HyScale: Hybrid Scaling of Dockerized Microservices Architectures

by

Jonathon Paul Wong

A thesis submitted in conformity with the requirements
for the degree of Master of Applied Science
The Edward S. Rogers Sr. Department of Electrical and Computer Engineering
University of Toronto

Abstract

HyScale: Hybrid Scaling of Dockerized Microservices Architectures

Jonathon Paul Wong

Master of Applied Science

The Edward S. Rogers Sr. Department of Electrical and Computer Engineering

University of Toronto

2018

Recently, microservices architectures have garnered the attention of many organizations—providing higher levels of scalability, availability, and fault isolation. Many organizations choose to host their microservices architectures in cloud data centres to offset costs. Incidentally, data centres become over-encumbered during peak usage hours and under-utilized during off-peak hours. Traditional scaling methods perform either horizontal or vertical scaling exclusively. When used in combination, however, both offer complementary benefits and compensate for each other's deficiencies. To leverage the high availability of horizontal scaling and the fine-grained resource control of vertical scaling, two novel reactive hybrid autoscaling algorithms are presented and benchmarked against Google's popular Kubernetes horizontal algorithm. Results indicated up to 1.49x speedups in response times, 10 times fewer failed requests, and 35% increase in resource efficiencies. Moreover, a novel predictive approach is also explored using neural networks and a custom online learning technique indicating prediction errors as low as 3.77%.

Acknowledgements

I would like to thank Anthony Kwan for managing and tuning experiments throughout these projects. I would also like to thank Geoff Elliott for supporting our experiments on the cluster. I would also like to thank Andrew Pelegris for his help and advice on machine learning techniques. Finally, I would like to thank Professor Jacobsen for giving me the support and flexibility throughout my time with the Middleware Systems Research Group that made pursuing my projects possible.

Contents

Chapter 1

Introduction

1.1 Motivation

Microservices architectures have gained widespread popularity in the software development community, quickly becoming a best practice for creating enterprise applications [43]. Under the microservices architecture model, a traditional monolithic application is dissociated into several smaller, self-contained component services. Three of the most notable benefits include an application's enhanced deployability, fault-tolerance and scalability. Hosting one's own microservices architecture, however, comes at a high price, including server-grade hardware acquisition costs, maintenance costs, power consumption costs, and housing costs. Instead of bearing these expenses themselves, software companies typically choose to pay cloud data centres to host their applications. Companies relinquish control of their microservices' resource allocations and run the risk of performance degradation.

Owners of these microservices architectures, known as tenants, negotiate a price for a

specified level of quality of service, usually defined in terms of availability and response times. This information is encapsulated in a document referred to as a service-level agreement (SLA). The SLA stipulates the monetary penalty for each violation and impels the cloud data centre to provision more resources to the tenants. To reduce operating costs and improve user-perceived performance, it is paramount to cloud data centres to allocate sufficient resources to each tenant.

Unfortunately, data centres are reaching their physical and financial limitations in terms of space, hardware resources, energy usage, and operating costs [15]. As such, it is not always possible to simply provision more resources as a buffer against SLA violations. In fact, this approach often results in higher costs to the data centres as the number of machines and power consumption increase [2]. Conversely, data centres can suffer from resource underutilization. During off-peak hours, tenants are typically overprovisioned resources [27]. These unused resources can be reclaimed to conserve power and be more readily allocated to another tenant when there is an immediate need. Most cloud clusters are also heterogeneous in nature, implying that machines can run at different speeds and have different memory limits. Increasing the efficiency of resource utilization on each machine, while minimizing the number of machines used, presents another way to lower the overall power consumption cost. If individual machine specifications are not taken into account, however, this can lead to overloaded machines. For example, exceeding memory limits forces the machine to swap to disk, resulting in significantly slower response times and poorer overall performance. Scaling resources efficiently for virtualized microservices should therefore be imperative for data centres as it can result in significant cost savings [48].

Traditional methods for scaling can generally be categorized into vertical or horizontal scaling with the more popular approach being horizontal scaling [8, 41, 4, 28]. This scaling technique involves replicating a microservice onto other machines to achieve high availability. By replicating a microservice onto another machine, its resource allocations are also copied over. This approach, however, is greedy and presumes there is no shortage of hardware resources [52]. Furthermore, horizontal scaling creates additional overhead on each replicated machine and is confronted with bandwidth limitations on network and disk I/O, and hardware limitations on socket connections.

Vertical scaling, on the other hand, aims to maximize the utilization of resources on a single machine by providing the microservice with more resources (e.g., memory and CPU) [46, 36, 40, 34]. Unfortunately, this method is also limited as a single machine does not necessarily possess enough resources. Upgrading the machines to meet demands quickly becomes more expensive than purchasing additional commodity machines.

Currently, most cloud data centres employ the use of popular tools and frameworks, such as Google's Kubernetes, to automatically scale groups of microservices [4]. Many of these autoscaling algorithms are devised to achieve high availability within a cluster. These frameworks, however, usually consider only one aspect of resource scaling (e.g., CPU utilization, memory consumption, or SLA adherence) and use either vertical or horizontal scaling techniques, exclusively [36, 34, 10]. Moreover, an administrator must manually reconfigure the resource allocations within their own system when the framework's algorithm does not output the optimal configuration. If allocations are left sub-optimal, higher costs are incurred leading to loss of profit [2]. For the most part, frameworks such as Kubernetes have simple autoscaling algorithms that frequently lead

to non-optimal resource allocations.

These industry state-of-the-art solutions are reactive in nature and provision the required computing resources based on observed resource utilization (e.g., CPU, memory, disk, and network I/O utilization) [46, 41, 34]. Typically, resources are provisioned such that there is always a constant percentage of unused resources reserved as contingency, in case usage increases suddenly [2]. Apart from the self-evident resource overhead used by the buffer, these solutions usually work well for non-volatile workloads that do not vary drastically. They will, however, over and under-provision resources for workloads that change more frequently and suddenly. This is due to the delays between the observations and reactions of the autoscaling system, resulting in non-optimal resource configurations.

At the other end of the spectrum, predictive approaches present an alternative that can minimize delays while also helping to converge to more optimal configurations. By predicting future resource usage, both cloud service providers and tenants could realize massive cost-savings. Solving this problem is akin to solving the time-series forecasting problem, since recent usage statistics usually indicate a continuing upward or downward trend. Time-series predictions in this context, however, are complex, as trends are highly non-linear and include several other variables, such as time of day and seasonality [39]. Moreover, no two tenants will share the same workload trends, since their applications and client demographics vary immensely. These trends are also often extremely sensitive to external factors, such as major world or political events.

To realize a highly available data centre, a comprehensive hybrid scaling solution incorporating a combination of both reactive and predictive approaches would be required. In such a system, a more reactive approach would be appropriate during stable loads,

while during unstable loads, a predictive approach would be more highly weighted. In this way, the reactive approach could quickly adjust to small fluctuations in resource demand, and the predictive approach could preallocate resources in the advent of much larger fluctuations.

1.2 Approach

As steps towards actualizing this predictive-reactive hybrid scaling solution, this thesis presents two novel reactive hybrid scaling solutions and a novel predictive approach. Due to the lack of hybrid scaling systems, an architecture capable of supporting hybrid reactive and predictive scaling algorithms is also presented. The following two sections describe the reactive and predictive approaches.

1.2.1 Reactive Hybrid Scaling Approach

We propose and investigate the possibility of hybrid scaling techniques for allocating resources within a data centre. Hybrid scaling techniques reap the benefits of both the fine-grained resource control of vertical scaling, and the high availability of horizontal scaling. This makes hybrid scaling a promising solution for effective autoscaling.

Several challenges arise, however, when designing a hybrid scaling algorithm. Finding an optimal solution with hybrid scaling can be viewed as a complex multidimensional variant of the bin packing problem. When presented with a limited number of physical resources and a set of microservices with variable dimensions (e.g., memory, CPU), finding the optimal configuration is an NP-complete problem [12, 32, 29]. These resource

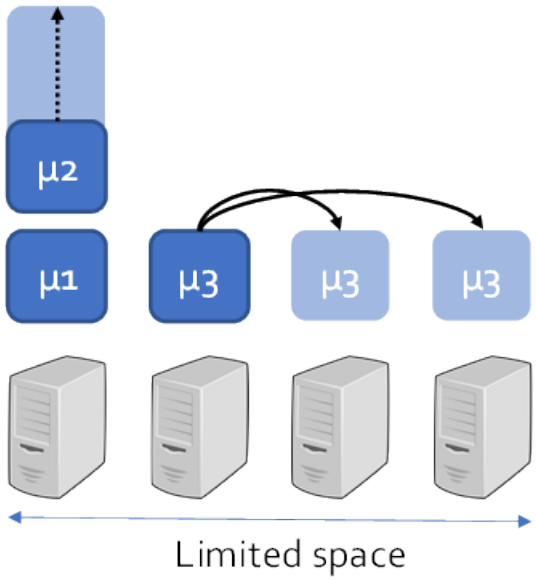


Figure 1.1: Illustrates a hybrid autoscaling scenario. Container 1 and 2 (left) reside on the same machine. Container 2 is vertically scaled while Container 3 is horizontally scaled.

allocations and reconfigurations must be determined in real-time, thus limiting the time spent searching the solution space. Moreover, when user load is unstable, an aggressive algorithm can induce successive conflicting decisions (i.e., thrashing), leading to scaling overhead. As the exact correlation between resource allocations and performance metrics is unclear, the definition of the optimal configuration is ambiguous making closed-form calculations of the optimal configuration difficult.

1.2.2 Predictive Approach

A popular technique for performing time-series predictions is using a variant of artificial neural networks (ANN), known as a recurrent neural network (RNN) [21]. ANNs are known for their ability to model highly non-linear problems, as well as their ability to generalize what has been seen in the past. RNNs also retain these traits with the added

perk of being able to remember longer term trends. One particular RNN, known as the long short term memory (LSTM) network, has shown great promise in time-series forecasting [20].

Most solutions only train the model once using an offline profile of the workload that consists of previous data from the same workload or data from related workloads [47, 55, 30]. The first option may be hampered by a lack of training data resulting from the reliance on just the history of one workload. Moreover, profiles must be created separately for each workload. In the second option, other workloads may not actually be representative of the workload being predicted. Regardless, for optimal results, the offline profile must be representative of future trends. In a practical setting, however, it is impossible to capture all of the trends for each tenant, as well as the external factors related to them in advance.

One solution is to pre-train a predictive model once as a baseline, and update it reactively to real-time data for each workload. This is referred to as online learning. Online learning would enable the model to adapt to changing trends in the workload, effectively allowing it to "learn" specific trends for a particular tenant and thus reducing prediction errors. Although this promises huge theoretical benefits, these are difficult to achieve in practice. Despite enabling the model to learn new and emerging trends, ANNs tend to "forget" old observations and trends, replace them with just the latest ones over time if online learning is performed naively [54]. This is a source of instability and can have an overall negative impact on prediction accuracies.

In this paper, we attempt to push the boundaries on workload prediction by first examining various baseline ANN prediction models. We, then, further improve prediction

performance by applying our own custom online learning technique.

1.3 Contributions and Organization

To address these issues, this thesis makes the following contributions:

1. Performance analysis of horizontal versus vertical scaling of Docker containers to quantitatively assess their trade-offs (Chapter 4).

2. Design and implementation of two novel reactive hybrid scaling techniques, HyScale$_{\text{CPU}}$ and HyScale$_{\text{CPU+Mem}}$ (Chapter 5).

3. Design and implementation of an autoscaling platform prototype to evaluate and compare various scaling techniques (Chapter 6).

4. Validation of the performance and resource efficiency benefits gained through the use of hybrid scaling by benchmarking HyScale against Google's Kubernetes horizontal autoscaling algorithm on microbenchmarks and Bitbrain's VM workload (Chapter 7).

5. Design of a custom online learning technique for predictive scaling (Chapter 8).

6. Comparison of various ANN models' prediction performances to create baselines for the online learning technique (Chapter 9.1).

7. Analysis of our online learning technique and its parameter space (Chapter 9.2).

Chapter 2

Machine Learning Background

This section serves as a high-level overview of ANN concepts that should be understood before moving forward.

2.1 ANNs

Loosely inspired by biological brains, ANNs consist of a collection of inter-connected nodes. These nodes are grouped into 3 categories of layers known as the input layer, hidden layers, and output layer. Nodes from one layer are typically fully connected to the nodes of the proceeding layer, in a feed-forward fashion. Each connection from one node to another is associated with a randomly initialized weight. As data flows from the input nodes to the hidden nodes and finally to the output nodes, the network attempts to update the weights of the network, resulting in its ability to "learn". The ANN's ability to model non-linearities derives from placing non-linear activation functions, such as sigmoid or exponential linear unit (ELU) functions.

2.2 RNNs and LSTMs

Unlike ANNs which are feed-forward, RNNs allow for feedback connections and storing of internal state for nodes. This enables the network to process sequences of inputs and thus is a strong candidate for time-series predictions.

LSTM units are a building unit for RNN layers [37]. An LSTM unit typically consists of a cell, an input gate, an output gate, and a forget gate. Depending on the variant of LSTM used, the connections between cell and gates differ, but their responsibilities remain the same. The cell "remembers" values over periods of time, while the gates act as regulators over the flow of information within the unit. Each gate can be thought of as a conventional ANN node, computing a non-linear activation function of a weighted sum.

2.3 Convolutional Neural Networks

Convolutional neural networks (CNN) refer to ANNs that consist of at least one convolutional layer [45]. The goal of the convolutional layer is to apply filters on the input to detect various features. For each filter, a convolution operation is applied to the input before passing the result to the next layer. Since these filters are "learned", the network is able to discover features on its own. CNNs are commonly used in image and video processing, but can be used as generic feature detectors in any dimension (i.e., 1D, 2D, 3D, etc.).

2.4 Training Networks

Although there are many different ways to train a network, we will only consider the most basic technique, known as supervised learning. Supervised learning consists of passing an input through the network model and comparing the resulting output with the corresponding true output, known as a label. The difference between the model output and label are then passed to an optimizer to determine which weights within the network caused the error. The optimizer then applies changes to the weights in the reverse direction of the error gradient. This process is known as an iteration and is repeated until the model converges to an optimum. Generally, this is performed over the entire training dataset multiple times, where each pass is known as an epoch. In practice, models are trained offline, since training can take anywhere from hours to days to complete and is very problem-specific.

