HyScale: Hybrid and Network Scaling of Dockerized Microservices in Cloud Data Centres

Jonathon Wong*, Anthony Kwan*, and Hans-Arno Jacobsen
University of Toronto, Toronto, ON
Emails: {jonathon.wong, anthony.kwan}@ece.toronto.edu,
jacobsen@eecg.toronto.edu

Vinod Muthusamy
Thomas J. Watson Research Center
Yorktown Heights, NY, USA
Email: vmuthus@us.ibm.com

***Abstract*—When designing modern software, care must be taken to allow for applications to scale based on the demands of its users while still accommodating flexibility in development. 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 underutilized during off-peak hours. Traditional microservice scaling methods perform either horizontal or vertical scaling exclusively. When used in combination, however, these methods 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, we developed two novel hybrid autoscaling algorithms and a dedicated network scaling algorithm and benchmarked them against Google's popular Kubernetes horizontal autoscaling algorithm. Results indicated up to 1.49x speedups in response times for our hybrid algorithms, and 1.69x speedups for our network algorithm under high-burst network loads.**

***Index Terms*—Docker, microservices, autoscaling, cloud.**

I. Introduction

Microservices architectures have gained widespread popularity in the software development community, quickly becoming a best practice for creating enterprise applications [1]. Under the microservices architecture model, a traditional monolithic application is dissociated into several smaller, self-contained component services or functions [2]. 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 [3]. 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 [4]. Conversely, data centres can suffer from resource underutilization. During off-peak hours, tenants are typically overprovisioned resources [4]. 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 [5], [6].

Traditional methods for scaling can generally be categorized into vertical or horizontal scaling with the more popular approach being horizontal scaling [7]–[10]. 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 [11]. 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

*Both authors contributed equally

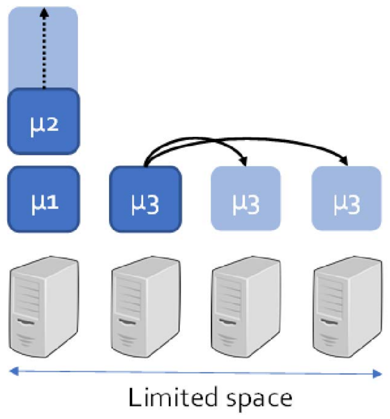


Fig. 1. Illustrates a hybrid autoscaling scenario. Container 1 and 2 (left) reside on the same machine. Container 2 is vertically scaled while Container 3 is horizontally scaled.

CPU) [12]–[15]. 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 [9]. 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 [13], [15], [16]. 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 [4]. For the most part, frameworks such as Kubernetes have simple autoscaling algorithms that frequently lead to non-optimal resource allocations.

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., CPU, memory, and network), finding the optimal configuration is an NP-complete problem [17]. These resource 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. Furthermore, resources intrinsically tied to other resources greatly obscure this relationship. Network I/O is one metric that is particularly complicated to scale in virtualized environments. Due to its dependence on CPU and socket resources, and independent ingress and egress filters, network bandwidth scaling is very convoluted and does not form the basis for many scaling algorithms.

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

1) Performance analysis of horizontal versus vertical scaling of Docker containers to quantitatively assess their trade-offs (Section III).
2) Design and implementation of two novel hybrid scaling techniques, HYSCALE$_{\text{CPU}}$ and HYSCALE$_{\text{CPU+Mem}}$, as well as a simple horizontal network scaling algorithm (Section IV).
3) Design and implementation of an autoscaling platform prototype to evaluate and compare various scaling techniques (Section V).
4) Validation of the performance and resource efficiency benefits gained through the use of hybrid and network scaling by benchmarking HYSCALE against Google's Kubernetes horizontal autoscaling algorithm on microbenchmarks and Bitbrain's VM workload (Section VI).

II. Related Work

Although container-based virtualization technology is relatively new, virtual machines (VMs) have been well researched and widely used for several decades [8]. Dynamic scaling of hardware resources has been approached from the VM perspective and is not a foreign concept in the world of virtualization [8], [12], [18], [19].

A. 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 [12] supports vertical scaling of VMs based on observed resource utilization. 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 [13] is a platform that provides mechanisms for memory oversubscription of VMs. 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 [15]. 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 [18], [19].

