2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)

Auto-scaling a Defence Application across the Cloud using Docker and Kubernetes

S. Kho Lin, U. Altaf, G. Jayaputera, J. Li,
D. Marques, D. Meggyesy, S. Sarwar, S. Sharma,
W. Voorsluys, R.O. Sinnott
School of Computing and Information Systems
University of Melbourne
Melbourne, Victoria, Australia
victorsankho.lin@unimelb.edu.au

A. Novak, V. Nguyen, K. Pash
Defence Science and Technology Group
Melbourne, Victoria, Australia

***Abstract*—The Australian Defence Forces (ADF) including the army, navy and air force often share resources. This is the case for helicopter training where the resources are often people, e.g. instructors. Understanding the current and future needs of the helicopter pilot training continuum is challenging. New students are continually entering the Defence services as trainees/cadets; passing or failing exams at specialist flying schools; progressing through or leaving the Defence services and potentially becoming trained pilots or instructors that can subsequently train future students. This continuum has an associated optimisation challenge. The ATHENA platform has been developed as a strategic discrete event simulation, optimisation and analysis system for manpower planning with specific focus on addressing the needs and demands of this helicopter pilot training continuum for the ADF. ATHENA has been developed as a simulation platform running on a Cloud infrastructure. This paper introduces ATHENA and describes the way in which the platform leverages container technologies to auto-scale across the Cloud with focus on the Kubernetes orchestration technology.**

***Keywords*-Auto-scaling, Container orchestration, Docker, Kubernetes, OpenStack**

I. Introduction

Cloud Computing [1] and more recently container-based approaches [2] have changed the way software packaging and deployment are now achieved. Containerised Applications [3], Infrastructure-as-Code (IaC) and associated systems administration and DevOps [4] underpin modern deployment practices. Advances in virtualization technologies have changed the Operating System landscape and it is now commonly required to support complex software systems/stacks for many enterprises. Several studies [5] [6] have shown that container-based virtualization has numerous advantages over traditional Cloud/hypervisor-based virtualization including reducing the performance overheads that arise when creating new instances.

One of the core advantages of Cloud-based solutions is to scale up and down with demand. For many applications it is essential to support auto-scaling [7], i.e. dynamically scaling to up/down to meet on-demand computing. Ideally this whole process should be fully automated and not require any manual intervention. There are a range of technologies and approaches that have been taken to support auto-scaling using container-based technologies. This project investigates auto-scaling solutions that have been implemented for a software stack supporting the helicopter training continuum of the Australian Defence Forces (ADF). Specifically we consider containerised application scaling of a strategic discrete event simulation, optimisation and analysis system for manpower planning (ATHENA) with specific focus on addressing the needs and demands of the helicopter pilot training continuum. We present ATHENA and demonstrate the use of Docker and Kubernetes to build a cloud-native application with orchestration on the Cloud.

The remainder of this paper is organised as follows. Section 2 covers related work focused on auto-scaling. Section 3 provides an overview of the ATHENA platform and its core functionality. Section 4 discusses the ATHENA deployment, containerisation and scaling using Docker. Section 5 presents the auto-scaling solution that is supported and Section 6 discuss the results of the solution. Finally section 7 give concluding remarks and identifies areas for future work.

II. Related Work

The concept of auto-scaling is not new. Early research on auto-scaling dates back to IBM's MAPE-K architecture to support Autonomic Computing – computing systems that were able to manage themselves given high-level objectives from administrators [8]. They identified key autonomic elements – Monitor, Analyse, Plan, Execute, Knowledge – which provided the reference model for autonomic control loops used for self-managed autonomic computing systems.

In [9] further challenges were identified for auto-scaling in each of the MAPE-K phases. They created a taxonomy for auto-scaling web applications in clouds. [9] presented a survey of auto-scaling surveys covering scaling indicators and different types of metrics such as resource estimation with rule and threshold based approaches, multi-tier application scaling and container-based auto-scaling. Although not specific to auto-scaling, [10] provides a survey and taxonomy on resource optimisation for executing Bag-of-Task applications on Public Clouds. [11] discusses dynamic provision with Aneka to burst out to Public Clouds from on-premise Private Clouds (Hybrid Cloud) to off-load deadline-driven data intensive applications required for big data processing tasks.

Auto-scaling techniques are also discussed in [12] where they focus on auto-scaling algorithm designs. According to [12], the Kubernetes Horizontal Pod Autoscaling (HPA) offers a reactive-approach where HPA is implemented as a threshold-based reactive controller that automatically adjusts the required resources based on application demand, e.g. through a control loop. Other approaches discussed include threshold-based rules which scale up/down based on observed CPU load or average response times.

Queuing theory is another approach that has been widely explored. It is based on a model of virtual machines, containers and/or application tiers as the basis for forming queues of requests.

