rate for HTTP requests, a minimum request duration ranging from 100qps and 300qps was deployed. The optimum pod count for each request rate was the pod count that produces the lowest energy consumption within this bound.

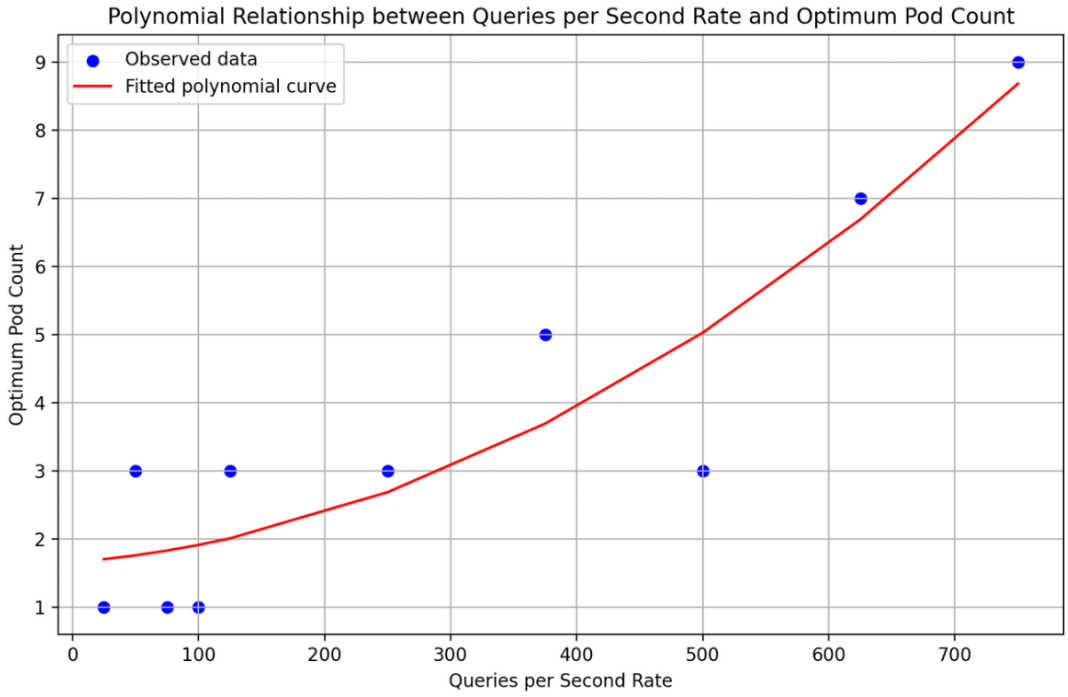

Figure 10 – Change in observed optimum pod count with an increase in HTTP request, as shown in blue. Demonstrates the polynomial relationship between QPS and Pod count to create the application profile, as shown with the red polynomial fit (polynomial regression of observed data points)

The regression model depicted in Figure 10 was then created to map the optimum number of resources to any request rate.

3.4 Standard HPA Results

Figure 11 - Provided here is a visualisation on the behaviour of the Standard HPA in the 4 test cases. Blue – Change in HTTP requests over time. Red – Change in pod count over time. A – Test case 1, B – Test case 2, C – Test 3, D – Test Case 4

Test case 1 - Highest Variance:

For this test, as depicted in Figure 11, it can be seen that for the first 20 minutes of the test, the request rate remained consistent and the standard HPA maintained a pod count of seven to reflect this. However, as soon as the request rate began to increase the HPA, the pod count increased rapidly, hitting the maximum amount of replicas permitted to that deployment within 5 minutes – at a request level of 24000 QPS. The number of pods also did not decrease when this load eventually subsided.

Test case 2 - Largest Decrease:

The same response to decreasing load found in Test Case 1, was repeated here. Despite the request rate decreasing right from the beginning, it was only until 16:38 (30 minutes into the test) that the replica count began to subside. Not only was the change late but it was also sudden. From when the HPA first began to slow down, it only took 15 minutes for a replica count of 6 to be reached, where it the stayed finding a steady CPU utilisation.