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 [16]. 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 [14]. 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.

B. 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 [8] provides users with the ability to automate horizontal scaling of VMs via Heat templates. 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 [20].

For container-based horizontal autoscaling, the most popular tools are Docker Swarm and Google's Kubernetes. Docker Swarm [7] takes a cluster of Docker-enabled machines and manages them as if they were a single Docker host. 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 [9] 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. It also attempts to provide a beta API for autoscaling based on multiple observed metrics [21]. 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. Kubernetes has also added support for a vertical autoscaler, however, at the time of publication, it was still a conceptual beta with few details on implementation [22].

C. Hybrid Scalers

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

SmartScale [23] 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. 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. [18], [24], [25], an extension of R. Han et al.'s VM work is presented by including a horizontal scaling aspect to the algorithm. 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 [26] also attempts to perform hybrid scaling of VMs by first vertically scaling where possible, then allocating new VM instances when required. 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 [27] presents a hybrid autoscaling technique to reduce costs for containers within VMs. 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 [28] monitors CPU, memory, network and disk usages to automatically trigger either vertical or horizontal scaling for a single application. 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 higher throughput input events and the lower latency nature of container ecosystems.

D. Network Scalers

NBWGuard [29] presents the first network bandwidth QoS solution for Kubernetes pods using tc commands. Egress traffic is drained by hierarchical token bucket filters allowing for varying network priorities amongst pods. Ingress traffic is redirected to a virtual interface using a kernel intermediate functional block, before applying tc. Their approach was then validated using iperf.

Most other solutions aim to address the issue of network performance and scalability through machine placement. X. Meng et al. [30] attempt to alleviate network traffic congestion by using a network-focused heuristic placement algorithm. This algorithm optimizes placement of VMs by shortening the distance between communicating VMs. Other algorithms present various placement algorithms that optimize communication traffic of all VMs to designated nodes [31], [32].

Work for network scaling and QoS of containers is sparse due to its intricate dependencies on various other resources. Network scaling solutions generally have been tailored to VMs. Due to high scaling overheads, physical placement becomes critical. Contrarily, containers have negligible scaling overhead and are lightweight enough to be replicated very quickly. During bursty traffic, these attributes can be exploited to reduce network congestion through the use of vertical and horizontal scaling.

III. Horizontal and 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 640 client requests on up to 16 machines:

A. CPU Scaling

A container's Docker CPU shares define its proportion of CPU cycles on a single machine [33]. 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 consume CPU time per request. The latency of a request is measured as the microservice execution time. This simulates CPU load on the system from the request/response mechanic that is inherent in a microservices architecture.

To create contention of CPU resources, the microservice is run alongside another container. This container runs progrium stress [34], which consumes CPU resources. 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. The equivalent horizontally-scaled instance 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 total CPU access time for each microservice, adding up to a total of

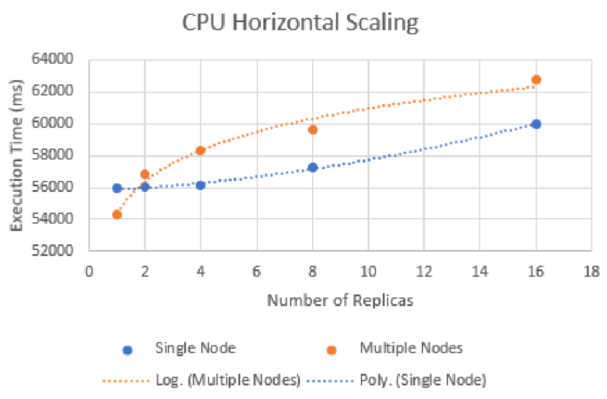


Fig. 2. Response times of horizontal scaling for the CPU tests.