978-1-7281-0359-4/18/$31.00 ©2018 IEEE
DOI 10.1109/UCC-Companion.2018.00076

When the queues are filled, the system needs to scale-out. When the queues are empty, the system can scale-down. This approach has also been considered for large-scale stream processing on the Cloud and automatic scaling on demand using technologies such as STORM and Heron [13].

Time-series analysis is a further proactive auto-scaling approach that uses the past history of time-series data, e.g. CPU use, to predict future values. Reinforcement learning is a machine learning based approach that supports automatic decision-making with no prior knowledge of the system model. [14] also reviews cloud auto-scaling approaches and presents a combination of fuzzy logic controller with reinforcement learning for auto-scaling Cloud/OpenStack-based infrastructure based on HTTP traffic load. It proposes a model-driven approach that focuses on attempting to converge resource utilisation to an optimal policy.

There has been a variety of efforts that have measured the performance of container technologies [15]. These have mostly focused on either comparing the performance of running containerised applications against that achieved by running the standard versions of the same applications on virtual machines VMs, and/or physical servers, or assessing their isolation capabilities, i.e. how the performance of an application running on a container or on a VM is impacted by external load in the same computing node. Quetier et al. [16] were among the first to highlight the performance advantages of using kernel-level virtualization of computing resources. They showed that this approach provides much better start-up time for applications, thus helping to achieve flexibility and scalability.

Other works focused on assessing the performance penalties incurred by container based technologies and virtual machines when particular physical resources are under load. Morabito et al. [17] showed that container-based solutions perform better than their VM-based counterparts. Felter et al. [18] used the SysBench OLTP benchmark on a single instance of the MySQL database management system, considering virtualized environments based on containers and VMs. Their results showed that the use of containers gives rise to a very small degradation in performance, when compared to a system that uses no virtualization technology. On the other hand, using vanilla Infrastructure-as-a-Service (IaaS) platforms and creating, configuring and deploying instances, this overhead can be as high as 40%.

Xavier et al. [19] compared the performance isolation capabilities of several container technologies, and compared them against the performance isolation provided by virtualization technologies. They used benchmarking suites with stress tests related to CPU, memory, disk and network. Their results showed that container technologies experience minimal interference regarding the CPU. On the other hand, other resources suffered more interference when container technology was used, as compared to the interference that occurs when a VM was used. In a later work, Xavier et al. [20] performed a deeper evaluation of the performance interference that disk-intensive containerised applications suffer under different stress scenarios. Their results show that in some scenarios, the performance degradation could be as high as 38%.

There has been minimal work in assessing the overheads induced by the use of tools to orchestrate the execution of multiple containers. [15] assesses the start up time of containers for both Docker Swarm and Kubernetes in experiments with as many as 30,000 containers on a cluster of a thousand nodes. The results showed that more than half of the Docker Swarm nodes could start a container in less than 0.5s when the cluster was no more than 90% full, whilst over half of the Kubernetes nodes could only start a container in over 2s when the cluster was 50% full. They identified that if the cluster of Kubernetes nodes was over 90% full, the start-up time of containers could exceed 124s and ultimately require manual intervention.

In supporting auto-scaling, numerous factors need to be considered. One key area of concern is the notion of state of the application to be scaled. As a case study, we present the ATHENA application and how it was developed and ultimately refactored to support auto-scaling across the Cloud.

III. ATHENA Platform

ATHENA is a state-of-the-art, purpose-built system developed for the Australian Defence Science and Technology Group (DSTG - www.dstg.defence.gov.au) [21] [22]. ATHENA is a strategic simulation and analysis system focused on manpower planning. The aim of the system is to tackle workforce planning challenges of the ADF across their training continuum. ATHENA provides a framework designed to address the needs of ADF to perform *what-if* scenario analysis based on computationally intensive simulations, e.g. what if we close a flight school, what if the number of instructors do not need the needs and demands of the ADF in 5 years time, what should the intake of new students be in the next few years? To address these issues, ATHENA provides facilities for simulation, visualisation and analysis of personnel, e.g. instructors, flight crew and cadets/trainees, and resources, e.g. flying schools and the courses/examinations that pilots must pass to progress in their flying careers within the ADF.

The ATHENA architecture is a multi-user, multi-tier microservices web application. The major components comprise the user interface, a backend web service, databases, a message broker and simulation workers. The backend service is based on a Spring framework-based modular application, written in Java. The backend service provides logic related to the strategic aircrew training continuum. It supports management interfaces, business logic, data persistence and distributed computation tasks. The data layer is composed of a PostgreSQL database which stores access control data, as well as a document-oriented NoSQL MongoDB for storing all data related to the simulation model. Computation of simulations are performed by a network of lightweight workers, also written in Java, which communicate with the backend via Java Message Service (JMS). Apache ActiveMQ is used for reliable delivery of messages containing simulation requests, results and status updates. The user interface is based on a AngularJS JavaScript Single-Page Application. The UI communicates with the backend application server using both the HTTP REST API and WebSocket channels.