Due to the nature of the complex matrix computations of each iteration, GPUs are leveraged to accelerate processing times [44]. Furthermore, iterations can be batched to take advantage of the inherent multi-threadedness of GPUs. The batch and model size, however, are constrained by the limited memory available on GPUs. Moreover, RNNs and LSTMs struggle to benefit from GPUs, since unrolling is unable to make full use of concurrency and consumes large amounts of memory [11].

2.5 Parameter and Hyperparameter Tuning

To optimize a particular ANN model, there are usually several parameters that must be tuned and modified. This includes generic parameters, such as batch size, activation

functions, learning rate, and number of epochs. For specific models even more parameters can be introduced, such as the number of hidden units and time steps unrolled for LSTMs or the number and shape of filters for CNNs. Altogether, these result in an expansive parameter search space, especially when models are combined together. Generally speaking, the tuning of parameters affect a model's capacity and speed of convergence. A model's capacity refers to its ability to learn highly non-linear trends and is typically affected by parameters, such as the number of hidden units and activation functions. On the other hand, a model's speed of convergence is usually affected by parameters such as learning rate and optimizer.

Chapter 3

Related Work

Although container-based virtualization technology is relatively new, virtual machines (VMs) have been well researched and widely used for several decades [41]. Dynamic scaling of hardware resources has been approached from the VM perspective and is not a foreign concept in the world of virtualization [46, 41, 22, 26].

3.1 Reactive Approaches

3.1.1 Vertical Scalers

VMs typically benefit greatly from vertical scaling, as compared to horizontal scaling, since they have long start up times. Thus, several reactive vertical scaling solutions exist in the VM domain, such as Azure Automation, CloudVAMP, and AppRM.

Azure Automation supports vertical scaling of VMs based on observed resource utilization [46]. Scaling, however, is relegated to tenants to perform manually. Azure bills tenants based on the amount of resources they have been allocated, thus encouraging

tenants to scale downwards. From the cloud centre perspective, this model does little to manage resource efficiency and fragmentation across the cluster. Moreover, the cloud centre is very susceptible to overprovisioning of resources, since tenants are required to scale their services manually, which is slow for a reactive solution.

CloudVAMP is a platform that provides mechanisms for memory oversubscription of VMs [36]. Memory oversubscription allows a VM to utilize more memory than the host has available (i.e., vertical scaling). This is made possible via the hypervisor retrieving unused memory from VMs co-located on the machine, and allocating it to VMs in need. CloudVAMP, however, does not support scaling through other types of resources, such as CPU or disk I/O, forfeiting the ability to achieve a more granular level of resource management. Similarly, AppRM vertically scales VMs based on whether current performance is meeting user-defined SLAs [34]. The system periodically monitors performance of each VM, comparing them to SLAs and vertically scaling them accordingly. Currently, this only supports CPU and memory scaling.

Several other works on VM scaling exist, however, they perform vertical scaling and horizontal scaling, exclusively [22, 26].

There are far fewer vertical scaling solutions for containers, however, due to their propensity for replication. A notable example of exclusive vertical scaling for containers is ElasticDocker [10]. ElasticDocker employs the MAPE-K loop to monitor CPU and memory usage and autonomously scales Docker containers vertically. It also performs live migration of containers, when the host machine does not have sufficient resources. This approach was compared with the horizontally scaling Kubernetes, and shown to outperform Kubernetes by 37.63%. The main flaw with this solution is the difference in

monitoring and scaling periods between ElasticDocker and Kubernetes. ElasticDocker polls resource usage and scales every 4 seconds, while Kubernetes scales every 30 seconds, giving ElasticDocker an unfair advantage to react to fluctuating workloads more quickly. Moreover, the cost of machines with sufficient hardware to support a container with high demands far exceeds the cost savings achieved.

Another example of a container vertical scaler is Spyre [40]. This framework splits resources on a physical host into units called slices. These slices are allocated a variable amount of CPU and memory resources and house multiple containers (similar to Kubernetes pods). Vertical scaling is then performed on these slices to allocate or deallocate resources to a group of containers. Unfortunately, resources are shared amongst containers within a slice making fine-grained adjustments difficult.

3.1.2 Horizontal Scalers

Although VM scaling usually does not benefit from horizontal scaling as much as vertical scaling due to long start up times, there exist various VM horizontal scaling solutions.

OpenStack provides users with the ability to automate horizontal scaling of VMs via Heat templates [41]. To achieve this, OpenStack provides two mechanisms that are user-defined: Ceilometer Alarms and Heat Scaling Policies. Ceilometer Alarms trigger based off observed resource usage (e.g., CPU usage exceeding a certain percentage), and invoke Heat Scaling Policies to instantiate or decommission VMs. Although containers are supported by OpenStack, users tend to use other container orchestrators in combination with OpenStack [7].

For container-based horizontal autoscaling, the most popular tools are Docker Swarm and Google's Kubernetes. Docker Swarm takes a cluster of Docker-enabled machines and manages them as if they were a single Docker host [8]. This allows users to horizontally scale out or scale in containers running within the cluster. Scaling commands, however, must be input manually and is far too slow to react to sudden load variations. Kubernetes offers a horizontal autoscaler that monitors average CPU utilization across a set of containers and horizontally scales out or scales in containers to match the user-specified target CPU or memory utilization [4]. It also attempts to provide a beta API for autoscaling based on multiple observed metrics [5]. This, however, does not actually perform scaling based on all given metrics. After evaluating each metric individually, the autoscaling controller only uses one of these metrics.

3.1.3 Hybrid Scalers

To our knowledge, there are very few container-based hybrid scalers in comparison to VM-based hybrid scalers.

SmartScale uses a combination of vertical and horizontal scaling to ensure that the application is scaled in a manner that optimizes both resource usage and reconfiguration costs incurred due to scaling [17]. Scaling is performed in two steps. First, the number of VM instances is estimated based on observed throughput. once the number of instances is determined, an optimal resource configuration is found using a binary search. Optimality is defined with respect to maximizing savings and minimizing performance impact. This approach assumes that each VM instance operates at maximum utilization.

In the cost-aware approach of J. Yang et al., an extension of R. Han et al.'s VM work is presented by including a horizontal scaling aspect to the algorithm [53, 22, 23]. The scaling method is divided into three categories: self-healing scaling, resource-level scaling, and VM-level scaling. The first two methods are vertical scaling, while the last method is horizontal scaling. Self-healing allows complementary VMs to be merged, while resource-level scaling consumes unallocated resources on a machine. Finally, VM-level scaling is performed using threshold-based scaling.

The self-adaptive approach also attempts to perform hybrid scaling of VMs by first vertically scaling where possible, then allocating new VM instances when required [25]. If a service within a VM instance requires more CPUs and there are CPUs available on the node, they are allocated to the VM and all services within that VM. If no VM instance with those resources available exist, a new VM is started. While this approach is interesting, the implementation limits the solution's scaling granularity as only whole virtual CPUs can be allocated or deallocated to a VM at any given time. Moreover, since the resources are allocated to the VM itself, the resource distribution within the VM cannot be controlled for each service or is not discussed.

Four-Fold Auto-Scaling presents a hybrid autoscaling technique to reduce costs for containers within VMs [24]. It models the scaling problem as a multi-objective optimization problem and minimizes cost using IBM's CPLEX optimizer. To simplify the search space, their model discretizes VM and container sizes into distinct, enumerated resource types and abstracts physical resource types. This forces a trade-off between the granularity of their scaling decisions and the complexity of their optimization problem. Furthermore, there are no guarantees on the optimality of the solutions generated. In

their implementation, they consider CPU and memory, and have shown a 28% reduction in costs. There are, however, negligible improvements in performance and SLA adherence. Furthermore, this approach requires manual tuning and fails to expose the performance and resource utilization ramifications.

Jelastic monitors CPU, memory, network and disk usages to automatically trigger either vertical or horizontal scaling for a single application [3]. Although Jelastic supports both types of scaling, it does not support them simultaneously. For vertical scaling, when user-specified thresholds are exceeded, Jelastic provisions set amounts of resources, known as Cloudlets, to the application. A cloudlet consists of a 400MHz CPU and 128MiB of memory. For horizontal scaling, the user must define the triggers and scaling actions for their application. For example, the user must specify the number of replicas to scale up by when exceeding their memory threshold. This approach lacks flexibility as users cannot use both simultaneously, and must manually tune their triggers and scaling actions, especially under varying and unstable loads.

Although several hybrid scaling solutions do exist, none of them are designed specifically for container-based ecosystems. Most of these approaches use VMs which are inherently different from containers. As VMs have inherently higher resource overhead and scaling costs compared to containers, frequent and responsive scaling actions are incompatible with a VM dominated cloud environment. Containers, on the other hand, do not suffer from these same constraints and therefore are contingent on different concerns than VM machine scaling. Similar techniques from VM scaling could be leveraged, but must be modified to conform to the container ecosystem.

3.2 Predictive Approaches

3.2.1 Statistical Models

Peter Bodik et al. investigate the feasibility of using statistical machine learning models to automate control in data centres [13]. Their work uses linear regression to predict the next 5 minutes of incoming client requests. These predictions are then fed into another model to estimate the number of server machines to employ to improve latency performance. This is a simplification of the workload prediction problem, as their scope is limited to horizontally-scalable Internet services, and not applicable to services where the underlying resource usage is more relevant (e.g., vertically-scalable services). ARIMAWorkload presents a similar workload prediction model which instead employs an Autoregressive Integrated Moving Average (ARIMA) model to predict the number of incoming requests before predicting the number of VMs to allocate [14].

Resource Central utilizes machine learning techniques to predict VM resource utilization trends and applies it to Azure's VM scheduler [16]. This work uses Microsoft Azure's public cloud dataset to train their random forest and extreme gradient boosting tree classifiers, and fast fourier transforms. This, however, relies on the offline profiling of the VM workload characteristics to improve accuracies. Moreover, they focus on VM placement rather than resource scaling, which is shown in their prediction outputs.

3.2.2 Neural Network Models

Feng Qiu et al. present a solution which utilizes deep learning techniques, namely the deep belief network (DBN) model, to predict VM CPU resource utilization [38]. Qazi

Ullah et al. also present two different resource prediction models, ARIMA and auto-regressive neural network (AR-NN), and compare their performance in predicting VM CPU resource utilization[49].

Weishan Zhang et al. utilize basic RNN cells to create workload prediction models to predict Google's CPU and RAM workload traces [55]. As the authors state, their method is unqualified for long-term time-series predictions, but could be alleviated using LSTMs.

Binbin Song et al. demonstrate the feasibility of LSTM networks in predicting data centre resource allocation [47]. They also show the flexibility and accuracy of LSTMs over two data centre datasets, Google and grid/HPC. This work, however, attempts to predict actual load several steps ahead and only predicts non-overlapping segments of input data.

The adaptive differential evolutional approach by Jitendra Kumar et al. showcases their own proposed multi-layer neural network model for predicting the number of HTTP requests in two separate HTTP traces [30]. They then compare its prediction error to the error of standard ANNs.

Although, the described works attempt to predict VM workload using various machine learning techniques, many simplify the problem by narrowing the scope and breadth of the workload applications. Moreover, their predictions are made to be less granular than what is required for resource management systems to perform fine-grained adjustments (e.g., number of VMs rather than resource utilization). Those that employ ANNs to overcome these issues train their models offline without considering online techniques to further reduce errors.

3.3 Predictive-Reactive Approaches

B. Urgaonkar et al. propose a horizontal scaling predictive-reactive solution for multi-tiered internet applications [50]. In their solution, they attempt to scale the number of servers to use at every tier. Their predictive algorithm operates at the timescale of hours to days, and monitors and stores the arrival rate of requests for each day. This is used to create a probability distribution of how many requests to expect at a given hour. If the predictions are consistently underestimated for several hours straight, corrective measures are invoked to handle the observed workload. On the other hand, the reactive algorithm operates at the minute timescale to handle unpredictable events, such as flash crowds. This is performed using a threshold-based technique.

RPPS presents a predictive-reactive scaler, where the predictive algorithm employs an ARIMA model and smoothing to predict CPU usages [18]. It is also claimed that their approach uses both horizontal and vertical scaling for VMs. However, this paper only elaborates on their predictive approach and its use of horizontal scaling.

Similarly, DoCloud uses horizontal scaling of containers to perform predictive and reactive scaling out, but only predictive scaling in [28]. This is to ensure that scale ins are not executed prematurely, resulting in oscillations in the number of containers. The predictive algorithm also employs an ARMA model to predict the number of requests per second, and converts that prediction to the number of containers needed. Its reactive algorithm is also threshold-based.

Chapter 4

Analysis of Horizontal versus Vertical Scaling

To motivate the hybridization of scaling techniques, the effects of horizontal and vertical scaling must be understood. Conducive to this, we stressed CPU-bound and memory-bound microservices under a fixed client load and measured their response times. As a baseline, we first measured the response times of each microservice on its own with full access to a 4 core node. Subsequent runs entailed manually varying resource allocations to simulate equivalent vertical and horizontal scaling scenarios. The following experiments were run with 400 client requests on up to 4 machines:

4.1 CPU Scaling

A container's Docker CPU shares define its proportion of CPU cycles on a single machine [9]. By tuning the CPU shares allocated to each microservice, we effectively control

their relative weights. For example, if two microservice containers run on a single machine with CPU shares of 1024 and 2048, the containers will have 1/3 and 2/3 of the access time to the CPU, respectively. We utilized this sharing feature to induce a form of vertical scaling, as increasing or decreasing shares directly correlate with an increase or decrease in CPU resource allocation to a container.

The baseline microservice is run on a single machine with no contention and is configured to calculate 20000 prime numbers for each request. The latency of a request is measured as the time it takes for the microservice to perform the prime number calculation task. This simulates CPU load on the system from the request/response framework that is inherent in a microservices architecture.