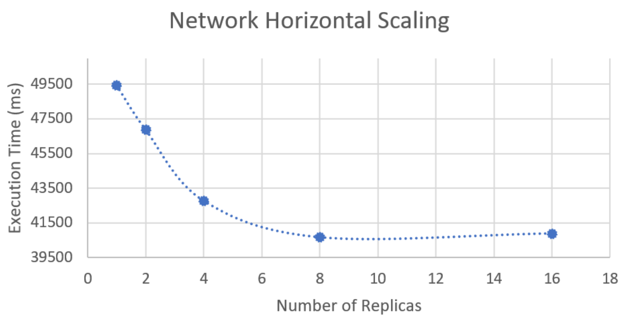


Fig. 3. Response times of horizontal scaling for the network tests with a total bandwidth of 100Mbps.

1/2 of the CPU access time. This is equivalent to the vertical scaled scenario.

Results of our studies on vertical versus horizontal scaling for CPU resources show a preference for vertical scaling. It provided negligible overhead in request processing times when compared to the equivalently horizontally scaled instances. Results also strongly indicate that more replicated instances decrease overall CPU performance (see Figure 2). Although Docker containers, themselves, have negligible overhead [35], 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. Significant overhead is also seen when replicas are distributed across several nodes resulting in a logarithmic increase with the number of replicas. Note that in practice, replicating a container in a cloud environment will inevitably force containers to co-locate with other containers further decreasing overall performance of every microservice.

B. *Memory Scaling*

Docker also allows users to impose memory limits on containers [33]. 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.

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.

C. *Network Scaling*

Although Docker supports container resizing for many resource types, network I/O is an exception. Limiting network bandwidth for containers in general must be done through third party tools, such as the traffic control (tc) command in conjunction with iptables routing. In this section, we explore the scaling of egress UDP and TCP traffic by using tc and iptables to scale network bandwidth in each scenario. Analogously to our CPU tests, a total bandwidth of 100Mbps was allocated to our microservice running iperf [36] and run alongside a custom stress container that attempts to hog all available CPU and network resources.

Results were averaged over 30 runs for each scenario and showed negligible changes for vertical scaling of network due to the effective and fair distribution of traffic using tc and iptables. For horizontal scaling, however, a large decrease in execution time is achieved at a larger number of replicas, tapering off at around 8 replicas (see Figure 3). This is mainly attributed to the alleviated contention over the network tx queues when using more machines. In general, horizontally scaling outwards provides significant benefits for network scaling for our cluster. Although results for only 100Mbps are shown, there are several other factors that affect actual performance, such as network speeds, number of requests, and size of requests. From varying these parameters, we found the results followed the same general trends.

IV. 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.

A. Horizontal Scaling Algorithms

1) Kubernetes 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. 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 replica *r* using the formula:

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

$$NumReplicas_m = \lceil \frac{sum(utilization_r)}{Target_m} \rceil$$

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 rescaling if the following holds true:

$$|\frac{average(usage_r)}{Target_m} - 1| > 0.1$$

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.

2) Network Scaling Algorithm: There are several aspects of networking that contribute to the complexity of its scaling, such as number of sockets, protocols, shared network interface cards and switches, kernel scheduling, and filtering. As a result, there is no known generic implementation of network bandwidth scaling nor is it natively supported in Kubernetes. Therefore, we chose to design an exploratory horizontal algorithm based on the results of our network scaling experiments. This algorithm uses the same algorithm as Kubernetes, but replaces CPU usage for outgoing network bandwidth usage in its calculations.

B. 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.

1) HYSCALE$_{CPU}$ *Algorithm:* This hybrid algorithm considers only CPU usage and calculates the number of missing CPUs for microservice *m* using the equation:

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

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:

$$ReclaimableCPUs_r = requested_r - \frac{sum(usage_r)}{Target_m * 0.9}$$

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:

$$RequiredCPUs_r = \frac{sum(usage_r)}{Target_m * 0.9} - requested_r$$

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

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.

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 adjustments quickly and frequently.

2) HYSCALE$_{CPU+Mem}$ Algorithm: This hybrid algorithm extends from the previous algorithm by considering memory and swap usage. The algorithm and equations used are analogous to those used for CPU measurements and are shown below.

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

$$ReclaimableMem_r = requested_r - \frac{sum(usage_r)}{Target_m * 0.9}$$

$$RequiredMem_r = \frac{sum(usage_r)}{Target_m * 0.9} - requested_r$$

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

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.

