Adaptive microservice scaling for elastic applications

Nathan Cruz Coulson, Stelios Sotiriadis and Nik Bessis

***Abstract*—Today, Internet users expect web applications to be fast, performant and always available. With the emergence of Internet of Things, data collection and the analysis of streams have become more and more challenging. Behind the scenes, application owners and cloud service providers work to meet these expectations, yet, the problem of how to most effectively and efficiently auto-scale a web application to optimise for performance whilst reducing costs and energy usage is still a challenge. In particular, this problem has new relevance due to the continued rise of Internet of Things and microservice based architectures. A key concern, that is often not addressed by current auto-scaling systems, is the decision on which microservice to scale in order to increase performance. Our aim is to design a prototype auto-scaling system for microservice based web applications which can learn from past service experience. The contributions of the work can be divided into two parts (a) developing a pipeline for microservice auto-scaling and (b) evaluating a hybrid sequence and supervised learning model for recommending scaling actions. The pipeline has proven to be an effective platform for exploring auto-scaling solutions, as we will demonstrate through the evaluation of our proposed hybrid model. The results of hybrid model show the merit of using a supervised model to identify which microservices should be scaled up more.**

***Index Terms*—Cloud computing, load balancing, microservices**

I. Introduction

IT is well established that the related trends of rising Internet of Things (IoT) and smartphone use [1], the ubiquity of cloud computing [2], and the proliferation of digital content and services have made the web application a staple in the everyday lives of billions of people. [3]. The advent of cloud computing and elastic web application architecture has allowed application owners to manage the ”busty” and ”unpredictable” workloads which are inherent in internet based services [4] by provisioning or removing computing resources according to demand in order to maximise performance (or minimise the so-called service level agreement violations) while at the same time reducing costs [5].

The most widely adopted elastic web application paradigm involves splitting the application into different logical tiers, including (a) the user presentation or web server tier, (b) the business logic tier and (c) the database tier [4], [6]. Following the standard monolithic architecture style, the business logic tier would be able to process a wide range of different requests, or in other words, it offers a range of services within the scope of the application [7], [8].

The elasticity of this multi-tier monolithic architecture is achieved by utilising a load balancer to receive and distribute HTTP web requests between a group of identical application instances (called clusters or availability groups). The application instances would normally consist of a combined user presentation and business logic layer hosted on a Virtual Machine (VM) [5]. A ”horizontal” auto-scaling system would determine if the application needed to add (scale-out) or remove (scale-in) an instance in order to maintain performance or reduce costs. It is also possible to reconfigure VMs by adding (scale-up) or removing (scale-down) computing resources [9]. This work focus on the more widely used horizontal auto-scaling systems (henceforth just auto-scaling systems).

In recent years there has been a movement towards a more distributed and service-orientated application architecture style which involves breaking complex applications down into logically distinct and complementary microservices [10]. These microservices can be full stack applications with a dedicated data persistence tier or they can be stateless [8]. The microservices architecture brings significant potential benefits in terms of development and deployment agility, scalability and robustness as demonstrated by its successful adoption by many leading technology companies including Netflix, Google, and Amazon [11]. However, the increased complexity inherent in microservices architecture also presents us with a new set of challenges with regards to how to most effectively and efficiently auto-scale such an application. A key question with regards to auto-scaling such applications, given that nodes are heterogenous in a microservices architecture, is ”which microservice should be scaled?”. This work focuses on the development of a solution to allow autoscaling of microservices in an efficient way.

In this work we work towards these challenges, and we focus on developing an autoscaling system for microservice architectures. In the general case, an auto-scaling system would respond to current (”reactive”) or predicted (”proactive”) changes in workload or resource utilisation by adding or removing applications instances (”horizontal scaling”) or computing resources like CPU cores (”vertical scaling”) as required to meet the SLAs [6], [12], [13]. For example, ”reactive” threshold-based rules are the most commonly deployed solution by Cloud Service Providers (CSPs) like AWS, Azure and Google Cloud. In such a system, the workload or resource utilisation metrics of the web application are monitored and scaling actions are taken when those metrics breach or fall below certain user-defined thresholds [5], [6], [14].

In the context of single or multi-tier monolithic application

N. Cruz Coulson and S. Sotiriadis is with the Department of Computer Science and Information Systems, Birkbeck, University of London, UK e-mail: stelios@dcs.bbk.ac.uk

N. Bessis is with the Department of Computer Science, Edge Hill University, UK e-mail: nik.bessis@edgehill.ac.uk

Manuscript received October 13, 2019

the question of "What to scale?" is typically self-evident; an application instance of the user-presentation and/or business logic tier hosted by a VM. This is because, as per the monolithic architecture style, most of the application logic and processes occur in this tier. It is also relatively straightforward to replicate identical instances of this tier on-demand because they are normally stateless with regards to processing requests. This node homogeneity, together with load balancing, is what allows a web application to grow "horizontally" by adding new instances to a cluster in response to increased demand [15], [16], [17]. In a microservices context the picture is more complicated, there could be dozens of separately deployed microservices which are unevenly stressed by workloads [18].