Figure 1 shows an example of ATHENA used for a 10 year simulation of the ADF helicopter pilot continuum. The Scenario view (top) is given as a Sankey diagram. The upper part shows a visualisation of the layered flow structure for the flow of the training continuum (*left to right*). In this, students/cadets enter from the left from a variety of locations, e.g. University graduates in aeronautics, and undertake a variety of courses, e.g. basic flight training, advanced training at specialist schools. These are typically at locations around Australia. Students may pass or fail these

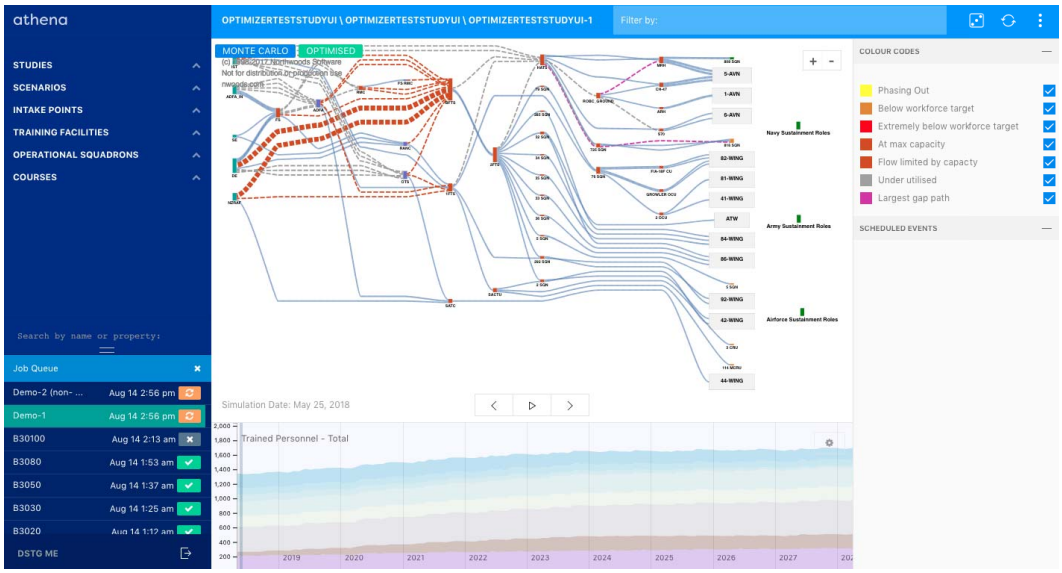

Figure 1. ATHENA Scenario View

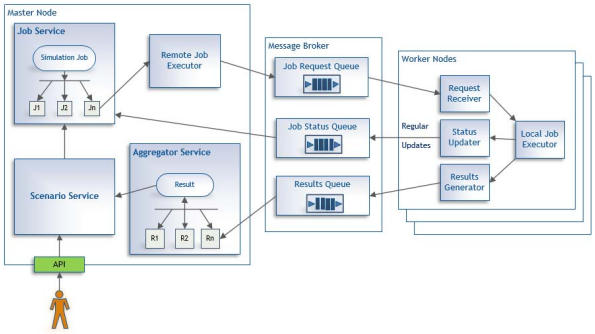

Figure 2. ATHENA Remote Job Execution

courses and if they fail then they typically cease their career in the ADF. As seen there are numerous pathways/careers in the training continuum, e.g. for those that wish to become pilots or navigators. Trained pilots may well go on active service duty or eventually become instructors. Too few or too many instructors impacts on the continuum, e.g. intake of future cadets/trainees will be impacted or courses will pass or fail more students depending on the flow of personnel. It is important to note that this is challenging since there are limited resources, e.g. typically only a hundred or so individuals that become fully trained pilots in the ADF. The Scenario timeline (bottom) shows the statistics of trainees and/or instructors over the whole simulation time span. Monte Carlo simulations and recruitment optimisation algorithm are used to create these results.

The ATHENA simulations can be computationally intensive, especially if one of the optimisation algorithms is activated. For this reason, a master-worker remote execution system has been designed that offers high-performance, scalable simulations as shown in Figure 2. Monte Carlo simulations, composed of many repetitions of the same process, are inherently a good fit for this system due to their embarrassingly parallel nature. In the master node, the job execution system is composed of a modular set of services. Once a scenario is submitted for simulation, the *Scenario Service* deals with assembling a simulation model with the correct inputs and creates one sub-scenario for each Monte Carlo repetition. Effectively, the simulation is parallelised into N independent jobs, created by the *Job Service*, which is responsible for managing the job state, i.e. creating, updating and cancelling job executions. The *Remote Job Executor* interfaces to the message broker via the JMS protocol. At this point, the *Job Request Queue* will contain multiple job definitions, ready to be consumed by workers.