To create contention of CPU resources, the microservice is run alongside another container. This container runs a third party program, *progrium stress*, which consumes CPU resources by performing *sqrt(rand())* calculations [33]. In both the horizontal and vertical scaling scenarios, the microservice is given an equivalent amount of resources overall to isolate the effects of both. For example, in the vertical scaling emulation, we allocated 1024 CPU shares to both the microservice and the progrium stress container, splitting CPU access time equally between the two. Assuming that nodes have 4 CPU cores, both are provisioned 2 CPU cores worth of access time. An equivalent horizontally-scaled resource allocation with 3 microservices running over 3 machines allocates 1024 and 5120 CPU shares to the microservice and the progrium stress container, respectively. This results in 1/6 of the CPU access time of each node for each microservice, again totaling to 2 CPU cores worth of access time (i.e., 1/6 of the 12 CPU cores). This equivalence was reproduced with 2 microservices and 2 nodes, and 4 microservices and 4 nodes.

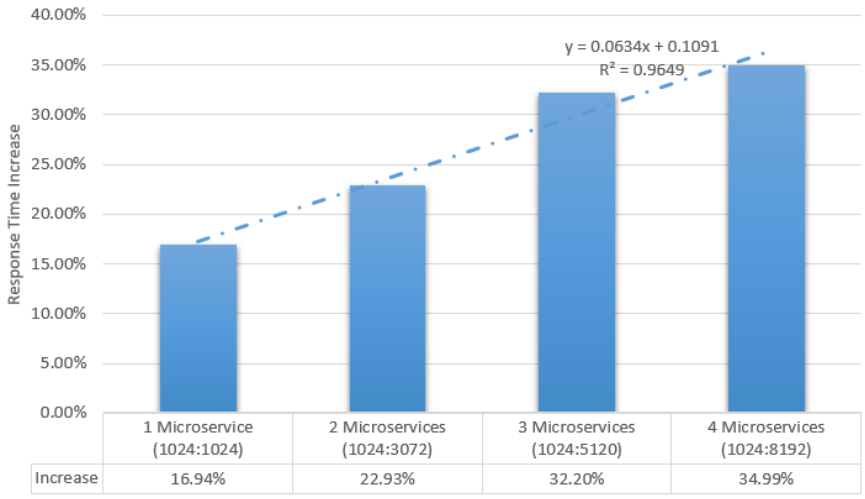


Figure 4.1: Response time percentage increases of vertical and horizontal scaling in contention with respect to vertical scaling without contention. The leftmost bar represents vertical scaling (down) with contention, while the bars to its right represent horizontal scaling with contention to various degrees.

Results of our studies on vertical versus horizontal scaling for CPU resources (see Figure 4.1) show a preference for vertical scaling. It provided the fastest request processing times when compared to the equivalently horizontally scaled instances. Results also indicate that more replicated instances decrease overall CPU performance. Although Docker containers, themselves, have negligible overhead [19], when contention over shared CPU resources is introduced, significant overhead becomes apparent. In our experiments, this manifested itself as a 17% increase in response times. This would be further exacerbated by the presence of more co-located containers. Moreover, the applications within the Docker containers also incur measurable costs. When replicated several times, this performance overhead becomes much more significant, and can affect response times. For our experiments, this overhead resides mainly within the Java Virtual Machine.

4.2 Memory Scaling

Docker also allows users to impose memory limits on containers [9]. Once a container uses memory in excess of its limit, portions of its memory are swapped to disk.

The effects of vertical and horizontal scaling on memory were tested analogously to the CPU tests, ensuring equivalent resources in each scenario (i.e., one 512 MB container is equivalent to two 256 MB containers). One difference, however, is that there is no contention for memory between Docker containers, and thus there was no need to run a progrium stress container. Additionally, microservices consumed and persisted memory directly during requests, rather than consuming CPU.

Results show negligible differences in request times between vertical and horizontal scaling scenarios. Also, increasing memory limits did not speed up processing times. Performance drastically degraded, however, when the number of incoming requests forced the microservice to swap. Moreover, horizontal scaling introduced slightly more memory overhead. This was due to the memory used by the application and the container image, itself. This overhead will vary from application to application. If high enough, horizontally scaled instances are much more likely to swap compared to a single vertically scaled instance, given the same amount of memory.

Chapter 5

Autoscaling Algorithms

To help understand our novel hybrid scaling algorithms, we first discuss the implementation details of the popular Kubernetes horizontal scaling algorithm. All variables in the following equations are measured as a percentage.

5.1 Kubernetes Horizontal Scaling Algorithm

The Kubernetes autoscaling algorithm utilizes horizontal scaling to adjust the number of available replica instances of services to meet the current incoming demand or load. The algorithm increases and decreases the number of replicated microservice instances based on current CPU utilization. If the average CPU utilization for a microservice and all its replicas exceed a certain target percentage threshold, then the system will scale up the number of replicas. Conversely, when the average CPU utilization falls below the threshold, the system will scale down the number of replicas. CPU utilization is calculated as the CPU usage over the requested CPU (i.e., the previously allocated

CPU resources). In order to measure average CPU utilization, the Kubernetes algorithm periodically queries each of the nodes within the cluster for resource usage information. By default, the query period is set to 30s, however, for our experiments, we query every 5s.

The Kubernetes autoscaling algorithm takes in 3 user-specified inputs: overall target CPU utilization, and minimum and maximum numbers of replicas. The system scales up and down towards the minimum and maximum whenever the overall CPU utilization is above or below the target, respectively. The algorithm calculates the target number of replicas to scale up or down for microservice m and its replica r using the formula:

$$utilization_r = \frac{usage_r}{requested_r}$$

$$NumReplicas_m = \lceil \frac{\sum_r utilization_r}{Target_m} \rceil$$

This, however, introduces a problem where thrashing can occur. To prevent thrashing between quickly scaling up and scaling down horizontally, the Kubernetes algorithm uses minimum scale up and scale down time intervals. Rescaling intervals are enforced when a scaling up or scaling down operation occurs, and notifies the system to halt any further scaling operations until the specified time interval has passed. Our experiments used 3s and 50s minimum scale up and scale down intervals, respectively.

There is another Kubernetes feature that mitigates thrashing. It only performs rescal-

ing if the following holds true:

$$|\frac{\sum_r usage_r}{NumReplicas_m * Target_m} - 1| > 0.1$$

Recently, Kubernetes has added support to use memory utilization or a custom metric instead of CPU utilization. Kubernetes has also attempted to provide support for multiple metrics, which is currently in beta. This support however is limited, as only the metric with the largest scale is chosen.

5.2 Hybrid Scaling Algorithms

The main goal of our hybrid autoscaling algorithms is to dynamically reallocate and redistribute resources amongst microservices in a fashion that preserves high availability, low response times and high resource utilization per node. Some microservices tend to use a mix of different resources, and cannot be scaled effectively when using Kubernetes, leading to longer response times and more SLA violations. Horizontal scaling is not always the best solution as the addition of a new replica instance may be grossly more than required. Additionally, horizontally scaling microservices that need to preserve state is non-trivial as it introduces the need for a consistency model to maintain state amongst all replicas. Hence, in these scenarios, the best scaling decisions are those that bring forth more resources to a particular container (i.e., vertical scaling).

Our hybrid autoscaling algorithm takes a similar approach to the Kubernetes autoscaling algorithm. As opposed to calculating only a coarse-grained target number of

replicas for a microservice, the hybrid approach is to deterministically calculate the exact microservice's needs. While still retaining the desired coarse-grained replication factor, this calculation also contains the required fine-grained adjustments. Two main distinctions separate our hybrid algorithms from Kubernetes: the use of vertical scaling, and the broadening of the measurement criteria to include both CPU and memory. These algorithms first ensure the minimum and maximum number of replicas are running for fault-tolerance benefits. They then attempt to vertically scale onto the same machines, granted enough available resources. If there are insufficient resources to meet demands, horizontal scaling is performed on another machine; one not hosting the same microservice, and advertising sufficient available resources. In the following sections, we present two such hybrid algorithms.

5.2.1 HyScale$_{\mathbf{CPU}}$ Algorithm

This hybrid algorithm considers only CPU usage and calculates the number of missing CPUs for microservice m using the equation:

$$MissingCPUs_m = \frac{\sum_r usage_r - (\sum_r requested_r * Target_m)}{Target_m}$$

If the overall CPU usage is equal to that of the target utilization, then the equation will output 0 signifying that no changes are required. If the result is negative, then the resource allocation is greater than the usage and signals to the algorithm that there are unused CPU resources for this microservice. Similarly, a positive result signifies that there are insufficient resources allocated to the microservice overall.

Once the number of missing resources has been calculated for every microservice, the algorithm enters the resource reclamation phase. For every microservice that indicated a negative value, downward vertical scaling (i.e., resource reclamation) is attempted on each of their replicas to move the instance towards the target utilization. If an instance has been vertically scaled downwards and its allocated resources drop below a minimum threshold (currently set to 0.1 CPUs), it is removed entirely. Moreover, any reclaimed resources contribute to increasing the number of missing resources back to 0. The amount of CPU resources reclaimable from each instance is calculated as follows:

$$ReclaimableCPUs_r = requested_r - \frac{\sum_r usage_r}{Target_m * 0.9}$$

Once reclamation is complete, the second phase of the algorithm attempts to acquire unused or reclaimed resources for microservices that indicated a positive number of missing resources. In a similar manner, each microservice instance is vertically scaled upwards by the following amount:

$$RequiredCPUs_r = \frac{\sum_r usage_r}{Target_m * 0.9} - requested_r$$

$$AcquiredCPUs_r = min(RequiredCPUs_r, AvailableCPUs_n)$$

Each replica instance will claim as many resources as it needs, up to the amount available on the node. If vertical scaling on all replicas does not provide sufficient resources to provide for the microservice as a whole, horizontal scaling is performed onto other

nodes that have free resources. Furthermore, a new replica can only be instantiated if the node advertises at least the baseline memory requirement of the microservice, as well as a minimum CPU threshold (currently set to 0.25 CPUs). This is to ensure that an instance is not spawned with resource allocations that are too small.

After each replica's new CPU requirements have calculated, Docker CPU shares are recalculated to actualize the scaling effects on each node. Empirically, CPU limiting was found to be less accurate when CPU shares were set to values further from the default value of 1024. Consequently, we aimed to scale the recomputed CPU shares to be averaged around 1024. This was achieved by forcing the total number of CPU shares on each node to be 1024 times the number of replicas it currently housed. The new CPU shares for each replica are computed using the following equations:

$$totalCPUShares_n = numReplicas_n * 1024 CPU shares$$

$$newRequestedCPUs_r = requested_r - ReclaimableCPUs_r + AcquiredCPUs_r$$

$$newCPUShares_r = \frac{newRequestedCPUs_r}{numCPUs_n} * totalCPUShares_n$$

Finally, similar to Kubernetes, the hybrid algorithm enforces rescaling intervals, whereby frequent horizontal rescaling is throttled to avoid thrashing. Vertical scaling, however, is exempt from this rule, as vertical scaling must perform fine-grained adjust-

ments quickly and frequently.

5.2.2 HyScale$_{\text{CPU+Mem}}$ Algorithm

This hybrid algorithm extends from the previous algorithm by also considering memory and swap usage. The algorithm and equations used are analogous to those used for CPU measurements and are shown below. Note that the following equations do not include CPU share recalculations since memory limits are not relative weightings and can be set directly.

$$MissingMem_m = \frac{\sum_r usage_r - (\sum_r requested_r * Target_m)}{Target_m}$$

$$ReclaimableMem_r = requested_r - \frac{\sum_r usage_r}{Target_m * 0.9}$$

$$RequiredMem_r = \frac{\sum_r usage_r}{Target_m * 0.9} - requested_r$$

$$AcquiredMem_r = min(RequiredMem_r, AvailableMem_n)$$

With the consideration of a second variable, horizontal scaling becomes much less trivial. The algorithm can no longer indiscriminately remove a container that is consuming memory or CPU, if it falls below a certain CPU or memory threshold, respectively. Furthermore, new containers cannot be added with no allocated memory or CPU. This changes the conditions for container removal and addition by requiring the CPU and

memory threshold conditions to be met mutually.

Chapter 6

Hybrid Autoscaling Architecture

To benchmark various scaling techniques on a common platform, we present an autoscaler architecture that supports vertical, horizontal and hybrid scaling. The autoscaler performs resource scaling on microservice containers, where a central arbiter autonomously manages all the resources within a cluster. Much like Apache YARN [51], this central entity is named the Monitor and is tasked with gathering resource usage statistics from each microservice running in the cluster. The Monitor interacts with each machine through the Node Managers (NMs). Each NM manages and reports the status of its machine and checks for microservice liveness. Additionally, distributed server-side Load Balancers (LBs) act as proxies for clients interacting with microservices. The different components in our autoscaling platform architecture are illustrated in Figures 6.1 and 6.2. Further details on each component are covered in following sections.

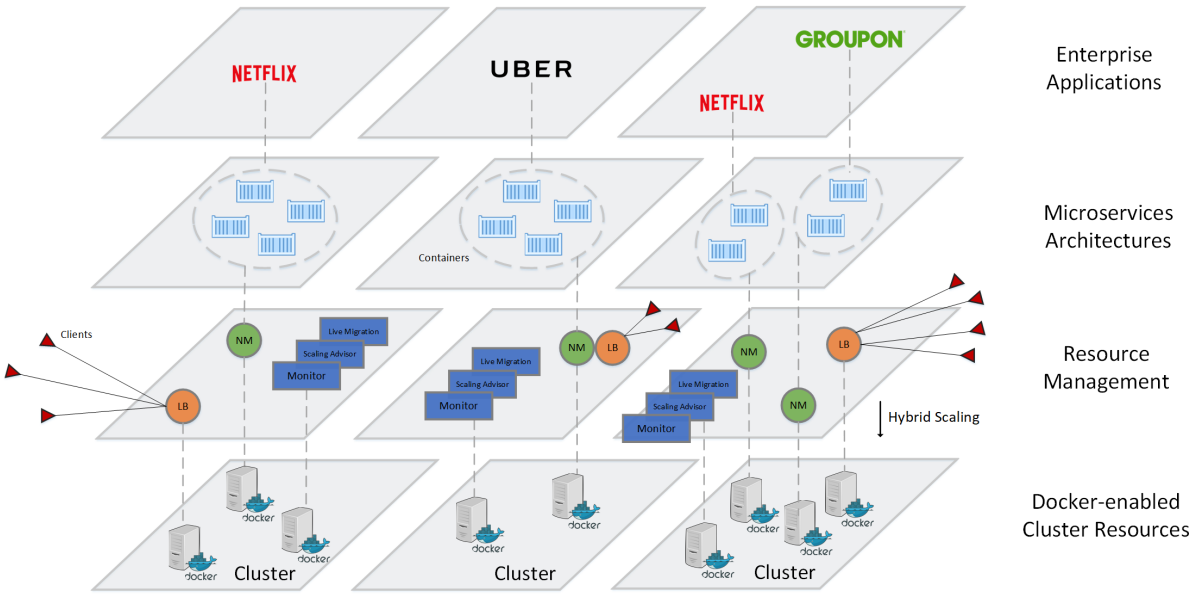


Figure 6.1: Full stack overview of microservices architectures showing our resource management architecture (second bottom layer) above the cloud data centre resources (bottom layer). Docker containerized microservices (second top layer) run alongside NMs and give rise to entreprise applications (top layer) that clients interact with.