The aim of this work is to contribute to the relatively new area of microservice based web application scaling by creating a predictive auto-scaling system which addresses the question of "What to scale?" as well as "When and how to scale?". Machine learning, in the context of auto-scaling applications, is typically focused on the prediction of future application load. If trends and spikes in load can be predicted through a time series machine learning technique like a Long Short Term Memory (LSTM) neural network, then the scaling system can react proactively and provision resources ahead of time. The same logic applies for predicting low load and thereby reducing resources to save energy and costs [19].

II. Literature review

There has been a significant amount of research done on how to most effectively and efficiently auto-scale web applications. This area has been extensively studied because of the sheer scale and importance of web applications in this current cloud powered era [1]. Much of the research has focused on the questions of "When to scale?" and "How to scale?" but not so much on "What to scale?". As we have mentioned in the introduction, the primary reason for this is that most studies take a load balanced multi-tier web application architecture as the model and a monolithic multi-service business logic layer, hosted on a VM or a container, as the object of the scaling decision [2], [5], [20], [21], [22]. In contrast, our work focuses on the emerging research area of microservice based web application scaling. The following sections present a literature review analysis on (a) auto-scaling multi-tier web applications, (b) time series analysis and machine learning approaches and (c) auto-scaling microservice based web applications, that are essential aspects of our work.

A. Auto-scaling multi-tier web applications

Threshold-based auto-scaling methods are the industry standard due in large part to their simplicity. The application owner sets arbitrary scaling thresholds based on a particular metric, usually CPU or request rate, if the average value in a cluster exceeds or falls below a threshold for a given amount of time, then a scaling decision is executed. Many authors have commented on the advantages and disadvantages of this approach [14], [5], [20], [6]. One of the main issues is that, despite appearing simple, to set the right threshold requires a deep knowledge of how the application operates [21]. Another key issue is that due to the inherently "reactive" nature of the threshold-based approach and the fact that any scaling action has some non-trivial delay attached to it (time-to-scale or TTS) it is likely that the performance of the application will degrade while the additional resources are being provisioned [22].

In their survey papers exploring application auto-scaling both [5] and [2] identified the common usage of the Monitor, Analyse, Plan and Execute (MAPE) pattern in the approaches of several authors, so it is a useful way to frame our discussions. In the monitoring phase of an auto-scaling system a decision must be taken beforehand about what data to collect and from where. The work of [2] categorised these metrics into low-level (CPU, memory, disk I/O), high-level (request rate, average response time, throughput), and hybrid (a combination of both) [2]. Typically, these metrics would be collected from the object of the scaling discussion which in the context of a multi-tier web application is usually the business logic tier [5]. The most common metrics used are CPU utilisation as a general measure of the processing capacity of an instance cluster and the undifferentiated request rate per second as a measure of relative load [5], [2]. These general metrics can work well under the assumptions that (a) on average all requests place the same level of load on a system and (b) all requests place this load on the same component of the system.

In [16] the authors note the significant difference in resource utilisation between static and dynamic web page requests from an efficient load balancing perspective. The literature around microservices also suggests that dealing with highly differentiated workloads, which can stress different microservices, is a challenge [23], [18]. If we hope to accurately predict the performance of a microservice based web application it is therefore likely that we need to take a "content aware" strategy similar to that used in [16] for smart load balancing and in [24] for web server performance prediction. It is likely that we would take a similar data collection approach to Peng et al by collecting and categorising requests by type and number received in a given time period [24]. The analysis phase is where we find the greatest variation between auto-scaling approaches. Lorido et al categorised these approaches into five groups [5] that include (a) threshold-based rules, (b) reinforcement learning, (c) queuing theory, (d) control theory and (e) time series analysis.

Threshold-based rules have been discussed already as setting the standard benchmark for simple auto scaling. We will not discuss reinforcement learning in this proposal but it is worth noting that due to the long training times this method is unlikely to be favoured in production environments. Queuing and control theory approaches will also not be directly addressed in this proposal however we can note that they do provide powerful mathematical models which do seem suitable for certain auto-scaling scenarios, however they can be difficult to generalise across different systems [5]. We will focus on time series analysis and machine learning approaches in a separate section as this is the type of analysis that we will utilise in our work.

B. Time series analysis and machine learning approaches

The most common time series analysis techniques, that are applied to predicting future values for workloads or resource utilisation, belong to the Auto-Regressive (AR), Moving-Average (MA) family of methods which include ARMA (Auto-Regressive Moving Average) and ARIMA (Auto-Regressive Integrated Moving Average) [5], [14], [25], [4], [20]. Lorido gives a good theoretical overview of these techniques in [5] and how different authors have applied them to the prediction of application or resource level metrics. Techniques like Exponential smoothing (ES) are similar in that they also fall under the category of univariate analysis using past values to predict future values with some error [5]. Broadly speaking, these techniques have performed well when applied to predicting a single variable based on a relatively small historical window [25], [4], [20].

The relative simplicity of this approach is one reason why it has been successful in the context of a multi-tier monolithic web application. If we are trying to predict the behaviour of a complex interconnected system, for example a microservices based web application, then we would need a technique which can handle greater dimensionality and non-linearity [26]. Techniques based on Recurrent Neural Networks (RNNs) offer these capabilities and are designed to learn complex patterns from a time series dataset. Furthermore, RNN-LSTM (Recurrent Neural Network with Long Short Term Memory) models have also demonstrated the ability to learn effectively from long term as well as short term patterns in the data.

