## Chapter 2 – Method

### Data

### Data used in Lit Review

Creating a time-series model that can effectively forecast, requires a dataset to train it.

There are a number of commonly used data sets to test web applications. [15] analysed the WorldCup98 dataset. 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.

[13] - 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.

[14] - Conducted a load test on a Kubernetes cluster for 6 hours, periodically extracting metrics from this cluster every two minutes. creating a dataset of 300 data points. Similar to [14], [12] - Created their own data set using the Kubernetes Metrics server to record metrics like: CPU and Memory utilisation for the node, number of pods each day over the course of 4 months.

For the purpose of this investigation, the WorldCup98 dataset will be 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. Creating my own would risk providing a solution without a solid foundation in comparative research.

This data set in particular provides 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 displays, regular substantial spikes and troughs in website traffic periodically throughout the day. Presenting a solid representation of seasonality that many websites observe. In turn, making it a perfect representation of predictable, event-driven web traffic dynamics. Therefore, ideal 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, M. and Jin, T [] conducted a research investigation to characterise the workload on the website for the three months it was active. In this they found that in the months surrounding the event the recorded traffic was significantly lower. The number of queries has a significant increase between 10th of June - 12th of July (the start and end date of the event). They 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 more sparse as the number of games being played reduced.

A graph showing a number of data

Description automatically generated

Web traffic on weekends was also lower compared to weekdays, despite more matches being played. Arlitt, M. and Jin, T attributed this to people's preference for watching the matches on television during weekends. I replicated this observation, performing a series of t-tests on the data to compare the web traffic between days in the week and the weekend. It showed a significant difference between weekend and week traffic.

[t-test table]

Ideally, we would want to train the time-series forecast model on the entire WorldCup98 dataset. However, as we have just outlined, there are some significant irregularities that will 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 means it makes more sense the isolate the training and testing to 15th -20th June, with training taking place Monday-Thursday and Friday acting as the Testing data. This still produces 5760 data points to train the model on, whilst maintaining a good representation of seasonal website traffic is used fit the model.

A graph of a graph

Description automatically generated

### Design

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

#### 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 is a Long-Short-Term Memory model (LSTM).

An LSTM is a type of Recurrent Neural Network (RNN). 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 [<https://www.ibm.com/topics/recurrent-neural-networks>]

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 []. This is where the model struggles to remember long term trends in the data - the further away the information is from the present the less impact it has on the output.

The LSTM model is designed to fix this issue. An LSTM is made up of three structures called gates. The Forget gate: decide what information is no longer important to keep in the memory. The input gate: deciding what part of the input is 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 makes them ideal for handling large amounts of time-series data.

A diagram of a flowchart

Description automatically generated

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. I will be implementing the LSTM model using TensorFlow’s [] LSTM library. For this library, the LSTM model has a number of attributes can be configured.: LSTM units, batch size, epochs, look-back window, and learning rate. To optimise these hyperparameters, grid search will be implemented [] . Training and testing the model at different configurations, ultimately choosing the configuration with the lowest RSME [find reference] value.

#### Application Profile

Where one model will be used to predict future data, this doesn’t directly inform us on 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 we will need to develop a separate model to map between the amount of traffic the web app is receiving and the number of pods that should 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 []. 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. [Energy-optimal mobile application execution: Taming resource-poor mobile devices with cloud clones. IEEE INFOCOM, 2012.]

This investigation will produce a similar model to precisely define at what load to change the amount of resources available to a web application. This approach is closer to that conducted by []. The way the application profile is developed in this study differs slightly. Toka, Dobreff, Fodor and Sonkoly [] 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 is to reduce the amount of power consumed by a web app. As a result the application profile is modelled with that in mind. Similar to [] the web app is isolated to it’s own node with end points to the application exposed by a Load Balancer.

However, a HPA won’t be used. Instead, for each test the deployment is scaled to a fixed number of pods (1-10). Using k6 a series of load tests are then conducted where for different stages of load ranging from 100 QPS to 2250. Each stage lasts 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 [] is used to collect the average CPU utilisation of all of the pods in the deployment and k6 is used to measure number of failed requests. Once a round of tests for a specific number of pods is complete, the script increases the pod count by one and repeats the process, thereby systematically exploring how the system handles increased loads with different numbers of pods.

To create the profile itself, the data collected in the tests is analysed to discover which number of pods consumes the least energy at each stage of load. Using this formula provided by Sara Bergman. [<https://devblogs.microsoft.com/sustainable-software/how-can-i-calculate-co2eq-emissions-for-my-azure-vm/#:~:text=The> memory energy consumption is,%3D (2\*P\_c).&text=This means we get%3A E,over this specific time period]

A close-up of a logo

Description automatically generated

Where TDP is the thermal design power of the CPU the node contains. This will be used to estimate ethe power consumption of the node along with the number of Virtual CPUs (vCPUs) and the percentage utilisation of the CPU itself. The optimal number of pods will be the configuration with the lowest consumption whilst keeping the amount of failed requests above 99.5% to ensure that the system can maintain a quality of service that a true production environment would require.

It is possible to attempt to tackle this task with one unified model. Using requests per minute, CPU utilisation, memory utilisation and more to together inform a model on the optimal number of pods. However analysing the work conducted by [][] there are some distinct benefits to opting for two individual models

Having two models allows for greater modularity in the system design. This means greater flexibility in adapting to changes in one area without affecting the other. For example, in a professional environment, should the web traffic of the web app change significantly due to a new strategy or an external event, only the forecasting model would need to be updated. Conversely, if the website or cluster architecture was updated only the application profile would need to be re-modelled.