Test case 3 - Largest Increase:

This test case presented the largest increase in the requests per minute over the hour period. Here the HPA scaled at its fastest rate increasing its replica count by 13 in just over twelve minutes. Reaching the peak pod count at just 13000, however, is significantly lower than that shown in Test Case 1. This

indicates that a large spike in traffic could course a constant rate of increase from the standard HPA until the CPU utilisation eventually settles. In this case, such a normalisation did not occur for another 10 minutes. Furthermore, this decrease took place whilst the request rate was rising, which indicates that the cause of such a large spike in pod numbers was closer attributed to the HPA ability to handle increase in load, rather than the amount of load itself.

Test case 4 - Lowest Variance:

This test was used to measure how well the autoscaler handles a constant level of resources. As shown in Figure 11, the standard HPA showed little sensitivity to the small fluctuations in request rate throughout the test. Maintaining a replica of 3 and 2 throughout.

3.5 Proactive HPA Results

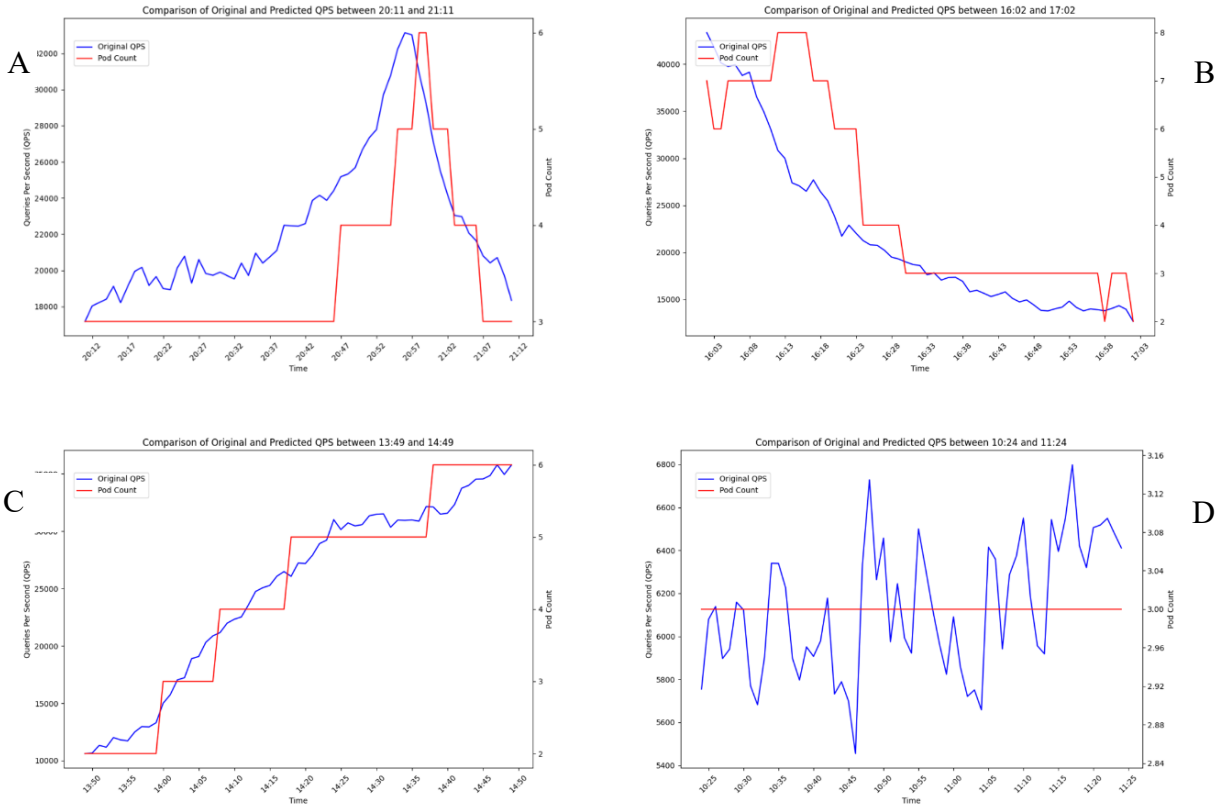

Figure 12 - Depiction of the scaling behaviour of the proactive HPA in all four test case. Blue – Change in HTTP requests over time. Red – Change in pod count over time. A – Test case 1, B – Test case 2, C – Test 3, D – Test Case 4

Test Case 1 - Highest variance

In Figure 12, it can be seen that the amount of pods available to the deployment was a much closer reflection of the rate of requests induced. It scaled slowly as load increased, only reaching its highest pod count just after the test reached its maximum load.