There are many variations of RNN based models such as the Bidirectional Long Short Term Memory (BILSTM) model used by Tang et al in [26] or the Deep Recurrent Neural Network (DRNN) used by Peng et al in [24]. These models attempt to capture relevant temporal and multivariate patterns in the data the authors choose to collect. We have chosen to focus our attention on RNN models in part due to the range of network architectures and hyper-parameters we can tune for our dataset. One of the key challenges for our work is to find a model that can best capture the important relationships in our dataset while remaining generalisable and avoiding issues around overfitting. We are encouraged that in studies like [24] RNN models have been used to accurately predict web server performance using a similar dataset to the one we plan to use: NGINX request logs. It is also promising that another related problem, that of predicting container load, has been successfully tackled with a BI-LSTM model [26].

C. Auto-scaling microservice based web applications

As is noted in [2] the work on service-based architectures is still at an early stage despite the rapid adoption of microservices by tech companies that need to operate at scale. In [18] the authors explicitly address the question of "What to scale?" in a complex microservices context by employing queuing and graph theory to dynamically model the relationships between microservices. The research carried out in [23] applies a variety of time series analysis and machine learning techniques to a microservices workload to predict future resource utilisation.

In this work we build on the early work done on auto-scaling microservices and seek to extend it by looking at how request differentiation as demonstrated in [24] can help use to decide which microservice to scale and when. In deciding on a methodology, we will apply the findings of the currently available research to address some of the potential gaps in the microservice auto-scaling literature. Specifically, we will avoid the relatively targeted and rigid analytical models demonstrated in [18] but using RNN based techniques to capture long term patterns.

III. Microservice auto-scaling pipeline

In our work we developed a prototype system to support microservice based web application scaling to execute a variety of auto-scaling experiments. The requirements for the web application were as follows:

- The application must accept a range of HTTP requests (from a fixed set) and deliver a response.
- The application must consist of several microservices, each with a distinct function which stresses the underlying resources to a greater or lesser degree.
- In response to an external HTTP request the application should make internal HTTP requests to the required microservices.
- The application should be scalable on a the microservice level, for example we should be able to scale up one microservice but not the others.
- The application must record all external HTTP requests and log the request response time.
- The application should be simple to configure and deploy in an agile manner.

For the web app stack we used the Flask web framework for microservice and API development. For example, this request: http://host/2/3-2-4-2/45 is received by App 1 (all requests go through App 1 first) and then the integer parameter 42 is passed to each of the apps in the path in order: 2, 3, 2, 4, 2. Each time the requests hits a microservices the specific functions are carried out on the input integer and the output is passed to the next microservice. The application is structured in such as way as to make it very straightforward to change any of the application functions or add new microservices, for example one that queries a database, if that was required by a different research scenario. Upon activation, the microservice functions execute one of a range of operations which are designed to utilise resources in different ways which are covered below.

- App 1 is the simplest microservice. Like all the microservices, the root domain returns a hello world statement, what makes it unique is that all requests are sent from the NGINX reverse proxy server to App 1 first and then the request is routed to the next microservice specified in the url.
- App 2 generates a list of random numbers five times the size of the input integer and then sorts it using the built in Python sort function which utilises the Tim sort algorithm. The function returns a random number from that list.

- App 3 is similar to App 2 but uses an implementation of the quick sort algorithm to sort the list.
- App 4 makes a request to an open API (http://numbersapi.com) using the input number as a parameter. Then returns the length of the response text multiplied by 12.

We use NGINX for its reverse proxy capabilities. An NGINX instance receives and logs the incoming requests before passing them on to App 1. Then, it records the response time once the request comes back to be passed on to the client. We used Docker and Docker Swarm as our container technologies. Each microservice was deployed as a Docker "service" and connected on a common network. Each service can contain one or more containers. Requests received by the service are then load balanced, with Ingress, between the containers in that service. The configuration of these connected services was defined as a Docker "stack". This stack could be said to be the top level microservice based application in deployment. This setup, using Docker Swarm, allows us to deploy the microservice application on any machine with Docker installed and define the starting state i.e. App 1 = 2 containers, App 4 = 3 containers and so on. Furthermore, through the Docker API we can manually, or automatically issue commands to take scaling actions on an already deployed application i.e. scale App 3 = 4 containers.

The following describes a simple microservice architecture and the flow of events.

- A HTTP request is sent from a client to the microservice VM: http://host/3/2-4/42
- The request is received by the NGINX container, it is logged and then passed on to Microservice 1
- Microservice 1 parses the request and then passes the request to the next microservice in the chain: microservice 3
- Microservice 3 parses the request and then carries out it's functions on the input parameter, it then passes the output to the next microservice: microservice 2
- Microservice 2 parses the request and then carries out it's functions on the input parameter, it then passes the output to the next microservice: microservice 4
- Microservice 4 parses the request and then carries out it's functions on the input parameter, including a call to an external API, it then passes the output response back to NGINX
- The NGINX container received the processed request and passes on the response to the client. The total time taken to process this request is recorded in the NGINX logs. In each instance that a request is received by a Docker service, that service is load balancing between 1 or more containers which are running the actual microservice.