Having one unified model would also be much more complex. The number of variables the model would need to account for simultaneously would make it much harder to train and optimise. A study by Gosiewska, A., Kozak, A. & Biecek, P found training time and computational resources were significantly reduced, with reductions noted up to 30-50% in some scenarios compared to training complex models [15].

In this case having these two models would therefore be more computationally efficient. A prediction for the optimal number of pods is needed every minute. Once the Application profile is created it exists simply as a database so the only computation needed is a univariate time-series forecast. However, performing at such a frequency, the amount of total computation of a multi-variate unified model would be significantly higher.

Once the two models are created, the third and final task is to design and building a framework that utilises them both to pre-emptively scale the deployment. This system will be designed as follows.

The system will be made up of a cluster containing two node pools. One will be used as a worker node. It is here that the web app, and the web app exclusively, will be deployed. As this will be the node being monitored in the experiments it’s resources must be exclusively utilised by the web app deployment we are testing. The other will be a system pool. This will contain all of the system and monitoring pods to abstract any background computation away from the worker node.

The cluster also contains a Con Job [<https://cloud.google.com/kubernetes-engine/docs/how-to/cronjobs>]. This repeatedly create Kubernetes Jobs on a pre-defined schedule. In this case, calling an endpoint inside of an external Virtual Machine (VM) [].

This VM will contain 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 [], our system will have an external HPA Controller contained in the VM. This external Controller will manually force the deployment to scale depending on the recommendations from the two models. This controller will be called every minute by the Cron job and in turn, will first extract the total amount of queries the web app has received over the last minute from Prometheus. It will then communicate this rate of traffic with the LSTM, receiving back a forecast for the rate of traffic over the next minute. This forecasted number is then provided to the image containing the application profile which will return the optimum number of pods that should be run for that level of traffic. The external Controller image will then instruct the true cluster Controller to scale to that number of pods, simulating the role of a true HPA Controller.

### Experimental Set up

Our experiment will be conducted to compare the energy consumption of the worker node in our cluster using the PA against the standard Kubernetes HPA. To do these 4 individual tests on both systems will be conducted to examine their capabilities at a variety of traffic conditions to ultimately evaluate 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 \_\_ of our PA we will configure the HPA to scale on CPU Utilisation. In other solutions that compare against a regular HPA, the exact limit that is scaled to varies from 60-80% utilisation. For this experiment we will follow the configuration conducted by [<https://ieeexplore.ieee.org/document/9709810>] and set the scale target to 80% utilisation.

The 4 tests will be taken from 4 hour long periods from Wednesday 24th June in the WorldCup98 dataset. Day in particular was chosen to ensure as it still follows the intraday seasonality that is found in the training-test week but is still completely unique outside of that dataset. This ensure the model being good and there is no contamination in the training and validation data.

K6 will again be used to induce load on the dataset to reflect the \_\_ of traffic from the dataset. To gain the best insight into the ability of the PA the 4-hour periods have been chosen specifically to each reflect different \_\_ of load that \_\_\_:

* Steady traffic (Total variance of **422,676 between** 10:24-11:24) - tests how well the autoscaler handles steady traffic.
* Largest increase (Increase of 75447.0 between 1998-06-24 13:49:00) - test how well the auto-scaler can scale up.
* Largest decrease (Decrease of 92008.0 between 1998-06-24 16:02:00) - to test how well the auto-scaler scales down.
* High Variance (Total variance of 986,889 between 20:11-21:11) - Tests how well the auto-scaler reacts to a change in trend.

A graph of a graph showing the number of traffic

Description automatically generated

Load is induced for the 20 minutes before each test case to ensure the final results aren’t affected by any “cold start” effects.

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

The CPU utilisation will once again use \_name formula [] to derive the energy consumption. This will be totalled to derive how much power the node/deployment consumed in the test period. It describes how well the autoscaler manages 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 will be combined with total requests to evaluate the effectiveness of the deployment at the \_\_ that each Auto scaler has configured it in. This follows the RED method (Rate, Errors, Duration) []. This is an adaptation of the USE method [<https://www.brendangregg.com/usemethod.html>] 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 is to expose how healthy the microservices within your architecture are.

Each of these metrics directly affect the quality of service and the user experience. By measuring how many requests are lost during peak times and how long these requests take, you can evaluate the reliability of your autoscaler in maintaining service availability and responsiveness. Assessing whether the scaling strategy is effectively preventing bottlenecks and system overloads.

By measuring both energy consumption and the number of lost requests, you can ensure that your autoscaler not only optimises operational costs but also maintains high service quality and reliability. Verifying if the whether the autoscaler can be used in a production environment.

### Project Management

#### Version Control

Git will be used as a repository for the codebase for the project. Commits will be clear and concise and outline what is changes or added and why this has taken place. Extra descriptions will be provided in cases where the commit message by itself is not enough to indicate what actions were undertaken.

#### Continuous Integration

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

1. First create a Kubernetes Cluster in Azure AKS and successfully deploy TeaStore on it.
2. Set up a k6 load testing framework and conduct a success test at constant load on the deployment.
3. Set up testing scripts for to create the application profile data set.
4. Apply the standard HPA to the TeaStore deployment and create test script to run the 4 different test cases on it.
5. Develop the application profile.
6. Train and Test the LSTM machine learning model.
7. Deploy the application profile and the machine learning model to azure.
8. Develop the Cron Job to utilise both the application profile and the LSTM model.
9. Deploy the Cron Job onto the Kubernetes cluster.