Test Case 2 - Largest Decrease

Although a small increase in the replica did take place; it lasted for only five minutes and increased the pod count by one. After this, the amount of pods decreased rapidly to catch up with the falling request rate, with the number reaching its low of 2 after just minutes. It maintained this level for the last 30 minutes of the test.

Test Case 3 - Largest Increase

This test, once again, displayed the accuracy of the proactive HPA. The number of pods available showed a much closer relation to the number of queries it was experiencing and it was also not sensitive to any sudden increases in load, as shown with the display of a consistent uptrend.

Test Case 4 - Lowest Variance

As the request rate in the test case was almost constant, the resources provided by the proactive HPA remained constant too. Maintaining a pod count of 3 throughout the entire test.

	Standard HPA	Predictive HPA		Standard HPA	Predictive HPA	
Test Case	Average HTTP 95% Duration	Average HTTP 95% Duration	% difference from HPA	Total Energy Consumption	Total Energy Consumption	% difference from HPA
Lowest Variance	38.67 ms	53.69 ms	32.52	21.31 kWh	17.73 kWh	-18.34
Largest Increase	737.15 ms	825.49 ms	11.31	112.91 kWh	55.24 kWh	-68.61
Largest Decrease	4179.00 ms	4479.00 ms	6.93	123.69 kWh	61.32 kWh	-67.43
Highest Variance	633.03 ms	717.03 ms	12.44	104.04 kWh	62.45 kWh	-49.95

Table 1 – Comparison of the HTTP duration and Energy Consumption for the standard and proactive HPA a cross each of the four test cases

Chapter 4
Discussion

This chapter looks at the trends and patterns aforementioned and outlined. As well as this, it evaluates the success of the project, whilst also ensuring to highlight areas would require further research and improvement, were the investigations to be taken further in the future.

4.1 Energy Consumption Conclusion

The AI model was used to predict future website traffic extremely well and was an important feature of this project, as it meant the auto scaler could assign resources prior to when they were needed. The model itself made predictions to a high standard of accuracy, with an RMSE number as low as 0.090, demonstrating success. It is important to note, that only one model was tested and used, however, due to the extent of the accuracy proven with the RMSE numbers, the conclusion to proceed with this one was well evidenced. The model was therefore accurate and ensured optimal pod values provided by the application profile were accurate to the amount of traffic that application was about to receive. The application profile informed the proactive auto scaler on how many pods to scale to, and therefore the scaling decisions reflected how this application uses resources at different loads. From the application profile, a polynomial function could be created to describe the optimal number of resources to provide the app at different loads, and when used with the autoscaler it saved a large amount of energy, further demonstrating the success of this section of the project. Although, the application profile is only applicable to the exact machine the application is deployed on. This means if the Kubernetes crashes or the node is not available, the deployment cannot be transferred to run on another node. However, this is one of the core reasons for the accuracies of this system as a whole and therefore, cannot be considered a negative factor.

With regards to the use of a proactive auto scaler, such as the one designed in this project, as opposed to a standard autoscaler. The results made clear that a proactive autoscaler has many benefits that the standard auto scaler did not provide.

Under the test cases, which provided a rapid increase in traffic, the amount of pods a standard auto scaler scheduled would continuously increase to the maximum limit of 15. This was not only inefficient but did not reflect the load on the system. By implementing the application profile and AI model, they worked together to ensure the amount of pods scheduled was a good reflection of the load on the system.

This inefficiency of the standard HPA can be seen through test cases 1 and 3, where it was seen to vastly over allocate resources when load increased. This can be attributed to two factors – a delay between when a pod is allocated and when it can start taking requests, and when CPU requests by the pod initially increased in the first few minutes after allocation. These two factors will have caused the overall percentage utilisation of the deployment to stay above the scaling threshold and have caused the standard HPA to request even more resources. This led to a snowball effect, where the number of pods allocated to the deployment constantly increased, until the maximum limit was achieved. This supports the findings of Gleeson, suggesting that CPU utilisation by itself is not an effective way to scale resources [14].

This conclusion was further supported by the behaviour of the autoscaler in the proactive HPA. The scaling strategy utilised the relationship between multiple resources that were configured to be bespoke to the application. As a result, the number of pods deployed to the system better reflected the load that was applied to it and so, throughout the tests it could be seen that the predicative autoscaler deployed pods much more efficiently. Consequently, the use of the predicative autoscaler resulted in the amount of energy consumed under the same conditions being just over 50% less than the standard HPA. It is important to note that, whilst energy efficiency increased, the request duration increased by

15%. However, it is possible that this could be improved by further optimising the application profile in order to find a configuration that reduces energy consumption, without impacting HTTP request duration.