Alongside creating the web application we also required a limited request set which stressed different microservices in different ways. Due to the way we designed our application, it will accept any url of the form:

host-ip-domain/*appX*/*appX*-*appX*-appX*-...*appXn/*integer-value*

It will be transparent from the URL text which microservices will be stressed by the request (see Flask section above). However this is usually not the case, the participation of microservices in servicing a request can be entirely opaque to the user. We will not know which microservices are stressed by a request and so we will not encode this information in the model. Instead we will one-hot-encode the requests. In order for this to work we needed to generate a request pool with subsets of requests that are "biased" towards certain microservices. This is because we need to create a sequence pattern for the prediction algorithm to learn. We achieved this by creating a simple function that generated valid URLs but with some bias towards one or more microservices.

To simulate the HTTP request workload we used Locust.io, which is a high performance configurable load testing tool written in Python. The Locust.io tool allowed us to simulate HTTP traffic to the web application. We were able to specify the number of concurrent users and how often requests would be made. This was kept constant in order to focus on the question of which microservice was being stressed more by changing request mixes. The Locust.io framework allowed us to easily create functions which randomly picked from our biased request subsets. By changing the ratio in which the Locust script picked which HTTP request to make we were able to "bias" the traffic for a specific period toward one or more microservices.

Once we had the web application deployed on one VM and the locust application, equipped with a bias url list, on another VM we were able to start our experiments. Our synthetic data would need to meet certain requirements to test the auto-scaler system's predictive capabilities. The data would have to contain some short and long term request sequence patterns. We were able to achieve the short term sequence pattern through the use of the URL bias as explained above. For example, during the first run (the first line in the table screenshot), the first 219,358 requests were biased towards microservice 4. This means that a predictive model should learn that certain URLs are more likely than others, given the last 1000 requests seem to be biased towards this subset. The long term pattern was achieved by repeating the same pattern of bias: from 2 to 3 to 4 to 2, and so on, through several cycles. If the request mix prediction model is able to predict that certain periods of bias follow others this should be reflected in predictions.

Once the log files were in place we developed an extraction and cleaning method which transformed this data into our experimental dataset with the following features.

- Request response time (resp_time): the time in seconds, as a float, that it takes for a request to be passed to the microservice app and then return to the NGINX reverse proxy server.
- Byte sent (bytes_sent): the number of bytes sent in the request as an integer.
- HTTP response code (resp_code): the response code of the specific request as a string i.e. 200 is "successful".
- URL (url): the url of the request as a string i.e. 2/3-2-4/45
- Number of containers in each microservice cluster (app_X_container): four integer features denoting the container allocation in each microservice cluster.

- Timestamp of the event.

IV. Exploratory Data Analysis: visualising patterns in log data

According to the previous section, the dataset size is 4,483,537 rows (requests) and 9 features. The key feature of interest is request response time. We explored the overall distribution in the dataset through a histogram. Figure 1, presents the response time distribution with App 2 scaled 3X. The mean of 2 scaling time is 17.7 seconds. Figure 2, presents the response time distribution with App 3 scaled 3X. The mean of 3 scaling time is 8.4 seconds. Figure 3, presents the response time distribution with App 4 scaled 3X. The mean of 4 scaling time is 17.8 seconds. Figure 4 presents the response time distribution with all Apps scaled 3X. The mean of all Apps scaling time is 10.9 seconds

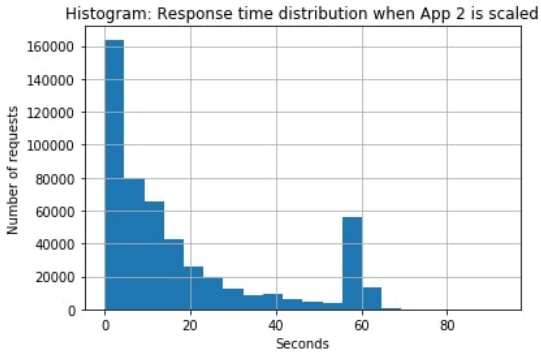

Fig. 1: Histogram: Distribution across requests where App 2 was scaled up

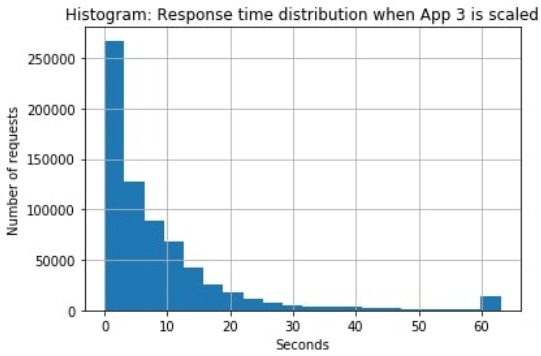

Fig. 2: Histogram: Distribution across requests where App 3 was scaled up

We can observe the following findings:

- As we can observe, *it seems likely from this initial analysis that App 3 is a significant bottleneck within the microservice*. We can see this because when App 3 is scaled the mean response time drops by over 50% and the distribution definitively skews to the lower range with the number of requests in the 60 second range significantly dropping.
- When the other Apps are scaled without scaling App 3 the result is a higher response time. Interestingly, the average request response time when all Apps were scaled is actually higher than when only App 3 is scaled.
- This could be explained by the fact that there are less bare metal resources available to the App 3 containers due to overcrowding on the VM.