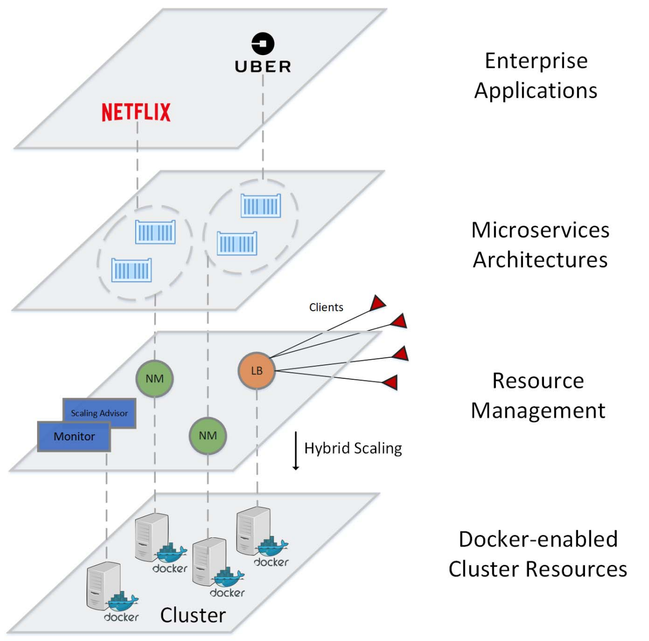


Fig. 4. Full stack overview showing the resource management layer (second bottom) above the cloud resources (bottom). Docker microservices (second top) give rise to enterprise applications (top) that clients interact with.

V. AUTOSCALER 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 [37], 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 4 and 5. Further details on each component are covered in following sections.

A. 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 Kubernetes pods) and each microservice component is housed within its own container. In our architecture, each Docker container houses a single microservice and is the unit of deployment for all microservices, including differing instances and replications. 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.

B. Node Managers

Each node runs a single NM (see Figure 5), 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 '*docker stats*'.

Additionally, the NM receives vertical scaling resource commands from the MONITOR for specific containers. NMs perform these adjustments by invoking '*docker update*'. 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.

C. 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.

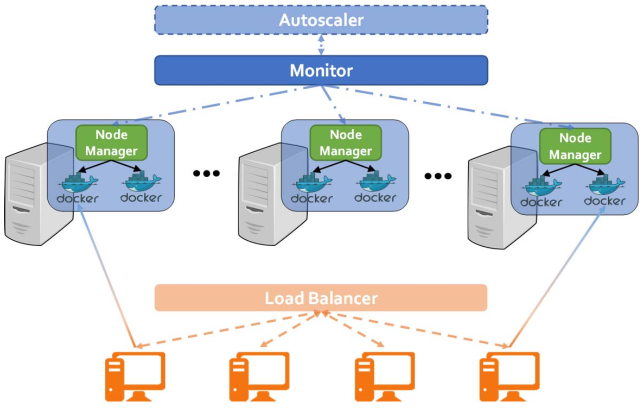


Fig. 5. Architecture overview illustrating the various interactions between each component.

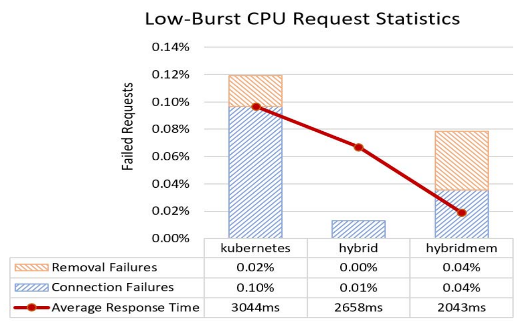
VI. EXPERIMENTAL ANALYSIS

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

We chose to evaluate each algorithm under various types of loads in the form of microbenchmarks 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 low amplitude bursty traffic, labelled *low-burst*, and the unstable load forms a spiking pattern, labelled *high-burst*. This wave-like bursty pattern simulates repeated peaks and troughs in client activity.