Overall, the aims of this project were met with great success. The proactive HPA was successfully built, and with efficiency that resulted in a 50% reduction in average energy saved. Although it is important to understand that this proactive HPA also increased HTTP request duration by the 15%, in comparison to the money and energy saved by this system, it is with fair judgement to conclude this outweighs the time spent. Overall, the proactive HPA, and the application profile and AI model used for it, were very successful and achieved what was set out to be.

4.2 Ideas for future work

In terms of improving this project if it were to be continued, there are a few ideas that could be implemented. The first is to use a better infrastructure to increase the amount of load that can be induced. One option is to use multiple k6 tests concurrently to remove the limitation that the tool has on the maximum requests per minute.

The next improvements centre around the use of Machine Leaning models in future methods. For example, cross validation could be used to improve the training of the models, as well as comparing the performance of different models. Whilst LSTM performed well in this investigation, as previously mentioned, there is not one-model-fits-all solution. The modular nature of the framework created in the study would make investigation into which type of model is optimal even easier. Further investigation could also be done on how far forward is most efficient. Analysis into start-up time of pods could mean looking further into the future provides more efficient scaling.

A final idea for future work is to incorporate a more accurate energy modelling, as the equation used to model the amount of energy consumed is still based around CPU utilisation. A more accurate model would consider other things like memory and networks and a more accurate depiction of the amount of energy consumed could uncover nuances in how a deployment consumes energy instead of just how it consumes CPU time. This could, in turn, create a completely different application profile.

List of References

1. RightScale. *State of the Cloud Report.* [no place]: Flexera, 2019. [Accessed 9 May 2024]. Available from: https://resources.flexera.com/
2. Columbus, L. *Roundup Of Cloud Computing Forecasts And Market Estimates, 2014.* [Online]. 2014. [Accessed: 9 May 2024] Available from: https://www.forbes.com/
3. Gartner. *Gartner Forecasts Worldwide Public Cloud End-User Spending to Reach $679 Billion in 2024.* [Online] 2023. [Accessed 9 May 2024]. Available from: https://www.gartner.com/
4. Belkhir, L. and Elmeligi, A. *Assessing ICT global emissions footprint: Trends to 2040 & recommendations.* 2018, **177**, pp.448-463
5. Hamilton, J. *Cooperative Expendable Micro-Slice Servers (CEMS): Low Cost, Low Power Servers for Internet-Scale Services*. 2009. [no pagination]
6. Jobin, P. *Cloud Computing Shift to Cooler Climates*. [Online]. 2012. [Accessed: 9 May 2024] Available from: https://www.datacenterknowledge.com/
7. Kahney, L. *Apple and Google can help power the switch to renewable energy*. [Online]. 2018. [Accessed: 10 May 2024] Available from: https://www.wired.com/
8. Hassan, H, B. et al. Survey on Serverless computing. *Journal of Cloud Computing*. **10**(39) [no pagination]
9. Sayfan, G. *Mastering Kubernetes. (Second Edition)*. Birmingham: Packt, 2018.
10. CastAI. *2024 Kubernetes Cost Benchmark Report*. [Online] 2024. [Accessed: 13 April 2024] Available from: https://cast.ai/
11. Barroso, L.A. and Holzle, U. The Case for Energy Proportional Computing. *Computer*. 2008, **40**(12), pp. 33-37
12. Ju, L. et al. *Proactive Autoscaling for Edge Computing Systems with Kubernetes.* [Online]. 2021. [Accessed: 13 April 2024]. Available from: https://arxiv.org/