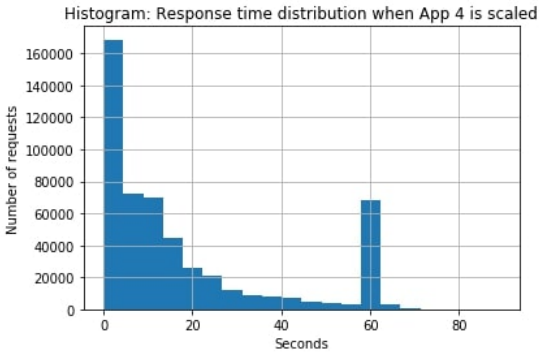

Fig. 3: Histogram: Distribution across requests where App 4 was scaled up

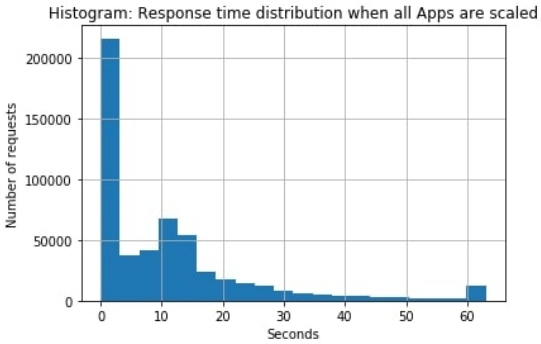

Fig. 4: Histogram: Distribution across requests where all Apps were scaled up

V. Auto-scaling recommendation system

As discussed in literature review section, this work employs a stacked LSTM model. In particular, we want to accurately predict the composition or ratio of the next n requests in order to recommend the appropriate scaling action. Given this requirement, it is not actually important that our algorithm predicts the next request with a great degree of accuracy, what we are interested in is the composition of the next n requests. As with any sequence or time-series problem the key inputs are the previous time-steps. The key question for us is *how many time-steps should we include in the input vector?* The ideal answer to this question may vary from scenario to scenario given the temporal structure of the data. In our case we are focusing on a proof-of-concept system and we have a good idea of the temporal structure due to the dataset being synthetic. This, together with considering resource constraints (the more timesteps are included the more computationally expensive training becomes), lead us to decide to fix the number of time-steps to 1000 requests.

In order to feed our request sequence into the model we have to first encode it in some numerical fashion. We chose to

create one-hot-encoded variables for each individual request, so a total of 129 features. Our output variable, the request mix we are trying to predict, will be a representation of the next n requests. Since we are using this to make a scaling decision, we are more interested in an overall pattern, or bias of, the requests towards stressing one or more microservices so the order of requests within the subset is not important to us.

The final step of the preprocessing stage is to split into training and test sets by splitting the odd and even requests. The upside of this strategy was that I could be relatively sure that the distribution of requests in both the training and test sets were very similar. The downside is that the model we train may not be as robust on unseen data due to the potential for data leakage. In a real world case we would recommend that data is split by a logical temporal structure in order to preserve distributions in training and test sets, i.e. four full weeks with seasonal effects in the training set and two full weeks in the test set. This would make the model more generalisable across unseen, but still similarly distributed data.

The aim of our model is to predict a vector of real numbers, the scaled cumulative sum of the next 250 requests, based an input of the last 1000 requests as a sequence. This can be described as a regression problem which seeks to minimise the Mean Absolute Error (MAE) between the predicted and actual request mix vector. For this reason we decide to use MAE as the primary loss function. However we were not just interested in whether or not the model was able to predict the exact request mix by url. Due to the way we generated the dataset, there are subsets of urls which are more likely to appear together during certain periods, i.e. when we activated the Locust script with a bias towards the subset of requests which stressed microservice 3 more than the others.

With this is mind, we wanted to have another metric which focused on the model's ability to predict the bias of the request mix as a whole. The way we did this was to create a set of functions (see appendix Z) which summed the number of requests for each bias in the predicted vector and compared it to the actual bias in the observed vector. This method would reward the model if it got close to the general bias of the request set even if the exact request predictions were often wrong. There are a large number of hyperparameters that one can tune for a stacked LSTM model in Keras. However, given our time and resource limitation we chose to focus on key model architecture and training hyperparameters in order to get a decent, if not optimal, model. We created a grid search method which we ran over the following hyperparameters, including (a) optimizers (Adam at lr = 0.05, 0.01, 0.001), (b) epochs (1,2,4,8,10), (c) batches (64, 128), (d) number of units (50, 75, 125) and (e) number of hidden layers (2,3,4,5,6).

The prediction element of this process can be formulated as a regression problem. Given that a fairly simple relationship could exist in the data (more resources for a microservice during a period of stress for that microservice should intuitively reduce the average request response time), a linear model may be reasonable successful. We selected a range of, primarily, linear regression models to start our modelling process including linear regression, ridge regression, lasso regression, elasticNet and random forest regression. As in the sequence prediction model we will also use a rolling sum transformation to obtain two key feature sets: the request mix over 250 time-steps and the average request response time over that same period. The other primary feature set that we'll include in the predictive model are the resource allocation features which indicate how many containers are active in each microservice. In addition to these feature sets, we also collected a number of other datapoints from the log data such as (a) bytes sent, HTTP response code and timestamp. At this point it would be possible to engineer some further features that may be of interest for example (a) the error rate in last 250 requests (as indicated by HTTP response code) and (b) the temporal and seasonal features (time of day or day of the week as indicated by the timestamp).