There can be many worker nodes, which consume messages from the *Job Request Queue*. These can interpret the job definition, read job inputs, and execute parts of the simulation model. As jobs are dequeued, start and finish execution, the worker sends status updates via the *Job Status Queue*, which are in turn consumed by the master node to provide progress updates to the end-user. Each individual simulation produces one result. These are sent via the *Results Queue* and consumed by an *Aggregator Service* that performs statistical aggregation, thus producing the mean and confidence intervals of all model statistics over all simulation runs. The status of simulations is shown in bottom left of Figure 1 with running (orange) and completed (green) boxes shown. It is the ability to scale these simulations in near real time across the Cloud that form the basis for the cloud-native auto-scaling of ATHENA application that is the focus of this paper.

IV. Requirements for Scaling ATHENA

For the initial use case of ATHENA, a monolithic, vertically scalable setup was suitable. As more organisations within the ADF showed interest in using ATHENA, a robust horizontally scalable platform was needed. Furthermore, certain advanced use cases were identified, which were meant to rigorously inspect and test the simulation and optimisation engine. These experiments required running the simulations a large number of times (typically

1,000-10,000 times) with different values for all input parameters. The results of these simulations were then analysed to explore relationships between inputs and bottlenecks in the operational pipelines. The existing architecture was not feasible for this work load, and elastically scaling ATHENA became essential to running these benchmarks within a reasonable time period.

ATHENA can be computationally expensive and should ideally leverage scalable and flexible infrastructure. ATHENA has been deployed on the National eResearch Collaboration Tools and Resources (NeCTAR - www.nectar.org.au) Research Cloud. NeCTAR is a federally funded project that offers an OpenStack-based Cloud infrastructure that is free to all academic researchers across Australia. NeCTAR currently offers 30,000 physical servers accessible through multiple availability zones across Australia. This is complemented by large-scale data storage solutions offered (again free!) to Australian academics as part of the Research Data Services (RDS - www.rds.edu.au) program. As part of the RDS program, the University of Melbourne currently makes available over 5 Petabytes of data storage to researchers. NeCTAR itself provides an Infrastructure-as-a-Service platform where instances of virtual machines can be created either through the NeCTAR dashboard (accessible through the Internet2 Shibboleth-based Australian Access Federation (AAF - www.aaf.edu.au)) or through associated tooling, e.g. use of libraries such as Boto (https://pypi.python.org/pypi/boto) for instance creation and management and libraries such as Ansible (https://pypi.python.org/pypi/ansible) for deployment and configuration of software systems.

ATHENA uses NeCTAR as an IaaS Cloud provider. The ATHENA deployment leverages two tenancies with total number of 300 vCPU cores ranging between 4 cores to 16 cores per instance. Unlike typical IaaS uses of NeCTAR, ATHENA uses a container-based approach focused predominantly around the Docker technology (www.docker.com) for auto-scaling. The ATHENA stack was originally deployed in a traditional IaaS manner, e.g. using scripting technologies such as Ansible to create instances and configure/deploy the ATHENA software through Ansible Playbooks. Initially ATHENA was deployed onto a single large VM instance (16 cores and 64Gb RAM). The advantage of this approach was the almost zero network latency between software component communications. However this approach had vertical scaling limitations, e.g. it could only run on a single VM and larger scale simulations were not possible. Horizontal scaling was thus essential. The first step to scale horizontally was to identify the *State* of the software stack and separate out **stateful** and **stateless** components. The components were then built into Docker images and run as containers using a pool of worker nodes as shown in Figure 3. With this arrangement, horizontal scaling was possible. The solution allowed more database nodes to be supported for MongoDB sharding and replication to handle the terabytes of simulation data that can be generated. Further worker nodes could also be added into the dedicated worker pool for higher throughput.

Unlike use of IaaS scaling where VM instance creation, configuration and activation can take several minutes, the ATHENA solution was required to be far more performance-oriented and reactive to ensure the user experience was not diminished. A dockerised version of ATHENA including orchestration capabilities to meet this dynamic scaling was thus need. We utilise Kubernetes for this purpose, based on experiences comparing Docker Swarm and Kubernetes in other work.