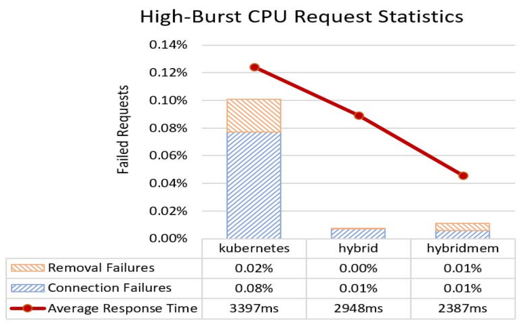
We also present the system with 4 different types of microservices: CPU-bound, memory-bound, network-bound, mixed CPU and memory. Microservices applications' workloads are emulated using a custom Java microservice with configurable workload. Upon instantiation, our emulated microservices are specified an amount of resources to consume per incoming client request. Based on these inputs, we can create microservices that vary in resource consumption. Additional computing resource types, such as disk I/O, are also supported, however, they are not currently implemented and will be part of future works.

Each experiment was performed using 15 different microservices for an hour on a cluster of 24 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 following sections. In the



	kubernetes	hybrid	hybridmem
▨ Removal Failures	0.02%	0.00%	0.04%
▨ Connection Failures	0.10%	0.01%	0.04%
● Average Response Time	3044ms	2658ms	2043ms

(a)



	kubernetes	hybrid	hybridmem
▨ Removal Failures	0.02%	0.00%	0.01%
▨ Connection Failures	0.08%	0.01%	0.01%
● Average Response Time	3397ms	2948ms	2387ms

(b)

Fig. 6. 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.

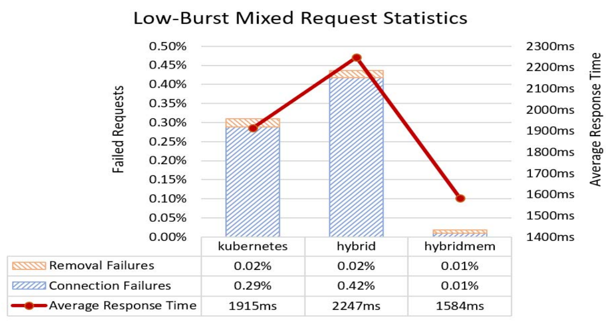
following figures and tables, 'hybrid' refers to HYSCALE$_{\text{CPU}}$ and 'hybridmem' refers to HYSCALE$_{\text{CPU+Mem}}$.

A. 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 6 and 7 show the average response times and percentage of failed requests of each algorithm.

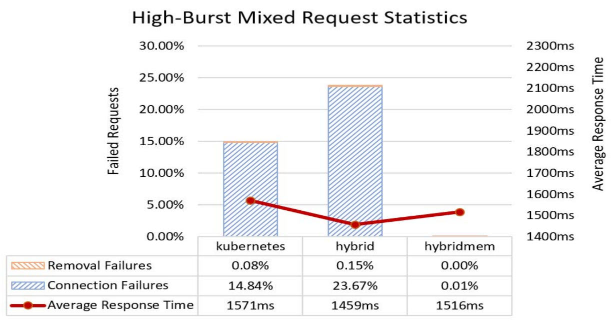
In the CPU-bound experiments (Figures 6a and 6b), HYSCALE$_{\text{CPU+Mem}}$ has the fastest response times overall, while Kubernetes has the slowest response times. There are clear improvements in response times of HYSCALE as compared to Kubernetes resulting in 1.49x and 1.43x speedups for the low and high-burst workloads, respectively. 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 7a and 7b), Kubernetes and HYSCALE$_{\text{CPU}}$ 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



	kubernetes	hybrid	hybridmem
▨ Removal Failures	0.02%	0.02%	0.01%
▨ Connection Failures	0.29%	0.42%	0.01%
● Average Response Time	1915ms	2247ms	1584ms

(a)



	kubernetes	hybrid	hybridmem
▨ Removal Failures	0.08%	0.15%	0.00%
▨ Connection Failures	14.84%	23.67%	0.01%
● Average Response Time	1571ms	1459ms	1516ms

(b)