In the case of error rate, we believe that this could be a crucial feature for certain use cases i.e. if reducing error rate is more important to the application owner than decreasing response time. In fact, error rate could be an output variable or a predictor. Given these feature selection decisions the final feature set for the supervised regression modelling table consisted of: container allocation data (scaled), cumulative rolling sum of the next 250 requests (scaled), and the output variable of the average request response time across those 250 requests (scaled). The aim of the supervised regression model is to predict the average request response time with the minimal possible error, hence we used the standard Mean Absolute Error (MAE) as the main evaluation metric. In order to optimise the model we developed a grid search function which iterated through four different linear regression models: vanilla linear regression, ridge, lasso and elastic net. For each of the regression models we tried a range of hyperparameters.

Once the request mix is predicted our resource allocation optimisation functions create a dataset of 360 rows in which the request mix is kept constant but each possible resource allocation is a separate row. We then use the predict method on our training supervised model in order to predict average response request time and finally order by predicted average request response time. The output was initially a top 10 of the most recommended configurations but we switched to an average of the top 50 in order to see variations towards specific App recommendations over multiple runs.

VI. Experimental analysis of the auto-scaling system components

We deployed the microservice based web application on a 2 Core, 4GB ram Ubuntu VM using Docker Swarm node. We deployed the Locust.io container which generated the traffic on an identical VM. We deployed the machine learning pipeline on another VM (32 cores, 128GB ram) so that the prediction activities did not take place on the VMs participating in the experiment. The experiments involved simulating traffic to the microservice based web application in a pattern that is similar to the pattern created in the training data but compressed and, of course, subject to variation due to the randomness used in the traffic generation process. The number of concurrent users was 500 and each user made a request, on average, every 4 seconds. The Locust.io app was activated for 3 X 30

minute back to back load sessions in which requests were sent to the microservice based web application. The first session was biased towards requests which stressed App 2, then App 3 and finally App 4. This pattern roughly follows patterns present in the training data. Our experiment was broken up into four stages. During each of the stages the traffic pattern and duration was kept fixed, as were the underlying resources. The experiment stages are as follows.

1) All Apps were kept at scale 1 (a single container for each service) with no scaling intervention taken
2) App 2 was scaled to 3 while the other Apps remained at scale 1. This was the recommended scaling action
3) App 3 was scaled to 3 while the other Apps remained at scale 1. This was not the recommended scaling action
4) App 4 was scaled to 3 while the other Apps remained at scale 1. This was not the recommended scaling action

The result section focuses on the evaluation of each part of the hybrid model as well as the efficacy of the auto-scaling system as a whole. We were able to train LSTM models to a good level of accuracy with regards to the MAE loss and our custom metric. See top 10 best performing models and the bottom 10 worst performing models below:

	epochs	batches	units	layers	val_loss	loss	MAE-holdout-set	app-bias-mean-holdout-set
B1	10	64	50	6	0.07	0.08	0.06	0.59
B2	10	64	75	3	0.07	0.08	0.06	0.60
B3	10	64	50	5	0.07	0.08	0.06	0.61
B4	10	64	50	4	0.07	0.08	0.06	0.67
B5	10	64	75	5	0.07	0.08	0.06	0.70
B6	10	64	50	3	0.08	0.08	0.07	0.73
B7	10	64	75	6	0.07	0.08	0.07	0.74
B8	10	64	75	4	0.07	0.08	0.07	0.88
B9	4	128	75	2	0.08	0.09	0.07	0.92
B10	1	128	50	5	0.09	0.18	0.09	0.93

Fig. 5: Top 10 models by app-bias prediction metric

	epochs	batches	units	layers	val_loss	loss	MAE-holdout-set	app-bias-mean-holdout-set
W10	1	128	125	5	1.00	2.95	1.00	9.28
W9	1	128	125	6	1.02	1.21	1.02	8.55
W8	1	64	125	4	0.99	2.08	0.99	7.79
W7	1	64	75	4	0.43	0.88	0.43	6.50
W6	1	64	125	3	0.88	1.75	0.88	6.09
W5	1	64	125	2	0.93	1.58	0.93	5.66
W4	1	64	125	6	0.67	1.18	0.67	5.29
W3	1	128	75	6	0.53	1.52	0.53	5.28
W2	1	64	125	5	0.53	1.04	0.53	3.94
W1	1	64	75	3	0.28	0.43	0.28	1.94

Fig. 6: Bottom 10 models by app-bias prediction metric

A clear pattern that you can see in the results is that models that were trained over 10 epochs (the maximum) are over represented in the top 10 models and those that were trained over 1 epoch and over represented in the bottom 10 models. Furthermore, the best models seem to favour relatively fewer units. However we will be careful not to draw conclusive insights from these results for several reasons:

- Our grid search was not exhaustive and was limited but resource and time constraints
- Our test train split method is vulnerable to overfitting
- Our main objective is not to build a robust request prediction model but to capture the pattern in our demo application and demonstrate the pipeline

With these points in mind we will choose the 10th best model for our final model since it was trained over only one epoch and yet has broadly similar performance to other models that may be more vulnerable to overfitting due to being retrained over multiple epochs.