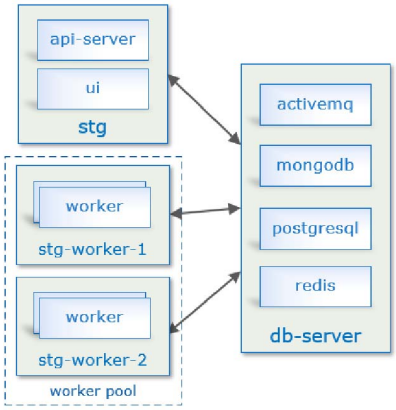

Figure 3. ATHENA worker on different nodes to form a pool of workers

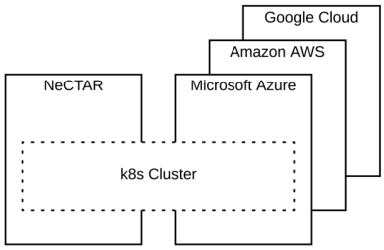

Figure 4. Kubernetes Cluster across different Cloud Providers

V. Auto-scaling ATHENA on the Cloud

As noted ATHENA leverages container-based technologies for auto-scaling. As market leader, Docker was used as the core container-based solution and Kubernetes was used as the container orchestration. Docker was thus an empirical choice based on the maturity of the software and built-in auto-scaling support through Kubernetes (https://kubernetes.io). Kubernetes is a production grade container orchestration system designed for the deployment, scaling, management, and composition of application containers across clusters of hosts. It provides a robust container-management system that creates a virtual abstraction layer on top of Cloud platforms that is used for deploying and maintaining scalable distributed systems. This abstraction enables developers to deploy their applications consistently across different Cloud providers. Figure 4 shows this concept.

Kubernetes is the third incarnation of Google's own container-management systems [23]. Since the process of containerisation encapsulates the application, Kubernetes introduces an *Application-Oriented Infrastructure* to abstract away details of hardware and operating systems from application developers. Kubernetes has a breadth of functionality that grows daily and it's associated documentation is outdated quickly. Every layer of Kubernetes provides several ways to setup and configuration clusters that can be used for a particular cloud deployment. For ATHENA, we explored numerous options and eventually selected *kubeadm* to bootstrap the Kubernetes cluster for ATHENA deployment.

In Kubernetes, the concept of a Pod is used to encapsulate containers. A Kubernetes Pod object holds one or more containers and, introduces an *IP-per-Pod* network model. This implies IP addressing is at the Pod scope. Therefore, containers within a Pod share their network namespaces including their IP address. The Pod networking requirement states that nodes and containers can communicate each other without Network Address Translation (NAT) [24]. Therefore, the IP that a container sees itself as is the same IP as others see it. The Kubernetes network model bars usage of intentional network segmentation policies such as NAT. Kubernetes documentation offers several ways for overcoming this and overlay network using *kube-flannel* add-on was adopted as the ATHENA solution.

In Kubernetes, Pods are ephemeral. That is, a Kubernetes cluster can replicate Pods (destroy and re-create new ones) for dynamically scaling up or down, for self-healing and/or for self-managing purposes. As a given Pod can be destroyed or recycled, the Pod IP address may subsequently change. This is challenging for application developers to keep track of. The Kubernetes *Service* is a network abstraction layer used to maintain a consistent endpoint within a cluster. The Kubernetes *Service* can be also seen as a front-end service proxy for other Kubernetes objects such as Pod, Deployment, ReplicaSet, etc.

For Service Discovery purposes, Kubernetes runs internal DNS. This comes in as Add-ons. *Kubeadm* deploys a *kube-dns* Pod for this purpose. Kubernetes offers several possibilities including *NodePort, LoadBalancer, HostNetwork, HostPort, Ingress* to expose a service endpoint to the outside world. For the ATHENA production setup, the cloud provider's *LoadBalancer* or *Ingress* should be used, however, on NeCTAR, every VM instances gets a Public IP address. Hence HAProxy routing was adopted. As with Docker containers, volumes in Kubernetes are ephemeral, i.e. data will be lost if the Pod is restarted. As such, the ATHENA production system utilises a combination of MongoDB and PostgreSQL to persist data, both of which exist outside of the Kubernetes cluster.

To support auto-scaling, it is essential to have access to real-time monitoring information from the underlying applications and infrastructure. Kubernetes core components are instrumented and exposed at `/metrics`. By default, metrics are given in Prometheus format[1]. When making requests to `/metrics` on the component being monitored, a set of line delimited metrics are returned. Amongst the Kubernetes core components, *kubelet* is one of the most important. A kubelet supports containers running inside a Pod. They support probing of containers and returning the status back to the *kube-apiserver*. A kubelet also comes with an embedded cAdvisor[2] instance which collects, aggregates, processes and exports metrics such as CPU, memory and network usage of running containers. A kubelet also exposes cAdvisor metrics at `/metrics/cadvisor`.

Before Kubernetes v1.9, the most popular stack for consuming these metrics and storing them as timeseries data was Heapster + InfluxDB. However, as of v1.10, Heapster has been deprecated based on an overhaul of the Metrics API used in the Kubernetes instrumentation and monitoring architecture. The *Prometheus Operator* and *kube-prometheus* stack are now extensively used.