6.1 Docker Containers/Microservices

Docker containers provide a lightweight virtualization alternative, allowing developers to package their application and its corresponding dependencies into a single isolated container. Instead of including the entire operating system into the virtual image, Docker utilizes operating system virtualization to emulate the operating system kernel. This makes Docker images significantly lighter in size, and quick to deploy onto machines. This differs significantly from traditional hypervisor virtualization, whereby hardware resources (e.g., CPU, memory, and hard disk) are emulated by the hypervisor, and guest operating systems are installed directly above. Furthermore, Docker containers leverage a copy-on-write filesystem to make the container images lightweight and compact, increasing their efficiency in deployment.

In industry, applications are deployed as groups of microservices (i.e., microservices architectures or Docker pods) and each microservice component is housed within its own container. These microservices are typically associated with an SLA defined by some availability metric. In our architecture, each Docker container houses a single microservice and is the unit of deployment for all microservices, including differing instances and replications. Each microservices' associated SLAs are defined by average request response times and can be modified to consider other metrics. We consider each microservice to be an individual entity and not part of a group of microservices, to isolate the effects of the scaling techniques.

6.2 Node Managers

Each node runs a single NM, in charge of monitoring the combined microservice resource usage of all microservices stationed on that node. NMs are also in charge of aggregating container failure information and request statistics, such as completion times, from all microservices. The NMs are written in Java and interface with the Docker daemon through *docker-java*, an open source Java library that provides a Java interface to Docker's API. NMs also gather relevant resource usage information (i.e., CPU and memory usage) through the Docker API via the '*docker stats*' command.

Additionally, the NM receives vertical scaling resource commands from the Monitor for specific containers or microservices. NMs perform these adjustments by invoking the '*docker update*' command through the *docker-java* interface. They have no control over vertical scaling decision-making for the node upon which they reside, as they only have

sufficient information to make locally optimal configurations. This can result in suboptimal global configurations. For example, the NM being unaware of any horizontal scaling decisions made by the Monitor allows the NM to vertically scale the microservice at its own discretion. This creates situations where the NM and the Monitor simultaneously increase and decrease allocated resources to a microservice, which then result in large oscillations around the target resource allocation. Moreover, the NM can also act to negate or lessen the intended effects of the Monitor. Therefore, the decision-making logic for resource allocation resides solely with the Monitor and not the NMs for our scaling architecture. Figure 6.2 illustrates the various interactions that the NM performs with other components in the architecture.

6.3 Monitor

The Monitor is the central arbiter of the system. The Monitor's centralized view puts it in the most suitable position for determining and administering resource scaling decisions across all microservices running within the cluster. The Monitor's goal is to reclaim unused resources from microservices, and provision them to microservices that require more. Scaling can be performed at the node level by the reallocation of resources to co-located containers (i.e., vertical scaling), and at the cluster level by the creation and removal of microservice replicas (i.e., horizontal scaling). The use of different scaling algorithms is also supported through communication to the Autoscaler module, and can be specified at initialization or through the command-line interface. This allows the architecture to permit for any autoscaling algorithm to be implemented and employed,

regardless of whether it is reactive, predictive, predictive-reactive, and/or hybrid. To supplement the accuracy of resource scaling decisions, the Monitor gathers resource usage and request statistics from periodic NM heartbeat requests. Currently, we have implemented Google's Kubernetes CPU scaling algorithm along with HyScale$_{\text{CPU}}$ and HyScale$_{\text{CPU+Mem}}$.

6.4 Load Balancers

The Load Balancers (LBs) are in charge of distributing incoming load amongst the various replicas of microservice instances, as a result of horizontal scaling actions. The LBs store connection information (i.e., IP address) about microservices and their replicas. To simplify the management of all microservices and their replicas, instantiation is performed at the Monitor. Subsequently, the Monitor forwards the microservice's connection information to the LBs. For LBs to track microservice replications, a naming convention is enforced by the Monitor. Each microservice is given a unique identifier consisting of the microservice name, and a replica number suffix delimited by a hyphen. For example, if two replicas of a '*helloworld*' microservice are instantiated, then they would possibly be identified as *helloworld-1* and *helloworld-2*.

When a client wishes to send a request to the microservice, the client contacts the server-side LB, first. Subsequently, the IP address of a microservice instance is sent back to the client. After this, the client connects to the service itself and sends its requests directly. In the event of request failures, clients simply resend their requests to the LB to be redirected to a potentially new instance.

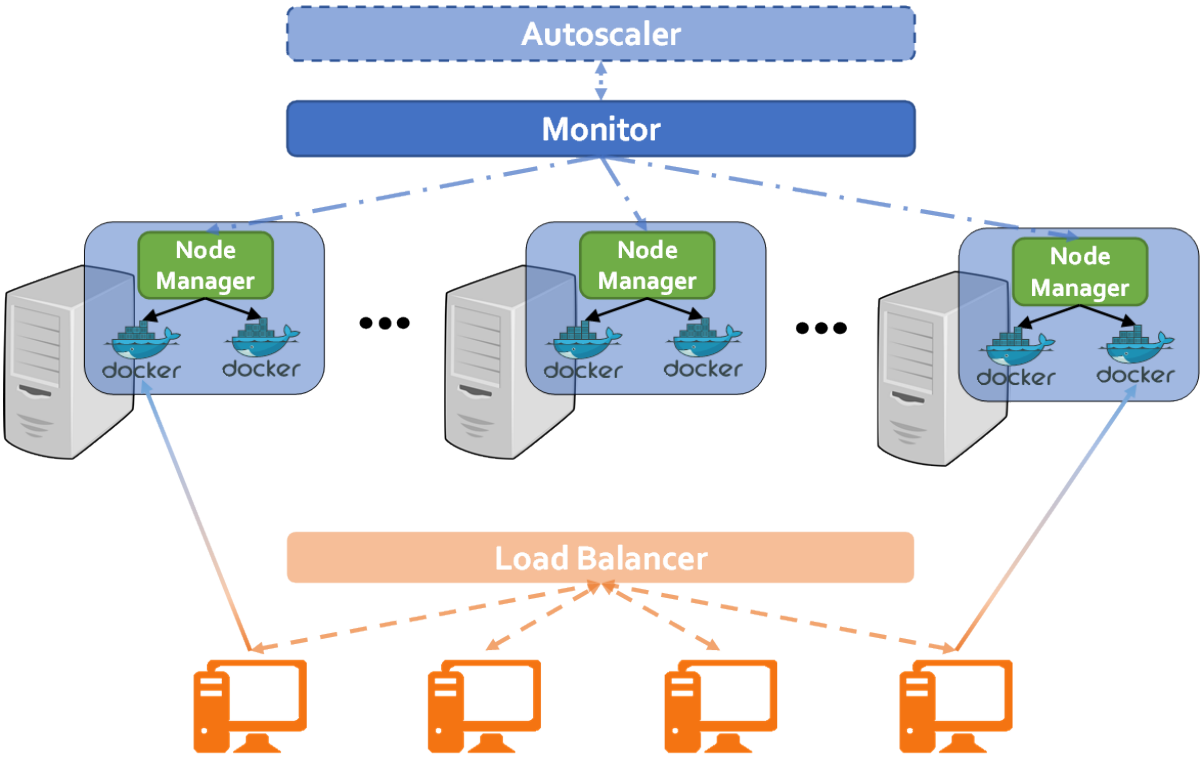


Figure 6.2: Architecture overview illustrating the various interactions between each component. The Monitor interacts with the NMs, which manage the Docker containers on a single machine. Different scaling algorithms from the Autoscaler make resource scaling decisions based on resource usage information and the Monitor instructs the NMs to execute the appropriate changes.

Currently, the LBs have random and Least Recently Used (LRU) load balancing implemented, where LRU selects the least recently used microservice replica for a client to contact. Empirically, we have found that LRU provides the most consistent results for all our experiments.

Chapter 7

Experimental Analysis

Conducive to validating the benefits of hybrid scaling techniques, we evaluate and compare the Kubernetes CPU horizontal scaling algorithm with our HyScale algorithms on our autoscaler platform. To evaluate the effectiveness of each algorithm, we look at metrics regarding user-perceived performance, SLA violations, and resource utilization.

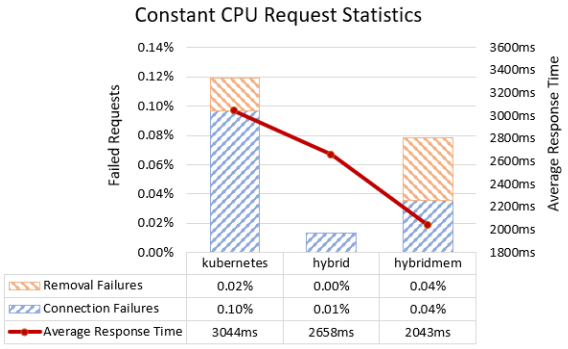
We chose to evaluate each algorithm under various types of loads and initial configurations that would encapsulate typical scenarios most data centres would experience. For client load, we emulate peak and off-peak "hours" to analyze how the algorithms react under stable and unstable loads. For our experiments, the stable load consists of a constant number of requests every second, labelled *constant*, and the unstable load forms a spiking pattern, labelled *wave*. This wave pattern simulates repeated peaks and troughs in client activity.

We also present the system with 3 different types of microservices: CPU-bound, memory-bound, and mixed. Microservices applications' workloads are emulated using a custom Java microservice with configurable workload. Upon instantiation, our emulated

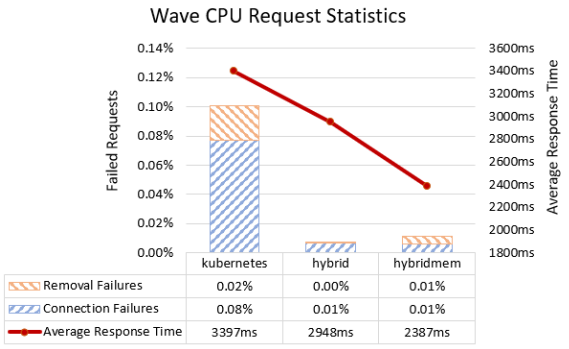
microservices take *amountCPU* and *amountMemory* as input, which is analogous to the additional number of computing cycles and memory that the microservice consumes per incoming client request, respectively. More specifically, *amountCPU* is the upper bound of prime numbers that are calculated, effectively consuming CPU resources. Based on these two inputs, we can create microservices that vary in resource consumption. Additional computing resource types, such as disk I/O and network I/O, are also supported, however, they are not currently implemented and will be part of future works.

For each client request in our *wave* and *constant* experiments respectively, we configured the CPU-bound microservices to compute between 12500 and 17000 prime numbers, the memory-bound microservices to consume 5MB, and the mixed microservices to compute between 9500 and 15000 prime numbers and consume 1MB. The additional memory consumption per request is transient and only released once the entirety of the request is complete. These values were tuned to the specs of our machines. Overall, there are 6 independent experiments each prefixed by a load name and suffixed with the microservice type.

Each experiment was performed using 15 different microservices, each initially consuming approximately 100MB, for an hour on a cluster of 12 nodes with the Monitor on a separate machine. Each cluster node runs Ubuntu 14.04 LTS and the exact same computing hardware, with 2 dual core Intel Xeon 5120 processors (4 cores in total), 8GB of DDR2 memory and 3Gbit/s SAS-1 hard drives. Five cluster nodes were designated as LBs and all other nodes hosted the NMs and Docker containers. All results were averaged over 5 runs. Since the Kubernetes and HyScale$_{\text{CPU}}$ algorithms are unable to handle memory-bound loads and crash, these results have been omitted from the fol-



(a)



(b)

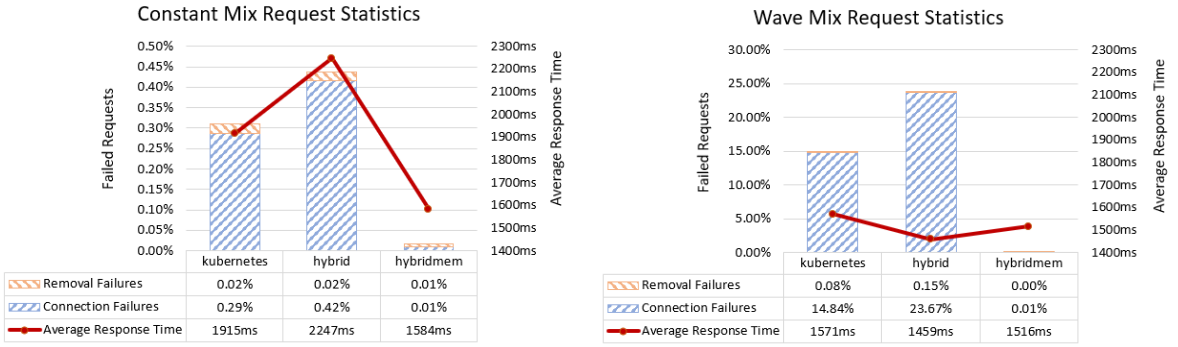
Figure 7.1: Graphs depicting the percentage of requests failed and the average request response times for the CPU-bound experiments. Removal failures are requests that end prematurely due to container removals. Connection failures are requests that fail prematurely at the microservice due to high load/traffic.

lowing sections. In the following figures and tables, 'hybrid' refers to HyScale$_{CPU}$ and 'hybridmem' refers to HyScale$_{CPU+Mem}$.

7.1 User-perceived Performance Metrics

Average microservice response times and number of failed requests were analyzed to determine the effectiveness of the scaling algorithms. Faster user-perceived response times reflect well on the resource allocations, whereas slower times reflect the opposite. Figures 7.1 and 7.2 show the average response times and percentage of failed requests of each algorithm for each experiment.