13. Dixit, A. et al. Machine Learning Based Adaptive Auto-scaling Policy for Resource Orchestration in Kubernetes Clusters. *International Conference on Internet of Things and Connected Technologies.* 2022, **340**, pp.1-16.
14. Gleeson, J. *RL-Scope: Cross-Stack Profiling for Deep Reinforcement Learning Workloads*. [Online]. 2021. [Accessed: 13 April 2024]. Available from: https://arxiv.org/
15. Cu, H. and Son, N. The effect of the resource consumption characteristics of cloud applications on the efficiency of low-metric auto scaling solutions. *International Journal on Cloud Computing Services and Architecture*. 2018, **8**, [no pagination]
16. Shah, V. A comparative study of univariate time-series methods for sales forecasting. *International Journal of Business and Data Analytics*. 2022, **2**(2), 187
17. Chayama, M. and Hirata, Y. When univariate model-free time series prediction is better than multivariate. *Physics Letters A*. 2016, **380**(31-32), pp.2359-2365
18. Taylor, J. W. A Comparison of Univariate Time Series Methods for Forecasting Intraday Arrivals at a Call Center. *Management Science*. 2008, **54**(2), pp. 253- 265
19. Toka, L. et al. Adaptive AI-based auto-scaling for Kubernetes. In: *International Symposium on Cluster, Cloud and Internet Computing, May 2020, Melbourne, Australia*. Melbourne: 2020, pp.599-608.
20. Chiorescu, R. and Djemame, K. Scheduling Energy-Aware Multi-Function Serverless Workloads in OpenFaaS. In: *Proceedings of the 16th IEEE/ACM International Conference on Utility and Cloud Computing, Dec 2023, Taorima, Italy.* New York: ACM, 2023
21. Arlitt, M. and Jin, T. Workload Characterization of the 1998 World Cup Web Site. *IEEE Network.* 2000, **14**(3), pp.30-37.
22. Hochreiter, S. & Schmidhuber, J. Long short-term memory. *Neural Computation*. 1997, **9**(8), pp.1735-1780.
23. APMonitor. *Long Short Term Memory Networks*. [Online] 2022. [Accessed 13 April 2024] Available at: https://apmonitor.com/

24. Lamurias, A. et al. Bo-lstm: classifying relations via long short-term memory networks along biomedical ontologies. *BMC bioinformatics.* 2019, 20(10), ISSN 1471-2105

25. Ghawi, R. and Pfeffer, J. Efficient hyperparameter tuning with grid search for text categorization using knn approach with bm25 similarity. *Open Computer Science*. 2019, **9**(1), 160-180.

26. Forkuor, G. et al. High resolution mapping of soil properties using remote sensing variables in south-western burkina faso: a comparison of machine learning and multiple linear regression models. *Plos One*. 2017, **12**(1), e0170478.

27. Wen, Y. et al., Energy-optimal mobile application execution: Taming resource-poor mobile devices with cloud clones. In: *Proceedings IEEE INFOCOM, 2012, Orlando.* USA: IEEE, 2012, pp.2716-2720.

28. Vishwakarma, G. et al. *Metrics for benchmarking and uncertainty quantification: quality, applicability, and a path to best practices for machine learning in chemistry*. [Online]. 2021. [Accessed 6 May 2024] Available at: https://arxiv.org/

29. Bergman, S. How can I calculate CO2EQ emissions for my azure VM? 15 February, *Microsoft Sustainable Software,* 2021. [Online]. [Accessed 15 April 2024] Available from: https://devblogs.microsoft.com/

30. Phuc, L.H. et al. Traffic-Aware Horizontal Pod Autoscaler in Kubernetes-Based Edge Computing Infrastructure. *IEEE Access*. 2022, **10**, pp.18966-18977.

31. Dam, J. The Red Method: How To Instrument Your Services. 2 August, *Grafana Labs,* 2018. Grafana Labs. [Accessed 15 April 2024] Available at: https://grafana.com/

32. Brendan Gregg, B. Thinking methodically about performance. *ACM Queue*. 2012, **10,** 12.

Appendix A
Self-appraisal

A.1 Critical self-evaluation and lessons learned

The project proved successful in achieving its aim of reducing the energy consumed by a deployment on a Kubernetes cluster, however, it was not without its challenges.

Before starting the project, I had no prior experience with Kubernetes, and had not come across what they were, how they operated or how they were used. So, despite creating an initial plan as to how to approach the project, there were many things that I had yet to fully understand. This included how to deloy an application as well as control the amount of resources it had access to - these were unknown factors that caused me to be both delayed in my progress and to change my approach to the project.

As a result, if I were to approach a project in the future with this limited foundation of knowledge, I would make ensure I dedicated a longer learning period before beginning software development.