[1]https://prometheus.io/docs/instrumenting/exposition_formats/

[2]https://github.com/google/cadvisor

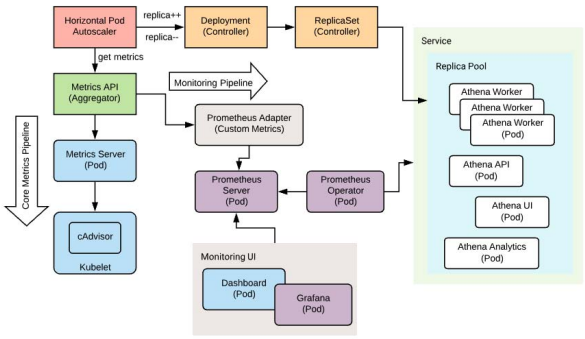

Figure 5. HPA and Monitoring for ATHENA

Prometheus is primarily a pull-style monitoring platform and hence it has less intrusive performance implications, i.e. it does not impose stress and load on the monitored application. Furthermore, Prometheus offers many third-party metrics exporters[3] such as the JMX exporter which can export from a wide variety of JVM-based applications (e.g. ActiveMQ, Kafka) as well as timing and monitoring HTTP Requests/Responses through Apache, Nginx and HAProxy metrics and workloads.

The Prometheus Operator creates, configures, and manages Prometheus monitoring instances and supports the monitoring of related namespaces. It includes additional resources such as an Alert Manager and Service Monitor to the Kubernetes API. The Service Monitor uses Kubernetes's Label and Selector to intercept Kubernetes Services and their exposed Prometheus-based format. The Metrics Server (Heapster replacement) is used to observe cluster-wide metrics and monitoring from the Kubelet Summary API.

The Kubernetes Horizontal Pod Autoscaler (HPA) dynamically adjusts the number of Pod replicas in a given deployment based on the observed CPU utilisation. The HPA is implemented as a Kubernetes API resource with an associated controller. The first version of HPA scaling of Pods was based on observed CPU utilization and memory usage. In Kubernetes 1.6, the Custom Metrics API was introduced that enabled HPA to access arbitrary metrics through the REST API. In Kubernetes 1.7, the API server aggregation layer allowed third party applications to extend the Kubernetes API by registering themselves as API *Add-ons*.

Figure 5 depicts the full orchestration of Prometheus metrics monitoring stack for ATHENA using HPA. In the Core Metrics pipeline, the Metrics Server retrieves node and container metrics from the Kubelet cAdvisor. In the Monitoring pipeline, the Prometheus Operator collect metrics through its Service Monitor and stores them in Prometheus. The Kubernetes Custom Metrics API along with the API Aggregation Layer make it possible for monitoring systems like Prometheus to expose application-specific metrics to the HPA controller. For applications like the ATHENA API service, it is relatively straight forward to expose metrics on JVM stats, embedded Tomcat (HTTP Requests/Responses traffic load) and any application-specific metrics (e.g. workload and number of Jobs in queue) using Spring Boot Actuator (www.spring.io) and Micrometer (www.micrometer.io).

[3]https://prometheus.io/docs/instrumenting/exporters/

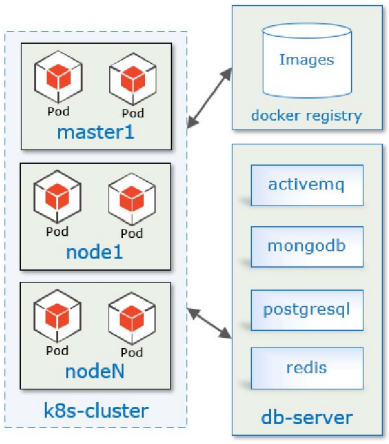

Figure 6. Kubernetes Cluster for ATHENA: Architecture

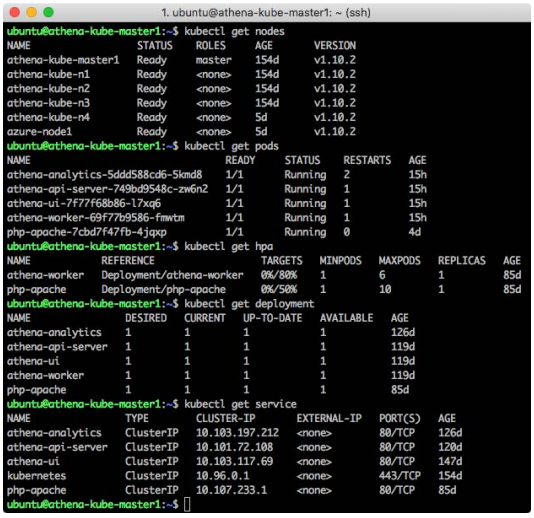

Figure 7. Kubernetes Cluster for ATHENA: Terminal

Figure 6 depicts the architectural view of ATHENA deployment within the Kubernetes cluster. The Docker Registry is a container image registry hosted through the Nexus repository manager.