In the CPU-bound experiments (Figures 7.1a and 7.1b), HyScale$_{CPU+Mem}$ has the fastest response times overall, while Kubernetes has the slowest response times. There are clear improvements in response times of HyScale as compared to Kubernetes resulting in 1.49x and 1.43x speedups for the constant and wave workloads, respectively.

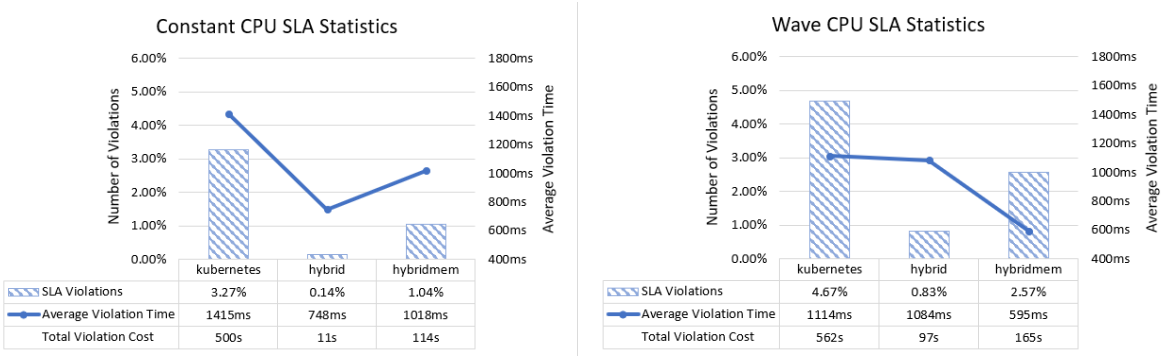


(a) (b)

Figure 7.2: Request statistics graphs for mixed resource experiments. Note the significant difference in failed requests between 7.2a and 7.2b.

Although availability is generally very high (at least 99.8% up-time), it is evident that HyScale drastically lowers the number of failed requests (up to 10 times fewer compared to Kubernetes). This shows our HyScale algorithms' high availability and robust performance under stable and unstable CPU loads.

In the mixed experiments (Figures 7.2a and 7.2b), Kubernetes and HyScale$_{CPU}$ showed significant percentages of failed requests, mainly due to the lack of consideration for memory usage. These numbers are positively offset by the partial CPU usage when forced to swap to disk. An interesting observation is made in Figure 7.2a, where Kubernetes appears to perform better than HyScale$_{CPU}$. This is caused by HyScale$_{CPU}$'s preference to vertical scaling over horizontal scaling. As an unintentional side effect of Kubernetes' aggressive horizontal scaling, more memory is allocated with each container scale out allowing it to perform better with memory requests. In Figure 7.2b, the response times are significantly skewed for Kubernetes and HyScale$_{CPU}$, since the microservices are effectively handling a smaller portion of requests (up to 23.67% less requests).

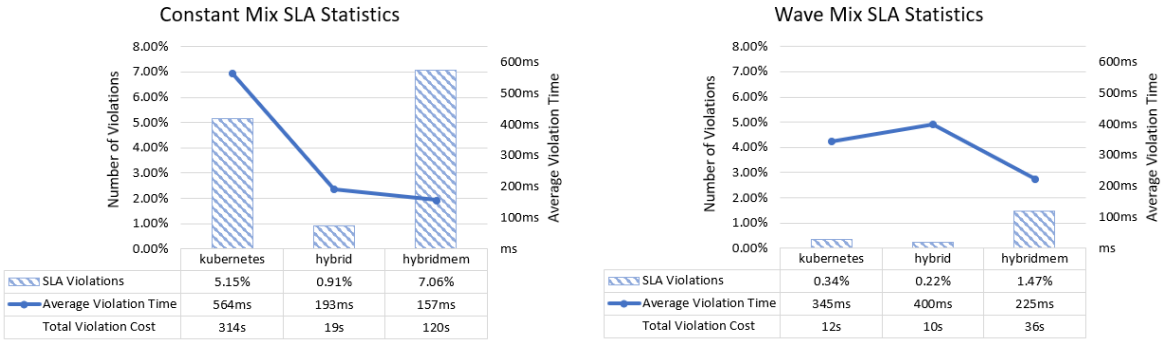


(a) (b)

Figure 7.3: Graphs depicting the total number of violations and the average degree of violation for CPU-bound experiments.

7.2 SLA Violations

Each microservice was also associated with a SLA based on average response times relative to their workload types. Failed requests did not contribute to SLA violations. The number of SLA violations and their overall cost to the provider were tracked. Figures 7.3 and 7.4 show the number of SLA violations and violation costs of each algorithm for each experiment. With the exception of the wave mix experiment (Figures 7.3 and 7.4a), both HyScale algorithms violate less SLAs on average and for less time overall. HyScale$_{\text{CPU+Mem}}$, however, is outperformed by HyScale$_{\text{CPU}}$. This is caused by extra *docker update* commands being issued to the containers to maintain memory limits (see Figure 7.5). Although the average response times are faster, these container updates are forcing requests to take slightly longer, subsequently violating more SLAs for small periods of time. To mitigate overly frequent updates to containers, an operator can increase the threshold and scaling buffer parameters. Nevertheless, these induced total violation costs for HyScale$_{\text{CPU+Mem}}$ do not exceed an order of magnitude difference relative to HyScale$_{\text{CPU}}$.



(a) (b)

Figure 7.4: SLA violation graphs for mixed resource experiments.

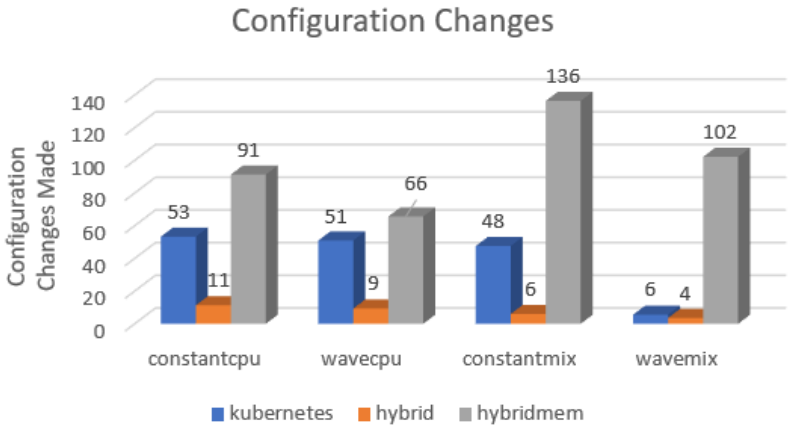


Figure 7.5: Graph depicting the number of configuration changes made by the Monitor.

In the wave mix experiment (Figure 7.4b), HyScale$_{CPU+Mem}$ causes the most violations. This comparison is invalidated since a large portion of requests failed for HyScale$_{CPU}$ and Kubernetes, effectively lightening the load on the machines. With less requests to process, meeting SLA requirements became much easier and therefore makes this an unfair comparison for HyScale$_{CPU+Mem}$. Nonetheless, HyScale$_{CPU+Mem}$ maintains a relatively high degree of SLA adherence.

7.3 Resource Utilization

By analyzing the percentage of resources in use, the efficiency of each algorithm's ability to utilize a machine's available resources can be discerned. To this extent, we are interested in the resource efficiency with respect to the average response time of each algorithm. To analyze this utilization efficiency, we use the ratio of response times (Figures 7.1 and 7.2) to average resource usages (Tables 7.1 to 7.4) as a comparison metric and refer to it as response time-resource efficiency (*RTE*). A lower *RTE* is desirable and indicates that less time is spent waiting for a response, for each unit of resource in use. Note that this is not a metric for evaluating power efficiency, and thus higher resource utilization may be desirable as long as it contributes to faster response times. In our experiments, CPU and memory usage were monitored for each machine. Resource usage statistics were polled every second on each node using *sar*, a Linux system activity monitoring tool.

Although Kubernetes had the highest resource usage for most experiments (Tables 7.1, 7.2, and 7.3), Kubernetes exhibited the highest *RTE*s. On the other hand, HyScale$_{\text{CPU+Mem}}$ had the lowest *RTE*s (approximately 35% more efficient than Kubernetes). This is explained by Kubernetes' low min and high max node usages, which show that the algorithm is inefficiently stacking several microservices onto some machines, while underutilizing others. These results are consistent with the mixed experiments (Tables 7.3 and 7.4), with the exception of the wave mixed experiment, where resource usage is much lower than HyScale$_{\text{CPU+Mem}}$ due to several failed requests. Ultimately, this further corroborates our HyScale algorithms' ability to efficiently utilize resources

Table 7.1: Averaged CPU usage statistics across all nodes for the constant CPU experiment. Response time to average usage ratio is shown on the right, where the bolded number represents the highest efficiency.

	CPU Usage (%)			
	Average	Min	Max	RTE_{CPU} (ms/%)
Kubernetes	50.34	21.56	83.75	60.47
Hybrid	44.72	40.89	53.18	59.43
HybridMem	45.94	35.96	54.95	**44.47**

Table 7.2: Averaged CPU usage statistics for the wave CPU experiment.

	CPU Usage (%)			
	Average	Min	Max	RTE_{CPU} (ms/%)
Kubernetes	37.38	8.98	67.13	90.88
Hybrid	34.21	26.10	39.90	86.17
HybridMem	35.41	32.63	41.19	**67.41**

Table 7.3: Averaged CPU and memory usage statistics across all nodes for the constant mixed experiment. Response time to average usage ratio is shown on the right, where the bolded numbers represent the highest efficiencies.

	CPU (%)			Memory (%)		
	Avg	Min	Max	Avg	Min	Max
Kubernetes	38.86	24.08	62.21	42.54	27.78	50.79
Hybrid	36.25	33.42	42.56	41.00	28.21	48.61
HybridMem	37.48	34.72	46.81	41.03	28.17	48.10

	RTE_{CPU} (ms/%)	RTE_{Mem} (ms/%)
Kubernetes	49.28	45.02
Hybrid	61.99	54.80
HybridMem	**42.26**	**38.61**

Table 7.4: Averaged CPU and memory usage statistics for the wave mixed experiment.

	CPU (%)			Memory (%)		
	Avg	Min	Max	Avg	Min	Max
Kubernetes	16.30	9.72	21.58	42.08	31.66	47.82
Hybrid	19.98	17.03	23.11	41.04	27.68	48.70
HybridMem	24.22	22.13	28.74	42.16	29.35	50.36

	RTE_{CPU} (ms/%)	RTE_{Mem} (ms/%)
Kubernetes	96.38	37.33
Hybrid	73.02	**35.55**
HybridMem	**62.59**	35.96

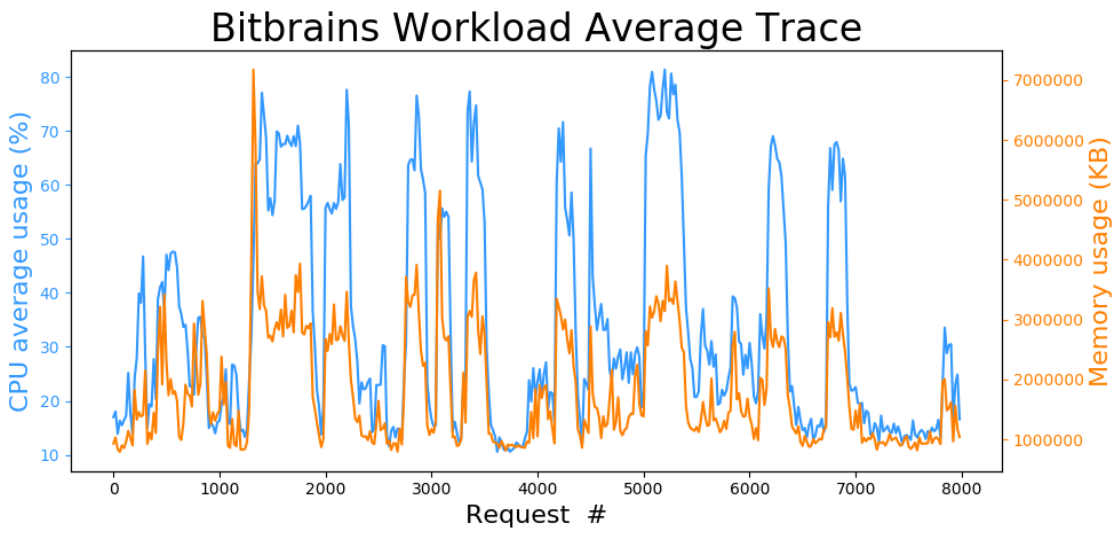


Figure 7.6: Graph of the Bitbrains Rnd workload trace for CPU and memory usage averaged over all microservices.

and achieve better performances.

7.4 Bitbrains Workload

To emulate the stress that a microservices architecture would undergo in a realistic cloud data centre, we benchmarked the algorithms using the **Rnd** dataset from the GWA-T-12 Bitbrains workload trace [1]. Bitbrains is a service provider that specializes in managed hosting and business computation for enterprises. Customers include many major banks (ING), credit card operators (ICS) and insurers (Aegon). The **Rnd** dataset consists of the resource usages of 500 VMs used in the application services hosted within the Bitbrains data centre. We re-purposed this dataset to be applicable to our microservices use case and scaled it to run on our cluster. This trace (see Figure 7.6) exhibits the same behaviour as the *constant mix* and *wave mix* workloads, and thus is expected to manifest the same result trends.

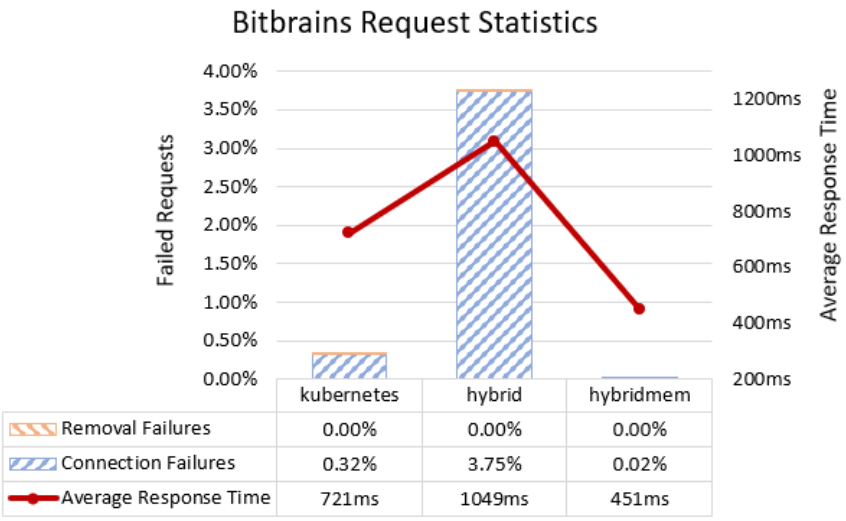


Figure 7.7: Request statistics graph for the Bitbrains experiment.