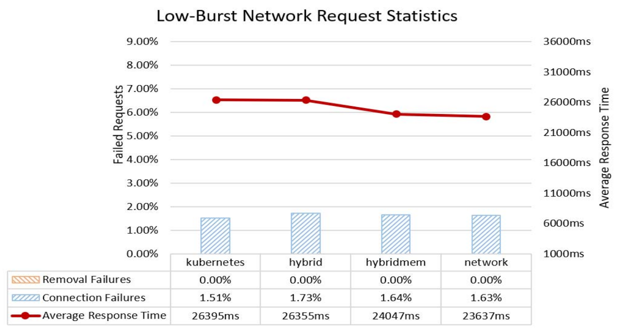
Fig. 7. Request statistics graphs for mixed resource experiments. Note the significant difference in failed requests between 7a and 7b.

is made in Figure 7a, where Kubernetes appears to perform better than HYSCALE$_{\text{CPU}}$. This is caused by HYSCALE$_{\text{CPU}}$'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 7b, the response times are significantly skewed for Kubernetes and HYSCALE$_{\text{CPU}}$, since the microservices are effectively handling a smaller portion of requests (up to 23.67% less requests).

In the network-bound experiments (Figures 8a and 8b), our network scaling algorithm outperformed the others overall with Kubernetes being the slowest. Despite the other algorithms' lack of consideration for actual network usage, they still manage to stay competitive under low-burst stable workloads, due to the moderate use of CPU caused by networking system calls. This, however, does not hold for unstable workloads, where dedicated network scaling shows a clear advantage with response times dropping by up to 59.22%. The results for Kubernetes and HYSCALE stay consistent with previous experiments (Figures 7a and 7b), further corroborating our HYSCALE algorithms' ability to achieve better performances under CPU loads.

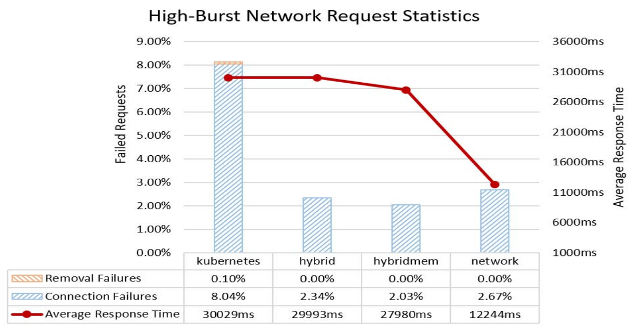
B. Bitbrains Workload

To emulate the stress that a microservices architecture would undergo in a realistic cloud data centre, we benchmarked the Kubernetes and HYSCALE algorithms using the Rnd dataset



	kubernetes	hybrid	hybridmem	network
▨ Removal Failures	0.00%	0.00%	0.00%	0.00%
▨ Connection Failures	1.51%	1.73%	1.64%	1.63%
● Average Response Time	26395ms	26355ms	24047ms	23637ms

(a)



	kubernetes	hybrid	hybridmem	network
▨ Removal Failures	0.10%	0.00%	0.00%	0.00%
▨ Connection Failures	8.04%	2.34%	2.03%	2.67%
● Average Response Time	30029ms	29993ms	27980ms	12244ms

(b)

Fig. 8. Graphs depicting the percentage of requests failed and the average request response times for the network-bound experiments.

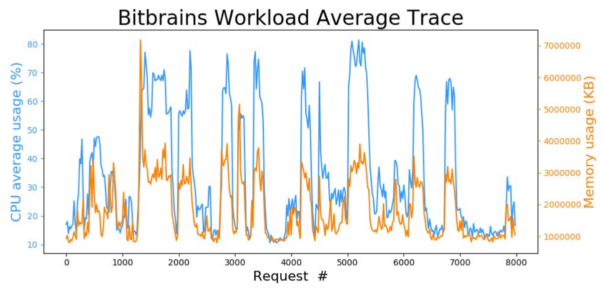
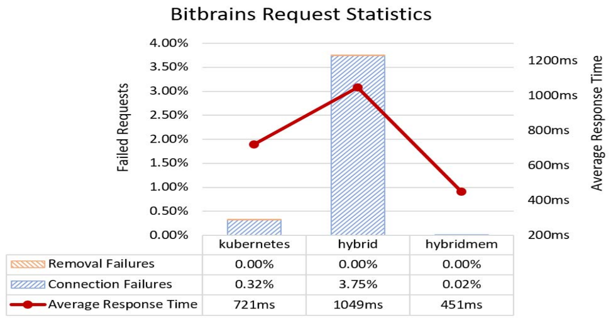


Fig. 9. Graph of the Bitbrains Rnd workload trace for CPU and memory usage averaged over all microservices.

from the GWA-T-12 Bitbrains workload trace [38]. 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 9) exhibits the same behaviour as the *low-burst mix* and *high-burst mix* workloads, and thus is expected to manifest the same result trends.