A. HPA Auto-scaling Algorithm

According to [24], the HPA implementation of the ATHENA auto-scaling algorithm works in the following manner:

1) A Control Loop is set for 30 seconds by default, however this is configurable;
2) Periodically Pods are queried to collect information on their CPU utilization;
3) The arithmetic mean of the Pods' CPU utilization with a given target is compared;
4) The number of Pod replicas required is checked to match the target based on:
   - MinReplicas $\leq$ Replicas $\leq$ MaxReplicas
   - CPUutilization (C) = recent CPU usage of a Pod (averaged over the last 1 minute) / CPU requested by the Pod;
   - TargetNumOfPods = ceil(sum(CurrentPodsCPUUtilization) / TargetCPUUtilizationPercentage (T)), i.e.

$$TargetNumOfPods = \left\lceil \left(\sum_{n=1}^{n} C_n \right) / T \right\rceil$$

5) Scale-up can only happen if there was no rescaling within the last 3 minutes to avoid temporary CPU fluctuations;
6) Scale-down will wait for 5 minutes from the last rescaling to avoid temporary CPU fluctuations, and
7) Any scaling will only be made if within 10% tolerance such that: avg(CurrentPodsConsumption) / TargetCPUUtilizationPercentage drops below 0.9 or increases above 1.1.

VI. Results and Findings

Figure 7 shows the Kubernetes cluster running the ATHENA software stack. The ATHENA Kubernetes cluster was comprised of five VMs (each with 12 vCPU, 48GB RAM) from the NeCTAR Research Cloud and one experimental node (4 vCPU, 16GB RAM) from the Microsoft Azure cloud.

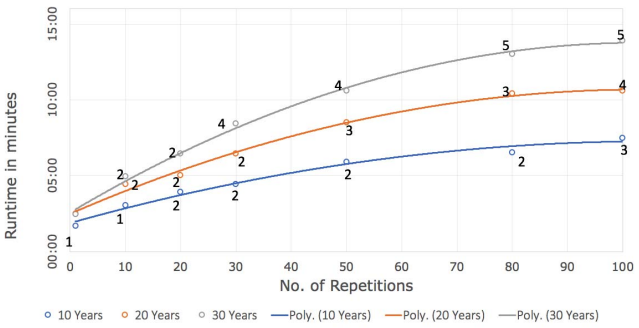

Figure 8. ATHENA Worker Runtime Benchmarking

To experiment with the auto-scaling setup and benchmarking of ATHENA, we configured the ATHENA Worker HPA to use 80% target CPU utilisation with 1 CPU resource request for each Worker Pod instance. We set the HPA replication factor to a minimum of one Pod to a maximum of six Pods. It is noted that the Azure VM node was excluded in the ATHENA runtime benchmarking.

Figure 8 shows the ATHENA Worker runtime for a simulation of 10, 20 and 30 years and different number of Monte Carlo repetitions. The integers next to lines show the maximum number of Pods spawned by the HPA algorithm. The runtime chart clearly shows that as we increase the workload (i.e. the number of repetitions and the simulation span) HPA successfully spawns more compute resources (Pods). The trend lines for each class of 'simulation year span' indicate that the rate of increase in runtime decreases as we add more resources, this is a clear indication that auto-scaling has been successful. However during the experimental runs, it was noted that the auto-scaler doesn't react immediately to usage spikes. By default, the metric synchronisation happens once every 30 seconds and scaling up/down can only happen if there was no rescaling within the last 3-5 minutes. In this way, the HPA prevents rapid execution of conflicting decisions (oscillation).