Table 7.5: Averaged CPU and memory usage statistics for the Bitbrains experiment.

	CPU (%)			Memory (%)		
	Avg	Min	Max	Avg	Min	Max
Kubernetes	10.57	5.66	33.19	21.01	19.41	22.63
Hybrid	12.52	4.90	67.48	20.75	19.43	22.09
HybridMem	8.08	6.05	15.40	21.30	19.50	23.19

	RTE_{CPU} (ms/%)	RTE_{Mem} (ms/%)
Kubernetes	68.21	34.32
Hybrid	113.34	68.39
HybridMem	**55.82**	**21.17**

The performance results (see Figure 7.7 and Table 7.5) were similar to the mixed experiment results (see Figure 7.2). HyScale$_{\text{CPU+Mem}}$ performs the best because of its ability to scale both CPU and memory. Kubernetes, however, outperformed the HyScale$_{\text{CPU}}$ because of its preference to horizontally scale, whereas HyScale$_{\text{CPU}}$ prefers to vertically scale. Kubernetes' horizontal scaling actions inadvertently allocated more memory to each replica, which reduced the number of timed out requests, as well as, the amount of memory swapped to disk.

Chapter 8

Predictive Machine Learning Approach

8.1 Baseline ANN Models

8.1.1 Data Input and Output

To train and evaluate our ANNs, the **fastStorage** dataset from the GWA-T-12 Bitbrains workload trace [1] was used. Bitbrains is a service provider that specializes in managed hosting and business computation for enterprises. Customers include many major banks (ING), credit card operators (ICS) and insurers (Aegon). **fastStorage**, consists of workload traces of 1,250 VMs that were used in software applications hosted within the Bitbrains data centre. These traces are stored as csv files with the schema shown in Table 8.1. Each file contains a month's worth of entries and each row within is separated by 5 minutes. As seen in Figures 8.1 and 8.2, the Bitbrains dataset encapsu-

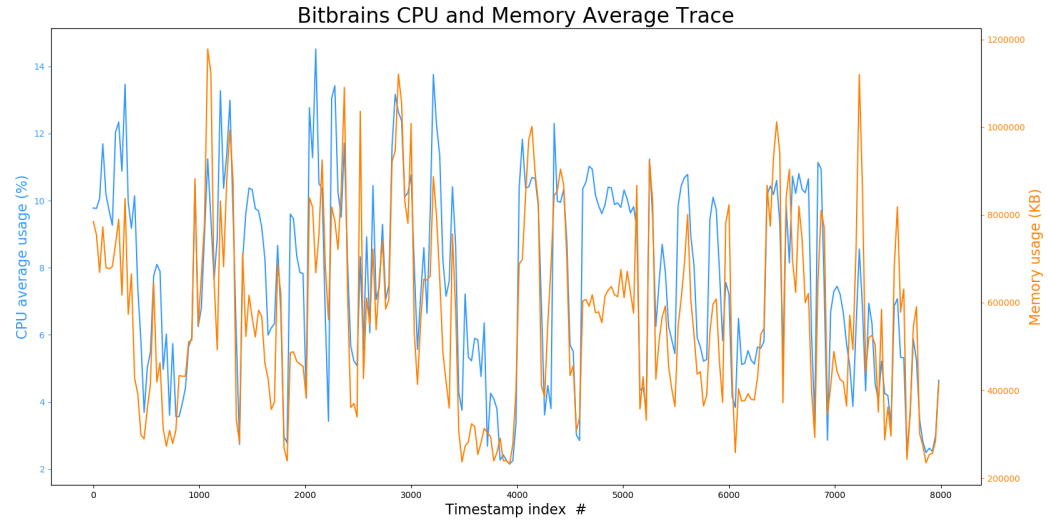


Figure 8.1: An illustration of the average CPU and memory utilization across all of the 1,250 Bitbrains fastStorage VMs.

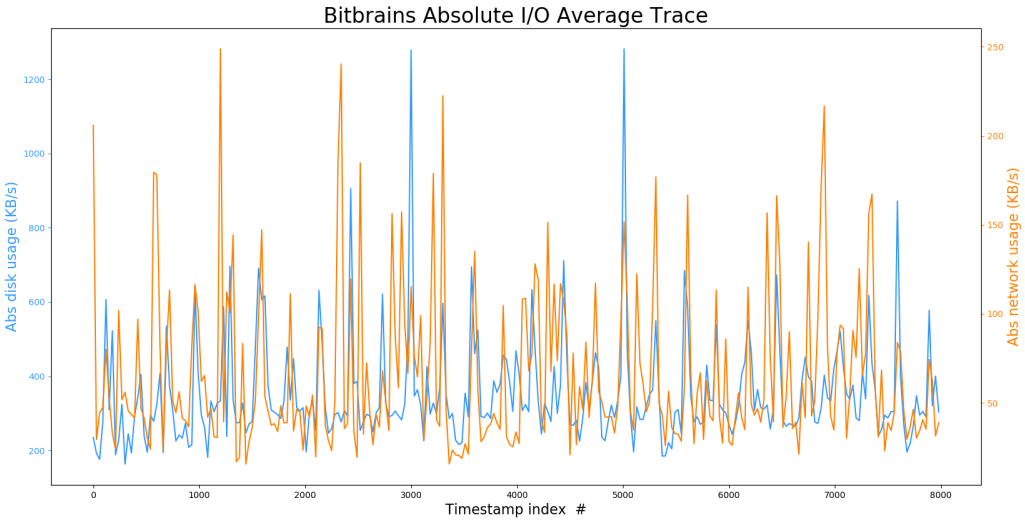


Figure 8.2: An illustration of the absolute value of the average I/O utilization across all of the 1,250 Bitbrains fastStorage VMs. These values were computed using $input + output$.

lates many different types of applications with workloads that are very bursty in nature making generalizations difficult.

In order for the data to match our criteria for inputs and labels, we rearranged the

Table 8.1: Schema of the GWA-T-12 Bitbrains fastStorage dataset

	Schema	
Index	Name	Description
0	Timestamp	Number of milliseconds since start of the trace
1	CPU cores	Number of virtual CPU cores provisioned
2	CPU capacity provisioned (MHZ)	The capacity of the CPUs in terms of MHZ
3	CPU usage (MHZ)	Utilization of the CPU in terms of MHZ
4	CPU usage (%)	Utilization of the CPU in terms of percentage
5	Memory provisioned (KB)	The capacity of the memory of the VM in terms of KB
6	Memory usage (KB)	The memory that is actively used in terms of KB
7	Disk read (KB/s)	Disk read throughput in terms of KB/s
8	Disk write (KB/s)	Disk write throughput in terms of KB/s
9	Network in (KB/s)	Network received throughput in terms of KB/s
10	Network out (KB/s)	Network transmitted throughput in terms of KB/s

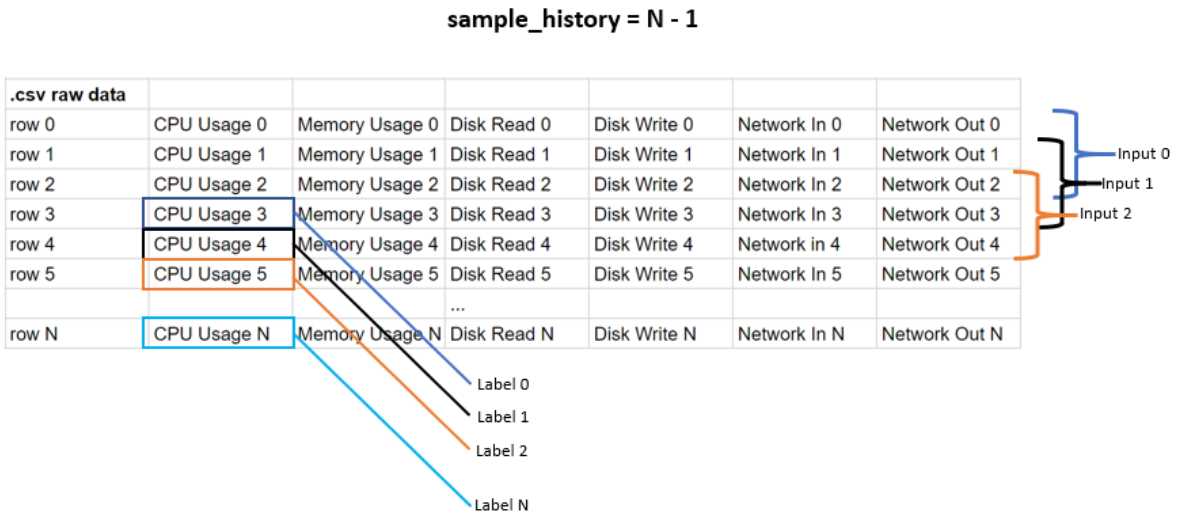


Figure 8.3: An illustration of data input and labels with a sample history $N-1$ ($5*(N-1)$ minutes).

raw input data. Since our goal is to predict future CPU utilization based on past resource utilization trends, the data needs to be presented as an on-going time series. In order to do so, multiple rows of data are cascaded. We refer to the number of rows cascaded as the *sample history*, since these values represent data that has already been observed. Empirically, a larger sample history leads to better predictions. Due to limited memory

constraints, diminishing returns, and difficulty in obtaining resource usage histories for extended periods of time, however, we set the sample history to 36 rows (i.e., 3 hours).

For the labels, the rows after the cascaded sample history are cascaded together. In this paper, we are interested in only the next 5 minute interval, so the next row is chosen as the label. This window of 37 consecutive cascaded rows is used as a sliding window to generate input-label pairs for each row in every csv file. These pairs are then fed into the ANN models for training and testing to generate predictions. A visualization for the schema of the inputs is shown in Table 8.2 and Figure 8.3.

Table 8.2: Schema of the preprocessed inputs and label for a sample_history of 10.

Inputs/Features	
Index	Name
0	CPU-usage-0 (%)
1	memory-usage-0 (%)
2	disk-read-0 (KB/s)
3	disk-write-0 (KB/s)
4	network-in-0 (KB/s)
5	network-out-0 (KB/s)
6	CPU-usage-1 (%)
7	memory-usage-1 (%)
8	disk-read-1 (KB/s)
9	disk-write-1 (KB/s)
10	network-in-1 (KB/s)
11	network-out-1 (KB/s)
	...
54	CPU-usage-9 (%)
55	memory-usage-9 (%)
56	disk-read-9 (KB/s)
57	disk-write-9 (KB/s)
58	network-in-9 (KB/s)
59	network-out-9 (KB/s)
Corresponding Label	
Index	Name
0	CPU-usage-10 (%)

Although there are many works that apply many data preprocessing and filtering techniques to improve prediction accuracies, we avoid these as many of them are tailored and tuned to specific workload characteristics. This reduces the model's generalizability and therefore would hinder our online learning technique.

8.1.2 Baseline ANN Models

Using the data input and output format described above, we highlight and describe different ANN models that are trained "offline" in this paper. For the following models, a rectified ELU (RELU) activation function and Adam optimizer are used. Other parameters and hyperparameters, such as learning rate and batch size, are set to 0.001 and 256, respectively.

Standard ANN

First, we train the most basic ANN consisting of a single fully connected hidden layer (see Figure 8.4). This serves as a hard baseline as to what accuracies can be achieved using ANNs. For these experiments, we set the number of nodes in the layer to 128.

LSTMs

A single standard LSTM cell with 128 units is also trained, since they are known for their strengths in performing time-series predictions. This is expected to perform significantly better than the standard fully connected ANN, at the cost of larger memory footprints and longer processing times. We also train a variant of the LSTM known as a layer normalized-LSTM (LN-LSTM). Layer normalization in RNNs significantly reduces

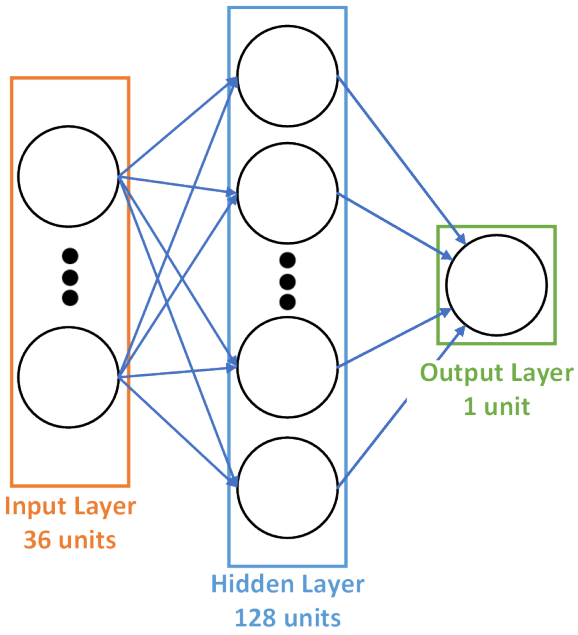


Figure 8.4: An illustration of a standard ANN. This ANN has 36 input nodes (one for each past data point), a hidden layer with 128 units, and an output unit for predictions.

training time [31].

Multi-layered LN-LSTM

Taking the LN-LSTM model, we strive to further enhance the LSTM's performance by adding more LN-LSTM layers to the model (i.e., making it deeper). For this model, we add a second identical LN-LSTM layer. This provides the model a larger capacity to learn trends, while not consuming excessive amounts of memory.

Convolutional Multi-layered LN-LSTM

Finally, we combine convolutional layers with the multi-layered LN-LSTM (see Figure 8.5), inspired by the CLDNN model [42]. By first passing the input through two 1D convolutional layers, the input is preprocessed before the LN-LSTM layers. The con-

volutional layers effectively reduces frequency variance in the input, while the LSTM layers model the input in time.