The model parameters include Optimizer: Adam (lr=0.05), Epochs: 1, Batches: 128, Units: 50, Layers: 5. The stacked LSTM model seemed to perform well with regards to it's standalone function of predicting the request mix of the next 250 requests. However, one of the reasons that the pipeline as a whole might have failed is the choice of prediction window. It may be that the input and output dimensions need to be extended in order to capture enough data for the supervised model to detect the signal in the data.

The next part of the hybrid model pipeline takes the predicted request mix over the next 250 requests, as predicted by the LSTM model which looks for patterns over the last 1000 requests, and attempts to predict average request response time over that predicted period. We were able to find a model with decent performance, with regards to MAE, using our grid search method. Each regression model achieved a similar level of accuracy with regards to MAE and a similar R-Squared value. See the best models in the table below:

Regression Model	R-Squared	MAE	Best parameters
Ridge	0.481418032	0.039959783	alpha': 0.01, 'fit_intercept': False, 'solver': 'sag'
Vanilla Linear	0.481417698	0.039960935	fit_intercept': True, 'normalize'=False
Lasso	0.481417697	0.039960944	alpha': 0, 'fit_intercept': True
Elastic Net	0.478523965	0.040094057	alpha': 0.0001, 'l1_ratio': 0.2, 'max_iter': 10

Fig. 7: Request response time regression model metrics

However the relatively decent error score might be indicative of a data set in which there is relatively low variance most of the time i.e. the majority of requests fall within the 0-10 second band. However there are some requests that take up 60 seconds and are then terminated. If these requests are relatively well dispersed in the data it is possible that the relationship that is present between resource allocation and length of these requests is missed by a model which is trained on a relatively small rolling window.

In order to verify this window size hypothesis we ran a supervised model on the entire static dataset but only looking at the relationship between resource allocation (containers per microservice) and request response time. We compared the coefficients and feature importance vectors of the linear and random forest models run on the rolling window and the static data. We can see that the 3rd value is relatively large in magnitude and negative as we might expect from our initial EDA which indicates App 3 as a bottleneck. However, the coefficients of some of the request type features are also of a comparable size and so the predicted request response time via permuting over resource combinations is likely to be relatively static unless large numbers are used. As mentioned above we trained a series of supervised models on just the App resource allocation features and their ability to predict average request response time at the per response level.

The results are as follows, the Static Linear Regression has a Mean Absolute Error: 0.125, Mean Squared Error: 0.029, Root Mean Squared Error: 0.171, Coefficients: (App 1) 0.0095 (App 2) 0.02 (App 3) -0.109 (App 4) -0.054. The Static Linear models coefficients with best models from grid search:

- (App 1) 0.009545 (App 2) 0.020500 (App 3) -0.109313 (App 4) -0.053224
- (App 1) 0.009510 (App 2) 0.020490 (App 3) -0.109238 (App 4) -0.053238
- (App 1) 0.009510 (App 2) 0.020490 (App 3) -0.109238 (App 4) -0.053238
- (App 1) 0.009742 (App 2) 0.017945 (App 3) -0.107034 (App 4) -0.051425

It can be observed that the linear models that App 3, and to a lesser extent 4, are strongly linked to decreasing response time. The near zero, yet positive, coefficient values for App 1 and 2 can be interpreted as taking underlying resources away from the App 3 and 4 containers as they are hosted on the same VM. The Static Random Forest results include Mean Absolute Error: 0.123, Mean Squared Error: 0.029, Root Mean Squared Error: 0.169, Feature importances: (App 1) 0.01699534 (App 2) 0.14090081 (App 3) 0.77981094 (App 3) 0.06229291. The same pattern, in terms of relative importance, is also present in the static Random Forest regression model.

VII. Conclusion and next steps

In conclusion, this work develops and tests a microservice auto-scaling research pipeline for modern systems such as fog and Internet of Things scenarios. Our work offers an evaluation of a hybrid sequence and supervised learning model and also provides a useful roadmap for developing, tuning and evaluating microservice auto-scaling solutions. The experimental analaysis demonstrates that performance of different models when modeling the microservice scaling problem.

VIII. Appendix

The App Manager and NGINX config of web app is available at *https://github.com/nathancoulson/app_manager*. The other modules of the project are available from the same repository under the names: prediction_app, locust_app, micro-app-*1/2/3/4*-bbk.

References

[1] (2019, January). [Online]. Available: https://wearesocial.com/blog/2019/01/digital-2019-global-internet-use-accelerates

[2] C. Qu, R. N. Calheiros, and R. Buyya, "Auto-scaling web applications in clouds: A taxonomy and survey," *CoRR*, vol. abs/1609.09224, 2016. [Online]. Available: http://arxiv.org/abs/1609.09224

[3] A. Taivalsaari and T. Mikkonen, "The web as an application platform: The saga continues," in *2011 37th EUROMICRO Conference on Software Engineering and Advanced Applications*, Aug 2011, pp. 170–174.

[4] H. Fernandez, G. Pierre, and T. Kielmann, "Autoscaling web applications in heterogeneous cloud infrastructures," in *2014 IEEE International Conference on Cloud Engineering*, March 2014, pp. 195–204.

[5] T. Lorido-Botran, J. Miguel-Alonso, and J. A. Lozano, "A review of auto-scaling techniques for elastic applications in cloud environments," *Journal of Grid Computing*, vol. 12, no. 4, pp. 559–592, Dec 2014. [Online]. Available: https://doi.org/10.1007/s10723-014-9314-7

[6] L. Vaquero, L. Rodero-Merino, and R. Buyya, "Dynamically scaling applications in the cloud," *Computer Communication Review*, vol. 41, pp. 45–52, 01 2011.