The performance results (see Figure 10) were similar to the mixed experiment results (see Figure 7). HYSCALE$_{\text{CPU+Mem}}$ performs the best because of its ability to scale both



	kubernetes	hybrid	hybridmem
▨ Removal Failures	0.00%	0.00%	0.00%
▨ Connection Failures	0.32%	3.75%	0.02%
● Average Response Time	721ms	1049ms	451ms

Fig. 10. Request statistics graph for the Bitbrains experiment.

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.

VII. 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 a dedicated network scaling algorithm and HYSCALE, two 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 each 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, our algorithms 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, HYSCALE$_{\text{CPU+Mem}}$ predominantly outperformed HYSCALE$_{\text{CPU}}$ and Kubernetes by considering multiple metrics simultaneously. Due to the complexity and obscurity of the network resource metrics, network scaling was considered separately and a preliminary horizontal scaling algorithm was presented. This was able to achieve significant performance benefits and shows huge potential. In future works, we aim to further explore network resource scaling and extend our hybrid autoscaling algorithms to incorporate a cost-based aspect, a machine learning aspect and various others. We also aim to support features such as the dynamic addition and removal of machines, and stateful microservices. We also intend to benchmark our solution against new and upcoming algorithms such as Kubernetes when it fully supports vertical scaling.

VIII. Acknowledgements

This research has in part been funded by NSERC and an IBM Canada CAS fellowship. We would also like to express our sincere gratitude to several individuals from IBM Research and IBM Canada who have supported this work with input and advice including Cong Xu, Karthick Rajamani, Allen Chan, and Suzette Samoojh. Finally, we would like to thank Geoffrey Elliott for managing the cluster for our experiments.

References

[1] M. Fowler and J. Lewis, "Microservices," https://martinfowler.com/articles/microservices.html, March 2014.

[2] "OpenFaaS - serverless functions made simple," https://docs.openfaas.com, accessed: 2018-12-16.

[3] A. Greenberg, J. Hamilton, D. A. Maltz, and P. Patel, "The cost of a cloud: Research problems in data center networks," http://doi.acm.org/10.1145/1496091.1496103, New York, NY, USA, pp. 68–73, Dec. 2008.

[4] P. Marshall, K. Keahey, and T. Freeman, "Improving utilization of infrastructure clouds," http://dx.doi.org/10.1109/CCGrid.2011.56, Washington, DC, USA, pp. 205–214, 2011.

[5] Q. Zhang, L. Cheng, and R. Boutaba, "Cloud computing: state-of-the-art and research challenges," *Journal of Internet Services and Applications*, vol. 1, no. 1, pp. 7–18, May 2010.

[6] 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.

[7] "Swarm: a Docker-native clustering system," https://github.com/docker/swarm, accessed: 2017-04-23.

[8] T. Rosado and J. Bernardino, "An overview of OpenStack architecture," in *Proceedings of the 18th International Database Engineering & Applications Symposium*, ser. IDEAS '14. New York, NY, USA: ACM, 2014, pp. 366–367. [Online]. Available: http://doi.acm.org/10.1145/2628194.2628195

[9] "Kubernetes," https://kubernetes.io/.

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

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

[12] K. Singh and R. 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.

[13] G. Moltó, M. Caballer, and C. de Alfonso, "Automatic memory-based vertical elasticity and oversubscription on cloud platforms," Amsterdam, The Netherlands, The Netherlands, pp. 1–10, Mar. 2016. [Online]. Available: https://doi.org/10.1016/j.future.2015.10.002

[14] K. Rajamani, W. Felter, A. Ferreira, and J. Rubio, "Spyre: A resource management framework for container-based clouds," 2015.