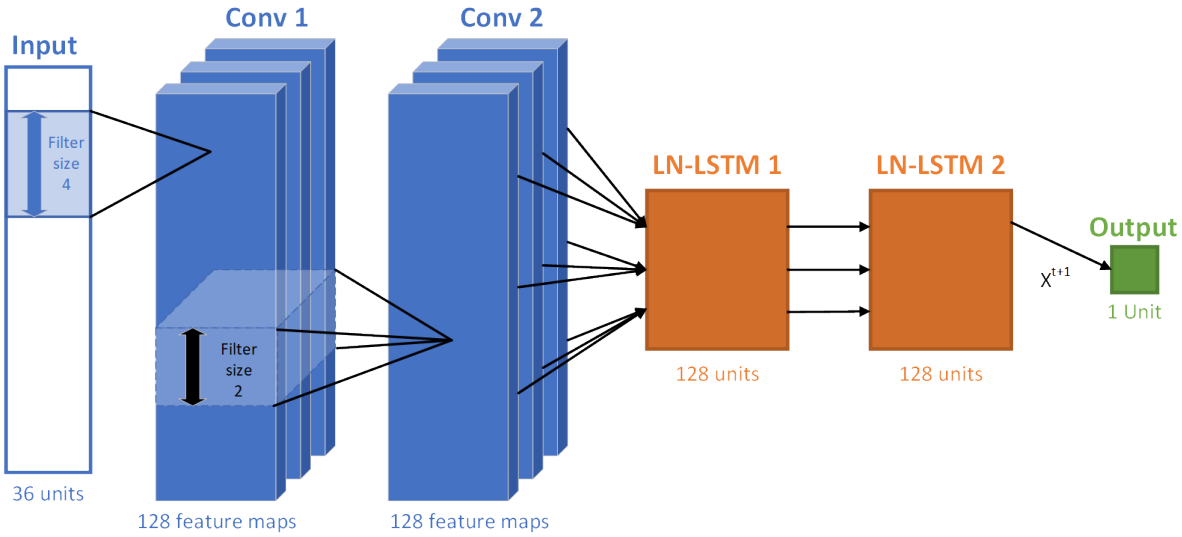


Figure 8.5: Convolutional LN-LSTM model with 2 1D convolutional layers and 2 LN-LSTM layers. Filter sizes of the convolutional layers are 4 and 2, respectively.

8.2 Online Learning Technique

In the standard case of ANN learning, after a model has been trained, the model is validated by using a separate validation or test dataset. The model's output is then compared against the ground truth and the overall error rate is reported. By contrast, in online learning, the principle is to have the model learn concurrently with evaluation.

A major challenge of online learning is the inherent forgetfulness of the ANN training algorithm. A naive approach to online learning is to train it on just the online dataset, continuously passing only new samples into the model. This causes the model to forget trends it had learned from the pre-training set and to instead over-emphasize the new trends. To mitigate this effect, we propose a custom online learning technique (see

Figure 8.6).

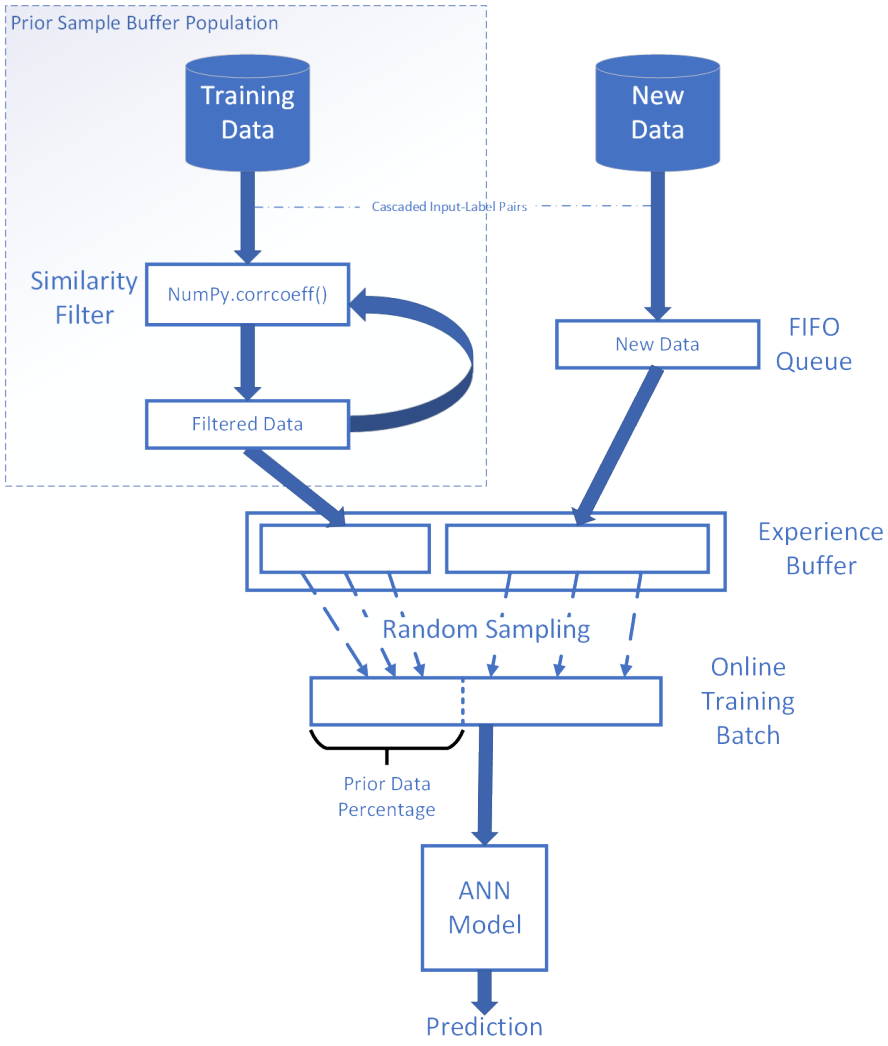


Figure 8.6: An illustration of how an online training batch is constructed using the experience buffer.

In order to preserve old trends while also learning new trends, it is necessary to interleave old data with new data when training. Moreover, due to space constraints, not all data from the pre-training dataset or the online learning dataset can be kept in memory. Thus, inspired by the concept of experience replay found in reinforcement learning, an experience replay buffer is used in the online learning procedure [35].

Ideally, the experience buffer is populated with a subset of the pre-training and online

data that is representative of the entire dataset. This dataset, however, is typically very large posing memory constraints on large buffer sizes. To this end, we require a measurement of similarity to filter out data containing the same information. For our technique, we calculate the Pearson product-moment correlation coefficients, as described by the NumPy *corrcoeff* function [6], between samples. Since this operation is slow, it is not performed on online data, as it would be impractical in a real-time setting.

When constructing an online training input batch, samples are randomly selected from the experience buffer such that a certain percentage of the batch contains old training data. Tuning this percentage allows the online training steps to emphasize retaining old training trends or newer data trends.

Other important parameters for online learning are the learning rate and training step frequency. If these values are set too high, the model will change too drastically. This will cause major instabilities in the learning–possibly to the point of overriding everything the model had learned. On the other hand, if these values are not high enough, there may be negligible improvements obtained by the model. Therefore, these parameters must be tuned to achieve the correct balance for the problem at hand.

Chapter 9

Machine Learning Experimental Analysis

In our experiments, we first evaluate the effectiveness of the various ANN models to try to minimize RMSE (i.e., prediction error). These are used to obtain a pre-trained baseline model for online learning. The Bitbrains dataset was split into training, validation and test sets using a 60:20:20 ratio (750:250:250 VMs). For the following sections, training was performed to predict CPU usage values using a mean squared error loss during training.

9.1 Baseline Model Selection

In this section, we highlight our ANN model performances. The errors shown in Table 9.1 were averaged over 5 runs. Although the standard ANN performs relatively well, the LSTM consistently performs slightly better. Similarly, the LN-LSTM model performs

slightly better than the simple LSTM model. This is expected because the layer normalization only speeds up learning. This also implies that both have mostly converged. The multi-layer LN-LSTM shows that increasing the models capacity does not improve error prediction for this particular problem. When adding the convolutional layers to the LN-LSTM model, however, reduces prediction error by almost 0.1%.

Table 9.1: Overall RMSE for various ANN models.

ANN Model	RMSE (%)
Standard ANN	4.1264
LSTM	4.0696
LN-LSTM	4.0118
Multi-layer LN-LSTM	4.0163
Conv + LN-LSTMs	**3.9302**

9.2 Online Learning Parameter Analysis

In this section, the parameter space of our online learning technique is explored. The four parameters of interest are update rate, learning rate, prior sample percentage, and similarity threshold. Rather than displaying the entire search space, we highlight certain results to illustrate the trends of each of the parameters. These experiments were run using the convolutional LN-LSTM model as the baseline for each validation/test file (i.e., VM application).

In Figure 9.1, the update rate is displayed in terms of the number of iterations before every training step. It is seen that when the model is updated too frequently, the prediction error increases due to over-training. In our experiments, the prediction error is 3.7955% with an update rate of every 50 iterations. When updates are not frequent

enough, however, the error also rises back to the original baseline prediction error. In theory the prediction error would return to the original error if the model never updated. Empirically, even with an update rate of every 250 iterations, prediction error is seen to rise to 3.7976%. Therefore, a balance must be found for update frequency. In our experiments, this was found to be every 100 iterations achieving a prediction error of 3.7769%. Although this difference seems negligible, it can manifest itself into significant cost savings for large data centres when dealing with hundreds of thousands of VMs, as opposed to 250 VMs.

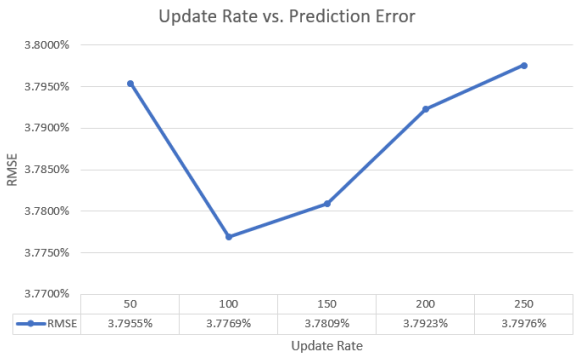


Figure 9.1: Graph of RMSEs with varying update rates. Other parameters were held constant (learning rate: $2e-6$; prior sample percentage: 10%; similarity threshold: 0.5).

Analogous to the update rate, learning rate follows a similar trend. If the learning rate is set too low, it would be equivalent to not training the model at all. If the learning rate is set too high, large instabilities in training are seen. This is, however, greatly accentuated for learning rate in comparison to update rate, as seen in Figure 9.2. Prediction error is seen to not only worsen, but perform worse than the original baseline error. In our experiments, the optimal learning rate was found to be $2e-6$ (500^{th} of the original 0.001 learning rate) with a prediction error of 3.7781%.

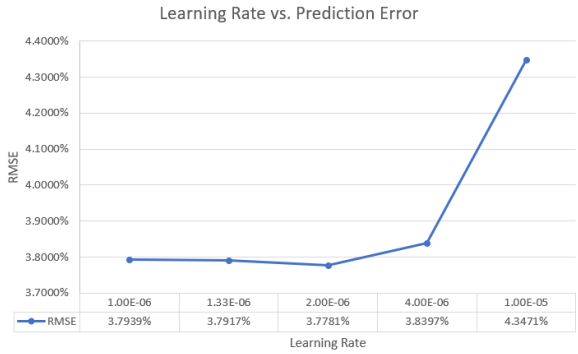


Figure 9.2: Graph of RMSEs with varying learning rates. Other parameters were held constant (update rate: every 100 iterations; prior sample percentage: 10%; similarity threshold: 0.55).

In Figure 9.3, the percentage of prior data in each batch is varied. For our experiments, the optimal percentage was found to be approximately 10-15% with prediction errors of 3.7769% and 3.7762%, respectively. When employing more prior data, errors were seen to rise implying that the model preferred new data over old data.

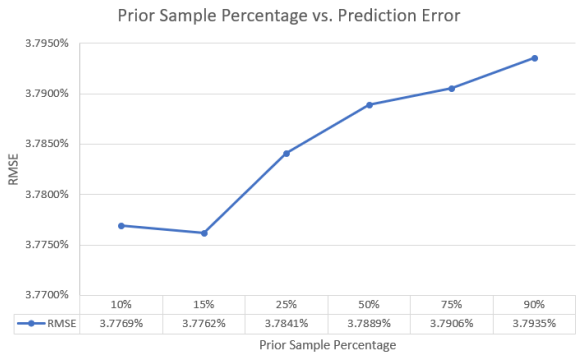


Figure 9.3: Graph of RMSEs with varying prior sample percentages. Other parameters were held constant (update rate: every 100 iterations; learning rate: $2e-6$; similarity threshold: 0.5).

The similarity threshold was also varied, as seen in Figure 9.4. The threshold value shown is the absolute minimum Pearson similarity coefficient that a sample needs to be included in the experience buffer. This dictates the minimum amount of dissimilarity that

samples should be from each other, where higher dissimilarity guarantees more breadth and variety in the prior data. In our experiments, a 0.7 similarity threshold was found to be optimal, achieving a prediction error of 3.7694%. For different workloads, this result may differ slightly, due to the characteristics of the workload. In this scenario, the workload consisted of a good mixture of new and old data.

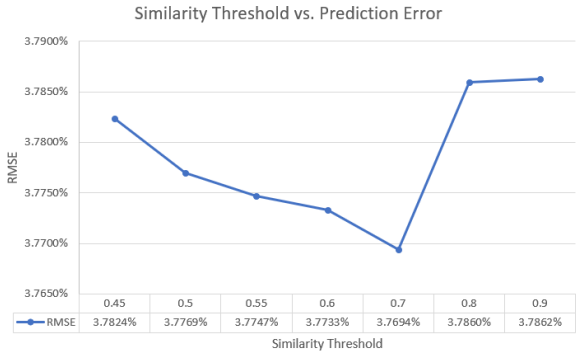


Figure 9.4: Graph of RMSEs with varying similarity thresholds. Other parameters were held constant (update rate: every 100 iterations; learning rate: $2e-6$; prior sample percentage: 10%).