[7] X. Liu, J. Heo, and L. Sha, "Modeling 3-tiered web applications," in *13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, Sep. 2005, pp. 307–310.

[8] M. Villamizar, O. Garcés, H. Castro, M. Verano, L. Salamanca, R. Casallas, and S. Gil, "Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud," in *2015 10th Computing Colombian Conference (10CCC)*, Sep. 2015, pp. 583–590.

[9] S. Sotiriadis, N. Bessis, C. Amza, and R. Buyya, "Elastic load balancing for dynamic virtual machine reconfiguration based on vertical and horizontal scaling," *IEEE Transactions on Services Computing*, vol. 12, no. 2, pp. 319–334, March 2019.

[10] C. Pahl and P. Jamshidi, "Microservices: A systematic mapping study," 01 2016, pp. 137–146.

[11] A. R. Sampaio, H. Kadiyala, B. Hu, J. Steinbacher, T. Erwin, N. Rosa, I. Beschastnikh, and J. Rubin, "Supporting microservice evolution," in *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Sep. 2017, pp. 539–543.

[12] S. Sotiriadis, N. Bessis, and R. Buyya, "Self managed virtual machine scheduling in cloud systems," *Information Sciences*, vol. 433-434, pp. 381 – 400, 2018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0020025517308277

[13] S. Sotiriadis, N. Bessis, E. G. Petrakis, C. Amza, C. Negru, and M. Mocanu, "Virtual machine cluster mobility in inter-cloud platforms," *Future Gener. Comput. Syst.*, vol. 74, no. C, pp. 179–189, Sep. 2017. [Online]. Available: https://doi.org/10.1016/j.future.2016.02.007

[14] A. Gandhi, P. Dube, A. Karve, A. Kochut, and L. Zhang, "Adaptive, model-driven autoscaling for cloud applications," in *ICAC*, 2014.

[15] K. A. Nuaimi, N. Mohamed, M. A. Nuaimi, and J. Al-Jaroodi, "A survey of load balancing in cloud computing: Challenges and algorithms," in *2012 Second Symposium on Network Cloud Computing and Applications*, Dec 2012, pp. 137–142.

[16] S. Sharifian, S. A. Motamedi, and M. K. Akbari, "A predictive and probabilistic load-balancing algorithm for cluster-based web servers," *Applied Soft Computing*, vol. 11, no. 1, pp. 970 – 981, 2011. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1568494610000220

[17] S. Sotiriadis, N. Bessis, A. Anjum, and R. Buyya, "An inter-cloud meta-scheduling (icms) simulation framework: Architecture and evaluation," *IEEE Transactions on Services Computing*, vol. 11, no. 1, pp. 5–19, Jan 2018.

[18] D. Jiang, G. Pierre, and C.-H. Chi, "Autonomous resource provisioning for multi-service web applications," in *Proceedings of the 19th International Conference on World Wide Web*, ser. WWW '10. New York, NY, USA: ACM, 2010, pp. 471–480. [Online]. Available: http://doi.acm.org/10.1145/1772690.1772739

[19] A. Bhattacharyya, S. A. J. Jandaghi, S. Sotiriadis, and C. Amza, "Semantic aware online detection of resource anomalies on the cloud," in *2016 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, Dec 2016, pp. 134–143.

[20] Y. Li and Y. Xia, "Auto-scaling web applications in hybrid cloud based on docker," in *2016 5th International Conference on Computer Science and Network Technology (ICCSNT)*, Dec 2016, pp. 75–79.

[21] T. Ye, X. Guangtao, Q. Shiyou, and L. Minglu, "An auto-scaling framework for containerized elastic applications," in *2017 3rd International Conference on Big Data Computing and Communications (BIGCOM)*, Aug 2017, pp. 422–430.

[22] S. Nadgowda, S. Suneja, and A. Kanso, "Comparing scaling methods for linux containers," in *2017 IEEE International Conference on Cloud Engineering (IC2E)*, April 2017, pp. 266–272.

[23] H. Alipour and Y. Liu, "Online machine learning for cloud resource provisioning of microservice backend systems," in *2017 IEEE International Conference on Big Data (Big Data)*, Dec 2017, pp. 2433–2441.

[24] J. Peng, Z. Huang, and J. Cheng, "A deep recurrent network for web server performance prediction," in *2017 IEEE Second International Conference on Data Science in Cyberspace (DSC)*, June 2017, pp. 500–504.

[25] N. Roy, A. Dubey, and A. Gokhale, "Efficient autoscaling in the cloud using predictive models for workload forecasting," in *2011 IEEE 4th International Conference on Cloud Computing*, July 2011, pp. 500–507.

[26] X. Tang, Q. Liu, Y. Dong, J. Han, and Z. Zhang, "Fisher: An efficient container load prediction model with deep neural network in clouds," in *2018 IEEE Intl Conf on Parallel Distributed Processing with Applications, Ubiquitous Computing Communications, Big Data Cloud Computing, Social Computing Networking, Sustainable Computing Communications (ISPA/IUCC/BDCloud/SocialCom/SustainCom)*, Dec 2018, pp. 199–206.