At the start of the project, I was unaware of the time that would be devoted even solely to running the tests. To be able to create an Application Profile, each pod needed to be tested at 10 different levels of loads, and to ensure the measured metrics were not affected by the change in load itself, each stage took five minutes took allow the pods to stabilise. This meant that each full test took 50 minutes as well as a 10 minute cool down period at the end to return the pods back to idle before starting the next test. As a result it would take 9 hours to create a new application every time I made changes to the test or cluster as it would make any previous application redundant.

This was not only a very time consuming aspect to my project, but in it also hindered any further developments I could make alongside the tests that were running. Because developing the proactive autoscaler required automatically scaling the TeaStore deployment, for the 9 hours the Application Profile was generating, I was not able to develop the proactive autoscaler without invalidating the results. This made time management very difficult and so, to approach this difficulty with hindsight, I would set up two identical clusters if I were to repeat this investigation. This would allow me to run tests without stopping the development of the proactive autoscaler.

Despite these issues this project has been an excellent learning experience for me. I have vastly increased my understanding of Kubernetes, how they work, and their capabilities involving the management of deploying real world applications. Although I have undertaken a number of software projects, none of them have directly involved the infrastructure that an application is deployed on. Getting to understand how the configuration of this infrastructure can effect the performance of the application deployed within it has been very insightful, and will be something I will apply to many projects in the future.

Utilising Machine learning in this project was also something I thoroughly enjoyed. Despite the intricacies of how these models operate being outside of the scope of this project, being able to observe the accuracy they can produce after a small amount of configuration was very satisfying, and has definitely opened my eyes to the capabilities of Machine learning in real world applications.

A.2 Legal, social, ethical and professional issues

A.2.1 Legal issues

Kubernetes is an opensource system, licensed under Apache 2.0. This licence means other developers can use and modify the system without restriction or permission from the authors.

The data used in this study was all from an open source database of anonymous http requests. As a result, there are no major legal concerns for the dataset. The data collected to build the application profile was recorded and generated by myself so also does not present any Legal issues.

A.2.2 Social issues

There are no major social issues with this project. The system is deployed on the cloud on a small scale, so the total energy consumed by the development of the system was negligible.

A.2.3 Ethical issues

The application Profile is configured based on the deployments use of resources on a specific machine, so the application is only permitted to run on that node. As a result, should the node fail, the application will not be able to be scheduled to another node in the cluster. This could compromise the reliability of the application if this solution was used in a production environment without proper fault tolerance procedures in place.

A.2.4 Professional issues

No professional issues arose throughout the completion of this project. All external code and literature used in this project is referenced and credited.

Professional software practices were employed throughout, using a git repository to maintain version control for the codebase as well as using consistent naming conventions.

Appendix B
External Materials

- B1 WorldCup 1998 Data set was used throughout to resents web traffic to a website
    - https://ita.ee.lbl.gov/html/contrib/WorldCup.html#:~:text=WorldCup98&text=This%20dataset%20consists%20of%20all,the%20site%20received%201%2C352%2C804%2C107%20requests.&text=The%20access%20logs%20from%20the,in%20the%20Common%20Log%20Format.
- B2 A Git repository was used the code repository
    - https://git-scm.com/doc
- B3 K6 was used as the main testing tool to load test the deployment.
    - https://k6.io/blog/how-to-generate-a-constant-request-rate-with-the-new-scenarios-api/
- B4 Docker was used as the container management tool throughout my project
    - https://docs.docker.com

Appendix C
Application Profile Dataset

replicas	virtual users	node response	failed rate	http reqs	http req duration (95%)
1	25.0	2.9730	0.0	7500	33.0209
1	50.0	4.2167	0.0	15000	37.7759
1	75.0	5.0854	0.0	22500	36.6920
1	100.0	6.7674	0.0	30000	42.2400
1	125.0	7.1421	0.0	37500	42.0023
1	250.0	12.2331	0.0	75000	46.1800
1	375.0	16.5433	0.0	112500	64.8595
1	500.0	22.4957	0.0	150000	120.9800
1	625.0	30.0547	0.0	187500	312.8758
1	750.0	34.8860	0.0	224726	780.4000
2	25.0	2.9953	0.0	7500	33.0209
2	50.0	4.2111	0.0	15000	37.7759
2	75.0	5.3958	0.0	22500	36.6920
2	100.0	6.6742	0.0	30000	42.4775
2	125.0	7.6556	0.0	37500	42.0023
2	250.0	12.9270	0.0	75000	46.1650
2	375.0	18.3591	0.0	112500	66.2195
2	500.0	24.6689	0.0	150000	121.7450
2	625.0	30.6223	0.0	187500	305.9158
2	750.0	36.4404	0.0	224726	662.3325
3	25.0	3.0155	0.0	7500	33.0209
3	50.0	4.2016	0.0	15000	37.7759
3	75.0	5.7091	0.0	22500	36.6920
3	100.0	6.5729	0.0	30000	42.7150
3	125.0	8.1750	0.0	37500	42.0023
3	250.0	13.6267	0.0	75000	46.1500
3	375.0	20.2036	0.0	112500	67.5795
3	500.0	26.8740	0.0	150000	122.5100
3	625.0	31.1774	0.0	187500	298.9558
3	750.0	38.0014	0.0	224626	644.2650
4	25.0	3.1514	0.0	7500	33.0209
4	50.0	4.3200	0.0	15000	37.7759
4	75.0	5.3866	0.0	22500	36.6920
4	100.0	8.4581	0.06	30000	42.9525