[15] L. Lu, X. Zhu, R. Griffith, P. Padala, A. Parikh, P. Shah, and E. 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*, 2014, pp. 1–9. [Online]. Available: http://dx.doi.org/10.1109/NOMS.2014.6838238

[16] 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)*, June 2017, pp. 472–479.

[17] R. E. Korf, "A new algorithm for optimal bin packing," in *Eighteenth National Conference on Artificial Intelligence*. Menlo Park, CA, USA: American Association for Artificial Intelligence, 2002, pp. 731–736. [Online]. Available: http://dl.acm.org/citation.cfm?id=777092.777205

[18] 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)*, May 2012, pp. 644–651.

[19] 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*, Feb 2011, pp. 1–6.

[20] "OpenStack & containers," https://www.openstack.org/containers/, accessed: 2017-04-20.

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

[22] "Vertical pod autoscaling," https://cloud.google.com/kubernetes-engine/docs/concepts/verticalpodautoscaler.

[23] S. Dutta, S. Gera, A. Verma, and B. Viswanathan, "Smartscale: Automatic application scaling in enterprise clouds," in *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*. IEEE, 2012, pp. 221–228.

[24] J. Yang, C. Liu, Y. Shang, B. Cheng, Z. Mao, C. Liu, L. Niu, and J. Chen, "A cost-aware auto-scaling approach using the workload prediction in service clouds," *Information Systems Frontiers*, vol. 16, no. 1, pp. 7–18, 2014.

[25] R. Han, M. M. Ghanem, L. Guo, Y. Guo, and M. Osmond, "Enabling cost-aware and adaptive elasticity of multi-tier cloud applications," *Future Generation Computer Systems*, vol. 32, pp. 82–98, 2014.

[26] N. Huber, F. Brosig, and S. 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*. ACM, 2011, pp. 90–99.

[27] P. Hoenisch, I. Weber, S. Schulte, L. Zhu, and A. Fekete, "Four-fold auto-scaling on a contemporary deployment platform using Docker containers," in *Service-Oriented Computing: 13th International Conference, ICSOC 2015, Goa, India, November 16-19, 2015, Proceedings*, A. Barros, D. Grigori, N. C. Narendra, and H. K. Dam, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 316–323. [Online]. Available: https://doi.org/10.1007/978-3-662-48616-0_20

[28] "Jelastic," https://jelastic.com/, accessed: 2017-8-10.

[29] C. Xu, K. Rajamani, and W. Felter, "Nbwguard: Realizing network qos for kubernetes," http://doi.acm.org/10.1145/3284028.3284033, New York, NY, USA, pp. 32–38, 2018.

[30] X. Meng, V. Pappas, and L. Zhang, "Improving the scalability of data center networks with traffic-aware virtual machine placement," in *2010 Proceedings IEEE INFOCOM*, March 2010, pp. 1–9.

[31] R. Cohen, L. Lewin-Eytan, J. S. Naor, and D. Raz, "Almost optimal virtual machine placement for traffic intense data centers," in *2013 Proceedings IEEE INFOCOM*, April 2013, pp. 355–359.

[32] J. T. Piao and J. Yan, "A network-aware virtual machine placement and migration approach in cloud computing," in *2010 Ninth International Conference on Grid and Cloud Computing*, Nov 2010, pp. 87–92.

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

[34] J. Lindsay, "Progrium stress," https://github.com/progrium/docker-stress, 2014.

[35] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and Linux containers," in *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium On*. IEEE, 2015, pp. 171–172.

[36] A. Tirumala, F. Qin, J. Dugan, J. Ferguson, and K. Gibbs, "iperf: Tcp/udp bandwidth measurement tool," 01 2005.

[37] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler, "Apache Hadoop YARN: Yet Another Resource Negotiator," in *Proceedings of the 4th Annual Symposium on Cloud Computing*, ser. SOCC '13. New York, NY, USA: ACM, 2013, pp. 5:1–5:16. [Online]. Available: http://doi.acm.org/10.1145/2523616.2523633

[38] "GWA-T-12 Bitbrains," http://gwa.ewi.tudelft.nl/datasets/gwa-t-12-bitbrains, accessed: 2017-10-30.