Overall, online learning outperformed its baseline in almost every case. It was also found to be very sensitive to workload characteristics, and thus must be tuned with care. Our most optimal tuning parameters for the Bitbrains dataset uses an update rate of every 100 iterations, a learning rate of $2e-6$, a prior sample percentage of 10%, and a similarity threshold of 0.7. This achieved an impressive RMSE of 3.7694%, which further improved the baseline by another 0.27%.

Online learning has proven to be an invaluable tool in detecting time-series aberrations. This can be attributed to the online learning's ability to analyze the latest incoming trends of data, as well as, the multi-layered convolutional LN-LSTM's ability to maintain accurate memory of previous long-term data trends.

Chapter 10

Conclusions

The microservices architecture model lends itself well to the continuous delivery of robust and maintainable applications. To help cloud data centres keep up with increasing demand, innovative scaling techniques must be developed to support effective utilization of available resources and ensure service to clients. In this paper, we propose HyScale, two reactive hybrid autoscaling algorithms that combine horizontal and vertical scaling techniques to achieve higher resource efficiencies. Using our autoscaler platform, we demonstrate the availability and performance benefits of HyScale when benchmarked against Google's Kubernetes horizontal autoscaling algorithm. The higher SLA adherence and faster response times attained will allow cloud data centres to save substantially on power consumption costs and SLA violation penalties. Furthermore, HyScale will help data centres adhere to their maximum capacities as larger numbers of microservices can be packed more efficiently onto available hardware. As our results showed, $\text{HyScale}_{\text{CPU+Mem}}$ predominantly outperformed $\text{HyScale}_{\text{CPU}}$ by considering multiple metrics simultaneously. This performance improvement was counterbalanced by extra

overhead from more frequent container updates resulting in slightly more SLA violation costs. Tuning of scaling parameters can be used to mitigate these SLA violations. In future work, we aim to extend our hybrid autoscaling algorithms to incorporate a cost-based aspect, a network and disk aspect, and various others. We also aim to support features such as the dynamic addition and removal of machines, and stateful microservices.

Furthermore, as a step towards a predictive-reactive autoscaling solution, predictive approaches were also explored using ANNs. The ANN and RNN models lend themselves well to predictive solutions of time-series workloads. Various predictive ANN models were evaluated and an online learning technique to help achieve even lower prediction errors is proposed. Starting from the most basic ANN model, a prediction error of 4.12% was achieved. Using our convolutional LN-LSTM model, we demonstrate its performance benefits when benchmarked against the other basic models. While the 3.93% RMSE obtained with this model is compelling enough given the highly volatile nature of the Bitbrains workloads, when coupled with our custom online learning technique, the RMSE was further reduced to 3.77%. With this technique, cloud data centres will be able to better allocate resources to tenants resulting in less SLA violations. In future work, we plan to continue improving upon both our baseline ANN model and online learning technique. For our baseline model, there are several other parameters to investigate, such as adding pooling layers and fully connected layers to reduce the dimensionality of data between layers. Moreover, finding a dataset consisting of workloads lasting longer than a month would be ideal for realizing the gains from online learning. The next step is to implement our predictive online ANN solution onto HyScale to actualize savings from both the client and data centre perspectives. Ultimately, this approach will be combined

with our reactive hybrid solutions to formulate our predictive-reactive hybrid solution.

Bibliography

[1] GWA-T-12 Bitbrains. http://gwa.ewi.tudelft.nl/datasets/gwa-t-12-bitbrains. Accessed: 2017-10-30.

[2] How to avoid overprovisioning: Dont waste money on IaaS! https://community.spiceworks.com/cloud/article/overprovisioning-servers-iaas. Accessed: 2017-5-4.

[3] Jelastic. https://jelastic.com/. Accessed: 2017-8-10.

[4] Kubernetes. https://kubernetes.io/.

[5] Kubernetes support for multiple metrics. https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/#support-for-multiple-metrics. Accessed: 2017-11-13.

[6] numpy.corrcoef. https://docs.scipy.org/doc/numpy/reference/generated/numpy.corrcoef.html. Accessed: 2018-04-20.

[7] OpenStack & containers. https://www.openstack.org/containers/. Accessed: 2017-04-20.

[8] Swarm: a Docker-native clustering system. https://github.com/docker/swarm. Accessed: 2017-04-23.

[9] Limit a container's resources. https://docs.docker.com/engine/admin/resource_constraints/\#understand-the-risks-of-running-out-of-memory/, 2017.

[10] Y. Al-Dhuraibi, F. Paraiso, N. Djarallah, and P. Merle. Autonomic vertical elasticity of Docker containers with ELASTICDOCKER. In *2017 IEEE 10th International Conference on Cloud Computing (CLOUD)*, pages 472–479, June 2017.

[11] Jeremy Appleyard, Tomás Kociský, and Phil Blunsom. Optimizing performance of recurrent neural networks on GPUs. *CoRR*, abs/1604.01946, 2016.

[12] Doina Bein, Wolfgang Bein, and Swathi Venigella. Cloud storage and online bin packing. In *Intelligent Distributed Computing V*, pages 63–68. Springer, 2011.

[13] Peter Bodík, Rean Griffith, Charles Sutton, Armando Fox, Michael Jordan, and David Patterson. Statistical machine learning makes automatic control practical for internet datacenters. In *Proceedings of the 2009 Conference on Hot Topics in Cloud Computing*, HotCloud'09, Berkeley, CA, USA, 2009. USENIX Association.

[14] R. N. Calheiros, E. Masoumi, R. Ranjan, and R. Buyya. Workload prediction using ARIMA model and its impact on cloud applications; QoS. *IEEE Transactions on Cloud Computing*, 3(4):449–458, Oct 2015.

[15] Jeff Clark. Raising data center power density. http://www.datacenterjournal.com/raising-data-center-power-density/, Oct 2013.

[16] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. Resource Central: Understanding and predicting workloads for improved resource management in large cloud platforms. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 153–167, New York, NY, USA, 2017. ACM.

[17] Sourav Dutta, Sankalp Gera, Akshat Verma, and Balaji Viswanathan. Smartscale: Automatic application scaling in enterprise clouds. In *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*, pages 221–228. IEEE, 2012.

[18] Wei Fang, ZhiHui Lu, Jie Wu, and ZhenYin Cao. RPPS: a novel resource prediction and provisioning scheme in cloud data center. In *Services Computing (SCC), 2012 IEEE Ninth International Conference on*, pages 609–616. IEEE, 2012.

[19] Wes Felter, Alexandre Ferreira, Ram Rajamony, and Juan Rubio. An updated performance comparison of virtual machines and Linux containers. In *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium On*, pages 171–172. IEEE, 2015.

[20] Felix A. Gers, Douglas Eck, and Jürgen Schmidhuber. Applying LSTM to time series predictable through time-window approaches. In Roberto Tagliaferri and Maria Marinaro, editors, *Neural Nets WIRN Vietri-01*, pages 193–200, London, 2002. Springer London.

[21] C. Lee Giles, Steve Lawrence, and Ah Chung Tsoi. Noisy time series prediction using recurrent neural networks and grammatical inference. *Machine Learning*, 44(1):161–

183, Jul 2001.

[22] R. Han, L. Guo, M. M. Ghanem, and Y. Guo. Lightweight resource scaling for cloud applications. In *2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*, pages 644–651, May 2012.

[23] Rui Han, Moustafa M Ghanem, Li Guo, Yike Guo, and Michelle Osmond. Enabling cost-aware and adaptive elasticity of multi-tier cloud applications. *Future Generation Computer Systems*, 32:82–98, 2014.

[24] Philipp Hoenisch, Ingo Weber, Stefan Schulte, Liming Zhu, and Alan Fekete. Four-fold auto-scaling on a contemporary deployment platform using Docker containers. In Alistair Barros, Daniela Grigori, Nanjangud C. Narendra, and Hoa Khanh Dam, editors, *Service-Oriented Computing: 13th International Conference, ICSOC 2015, Goa, India, November 16-19, 2015, Proceedings*, pages 316–323, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[25] Nikolaus Huber, Fabian Brosig, and Samuel Kounev. Model-based self-adaptive resource allocation in virtualized environments. In *Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pages 90–99. ACM, 2011.

[26] A. Inomata, T. Morikawa, M. Ikebe, Y. Okamoto, S. Noguchi, K. Fujikawa, H. Sunahara, and S. M. M. Rahman. Proposal and evaluation of a dynamic resource allocation method based on the load of vms on iaas. In *2011 4th IFIP International Conference on New Technologies, Mobility and Security*, pages 1–6, Feb 2011.

[27] P. Jones. Waging war on overprovisioning. `http://www.datacenterdynamics.com/content-tracks/servers-storage/waging-war-on-overprovisioning/80774.fullarticle`, 2013.

[28] C. Kan. DoCloud: An elastic cloud platform for web applications based on Docker. In *2016 18th International Conference on Advanced Communication Technology (ICACT)*, pages 478–483, Jan 2016.

[29] Richard E Korf. A new algorithm for optimal bin packing. In *Eighteenth National Conference on Artificial Intelligence*, pages 731–736, Menlo Park, CA, USA, 2002. American Association for Artificial Intelligence.

[30] Jitendra Kumar and Ashutosh Kumar Singh. Workload prediction in cloud using artificial neural network and adaptive differential evolution. *Future Generation Computer Systems*, 81:41 – 52, 2018.

[31] J. Lei Ba, J. R. Kiros, and G. E. Hinton. Layer Normalization. *ArXiv e-prints*, July 2016.

[32] Yusen Li, Xueyan Tang, and Wentong Cai. On dynamic bin packing for resource allocation in the cloud. In *Proceedings of the 26th ACM symposium on Parallelism in algorithms and architectures*, pages 2–11. ACM, 2014.

[33] Jeff Lindsay. Progrium stress. `https://github.com/progrium/docker-stress`, 2014.

[34] Lei Lu, Xiaoyun Zhu, Rean Griffith, Pradeep Padala, Aashish Parikh, Parth Shah, and Evgenia Smirni. Application-driven dynamic vertical scaling of virtual machines

in resource pools. In *2014 IEEE Network Operations and Management Symposium, NOMS 2014, Krakow, Poland, May 5-9, 2014*, pages 1–9, 2014.

[35] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. Playing Atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013.

[36] Germán Moltó, Miguel Caballer, and Carlos de Alfonso. Automatic memory-based vertical elasticity and oversubscription on cloud platforms. volume 56, pages 1–10, Amsterdam, The Netherlands, The Netherlands, March 2016. Elsevier Science Publishers B. V.

[37] C. Olah. Understanding LSTM networks. `http://colah.github.io/posts/2015-08-Understanding-LSTMs/`, August 2015.

[38] F. Qiu, B. Zhang, and J. Guo. A deep learning approach for VM workload prediction in the cloud. In *2016 17th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)*, pages 319–324, May 2016.

[39] W. Qu, Y. He, and W. Qu. Research on forecasting approach for complex time series based on support vector machines. In *2010 2nd International Conference on Information Engineering and Computer Science*, pages 1–4, Dec 2010.

[40] Karthick Rajamani, Wes Felter, Alexandre Ferreira, and Juan Rubio. Spyre: A resource management framework for container-based clouds, 2015.

[41] Tiago Rosado and Jorge Bernardino. An overview of OpenStack architecture. In *Proceedings of the 18th International Database Engineering & Applications Symposium*, IDEAS '14, pages 366–367, New York, NY, USA, 2014. ACM.

[42] T. N. Sainath, O. Vinyals, A. Senior, and H. Sak. Convolutional, long short-term memory, fully connected deep neural networks. In *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 4580–4584, April 2015.

[43] R. Sakhuja. 5 reasons why microservices have become so popular in the last 2 years. `https://www.linkedin.com/pulse/5-reasons-why-microservices-have-become-so-popular-last-sakhuja/`, March 2016.

[44] F. Shaikh. Why are GPUs necessary for training deep learning models? `https://www.analyticsvidhya.com/blog/2017/05/gpus-necessary-for-deep-learning/`, May 2017.

[45] Patrice Y. Simard, Dave Steinkraus, and John C. Platt. Best practices for convolutional neural networks applied to visual document analysis. In *Proceedings of the Seventh International Conference on Document Analysis and Recognition - Volume 2*, ICDAR '03, pages 958–, Washington, DC, USA, 2003. IEEE Computer Society.

[46] Kay Singh and Ralph Squillace. Vertically scale Azure linux virtual machine with Azure automation. `https://docs.microsoft.com/en-us/azure/virtual-machines/linux/vertical-scaling-automation`, Mar 2016.

[47] Binbin Song, Yao Yu, Yu Zhou, Ziqiang Wang, and Sidan Du. Host load prediction with long short-term memory in cloud computing. *The Journal of Supercomputing*, Apr 2017.

[48] N. Thomas and A Selvon-Bruce. How auto-scaling techniques make public cloud deployments more cost-effective. `https://cognizantsnapshot.com.au/auto-scaling-public-cloud-deployments/`, 2016.

[49] Qazi Zia Ullah, Hassan Shahzad, and Gul Muhammad Khan. Adaptive resource utilization prediction system for infrastructure as a service cloud. In *Comp. Int. and Neurosc.*, 2017.

[50] Bhuvan Urgaonkar, Prashant Shenoy, Abhishek Chandra, Pawan Goyal, and Timothy Wood. Agile dynamic provisioning of multi-tier internet applications. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 3(1):1, 2008.

[51] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. Apache Hadoop YARN: Yet Another Resource Negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 5:1–5:16, New York, NY, USA, 2013. ACM.

[52] Bill Wilder. *Cloud architecture patterns: using Microsoft Azure.* "O'Reilly Media, Inc.", 2012.

[53] Jingqi Yang, Chuanchang Liu, Yanlei Shang, Bo Cheng, Zexiang Mao, Chunhong Liu, Lisha Niu, and Junliang Chen. A cost-aware auto-scaling approach using the workload prediction in service clouds. *Information Systems Frontiers*, 16(1):7–18, 2014.

[54] Lean Yu, Shouyang Wang, and Kin Keung Lai. An online learning algorithm with adaptive forgetting factors for feedforward neural networks in financial time series forecasting. *Nonlinear dynamics and systems theory*, 7(1):97–112, 2007.

[55] W. Zhang, B. Li, D. Zhao, F. Gong, and Q. Lu. Workload prediction for cloud cluster using a recurrent neural network. In *2016 International Conference on Identification, Information and Knowledge in the Internet of Things (IIKI)*, pages 104–109, Oct 2016.