replicas	virtual users	node response	failed rate	http reqs	http req duration (95%)
4	125.0	7.6466	0.0	37500	42.0023
4	250.0	15.0386	0.0	75000	46.1350
4	375.0	18.8049	0.0	112500	66.2195
4	500.0	28.3999	0.0	150000	123.2750
4	625.0	30.3705	0.0	187500	265.9158
4	750.0	42.2513	0.0	224626	576.1975
5	25.0	3.2876	0.0	7500	33.0209
5	50.0	4.4373	0.0	15000	37.7759
5	75.0	5.0527	0.0	22500	36.6920
5	100.0	10.3771	0.11	29999	43.1900
5	125.0	7.1005	0.0	37500	42.0023
5	250.0	16.4688	0.0	75001	46.1200
5	375.0	17.3604	0.0	112500	64.8595
5	500.0	29.9354	0.0	150001	124.0400
5	625.0	29.5210	0.0	187500	232.8758
5	750.0	46.5583	0.0	224520	608.1300
6	25.0	3.3074	0.0	7500	33.0209
6	50.0	4.4802	0.0	15000	37.5580
6	75.0	5.1018	0.0	22500	37.2340
6	100.0	10.3541	0.06	29999	40.9705
6	125.0	7.1695	0.0	37500	42.5962
6	250.0	16.7194	0.0	75001	52.0885
6	375.0	17.5287	0.0	112500	69.6530
6	500.0	30.5481	0.0	150001	126.5400
6	625.0	29.8073	0.0	187500	265.3330
6	750.0	47.3711	0.0	224520	496.9807
7	25.0	3.3251	0.0	7500	33.0209
7	50.0	4.5204	0.0	15000	37.3400
7	75.0	5.1477	0.0	22500	37.7759
7	100.0	10.3225	0.0	30000	38.7511
7	125.0	7.2341	0.0	37500	43.1900
7	250.0	16.9616	0.0	75000	58.0570
7	375.0	17.6862	0.0	112500	74.4464
7	500.0	31.1480	0.0	150000	129.0400
7	625.0	30.0753	0.0	187500	297.7901
7	750.0	48.1616	0.0	225000	425.8314
8	25.0	3.3651	0.0	7500	33.5193

replicas	virtual users	node response	failed rate	http reqs	http req duration (95%)
8	50.0	4.5578	0.0	15000	36.6212
8	75.0	5.1905	0.0	22500	38.2635
8	100.0	10.2824	0.02	30000	38.0255
8	125.0	7.2942	0.0	37500	40.8992
8	250.0	17.1947	0.0	75000	51.8535
8	375.0	17.8327	0.0	112500	68.1373
8	500.0	31.7342	0.0	150000	124.7850
8	625.0	30.3246	0.0	187500	267.3457
8	750.0	48.9284	0.0	225000	413.8057
9	25.0	3.4031	0.0	7500	34.0177
9	50.0	4.5924	0.0	15000	35.9023
9	75.0	5.2299	0.0	22500	38.7511
9	100.0	10.2337	0.03	30001	37.3000
9	125.0	7.3496	0.0	37500	38.6083
9	250.0	17.4185	0.0	75001	45.6500
9	375.0	17.9679	0.0	112500	61.8281
9	500.0	32.3054	0.0	150001	120.5300
9	625.0	30.5547	0.0	187500	236.9012
9	750.0	49.6700	0.0	224504	411.7800