It was also found that auto-scaling based on observed CPU

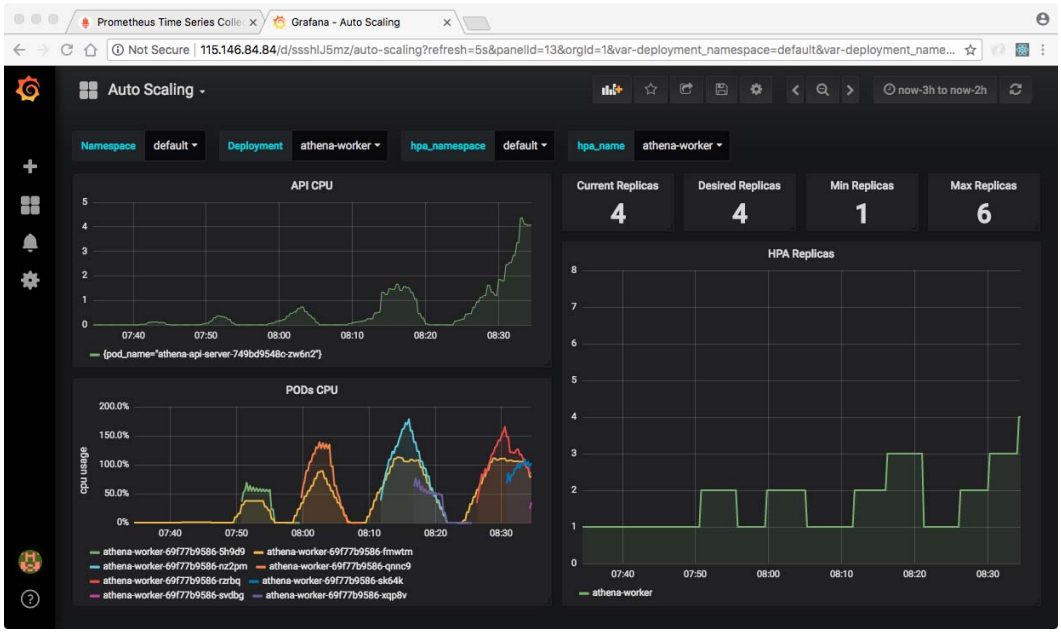

Figure 9. Visualisation of Auto-scaling of ATHENA through the Grafana Dashboard

utilisation alone was not sufficient for ATHENA workers, due to the dynamics of the different simulation runs, i.e. when selecting *simulation alone* or *simulation + optimisation algorithm*. CPU utilisation scaling works better with the optimisation algorithm activated. However, *simulation alone* with Monte Carlo jobs executes rapidly, and hence it missed the scaling observation period delay time and consequently, it missed the HPA replication trigger cycle. As a result there was only one worker Pod instance used for the simulations. This can be tuned accordingly to the target threshold and CPU utilisation feedback control loop, however, it is still an imperfect solution. Further refinements to the system are also possible, e.g. in addition to CPU utilisation, observing job queue metrics could be implemented for improved workload balancing.

Figure 9 shows a screenshot of the Grafana (www.grafana.com) front-end monitoring system. The auto-scaling dashboard was custom built with Grafana for observing the ATHENA Worker Pod replication created by the Kubernetes HPA controller. The auto-scaling dashboard is vertically split into two parts. On the *right* hand side, the HPA status is shown in boxes such as the number of *Current Replicas* and the number of *Desired Replicas*. These are derived from the HPA algorithm and the minimum and maximum HPA replication factor that has been set. The *HPA Replica* graph shows the discrete steps of the ATHENA Worker Pods scaling up/down during a simulation run based on demand. On the *left* hand side, the *API CPU* graph shows the CPU usage of the ATHENA API server. The *PODs CPU* graph shows the associated CPU usage of the ATHENA Worker Pods.

Since we set the minimal computation resource required for simulation Workers to one Pod unit, there will be at least one simulator engine running constantly, which we denote as the primary Worker. The primary Worker Pod's CPU usage ('athena-worker-69f77b9586-fmwtm') is represent as the *Yellow* trend in Figure 9. Whereas, the CPU usage of the additional elastically-scaled Worker Pods come and go as the HPA controller spins up or evicts the Pods as required based on real time work load.

VII. Conclusions and Future Work

In order to produce a reliable auto-scaling system, it is important to understand the target application. In this paper we presented an overview of the defence-oriented ATHENA system and how it was designed to exploit Docker-based scaling across the Cloud through separation and consolidation of stateless and stateful considerations. To support auto-scaling we described the capabilities of Kubernetes and how it can be used for auto-scaling of pods. It is important to note that this system is a live and functional system that is used by all parts of the ADF. At present the platform is in the process of being refined to accommodate the particular training needs and demands of submarine personnel and the issues of onshore/offshore constraints.

Auto-scaling system cannot meet the user performance demands by simply relying on CPU utilisation and memory usage metrics alone. Most web and mobile applications require auto-scaling based on *Requests Per Second* to handle bursty traffic and stochastic user load. With the new *Custom Metrics API* feature introduced in Kubernetes, we intend to extend the API and expose ATHENA's metrics to Prometheus, so that it can be consumed by the HPA controller and auto-scaled better. Auto-scaling can also be triggered by the ActiveMQ job queue length exceeding some empirical threshold. *Threshold-based Reactive approach* with *Control Theory* auto-scaling techniques can be explored.

Effective monitoring and use of a range of metrics is essential for auto-scaling. Auto-scaling with Kubernetes is a non-trivial task to setup, especially for *Production Ready* systems that are required

to operate in Private Clouds (such as the ADF Azure Cloud) and public Clouds such as the NeCTAR Research Cloud.

For the future, instrumenting ATHENA with Prometheus and exposing metrics to fine tune the auto-scaling will be explored to accommodate bursty usage whilst offering high availability and quality of service. We also intend to explore and design a range of other auto-scaling algorithms and exploit different techniques based on Time-series Analysis for Kubernetes, as well as implementing Docker Swarm based auto-scaling. To accommodate infrastructure scaling, we are already able to observe metrics produced by Kubernetes and Prometheus and offer auto-scaling nodes based on these metrics.

The use of Kubernetes here is not unique, however the application domain and combining defence-based services with dynamic Cloud-based technologies is unique. There are many challenges in prototyping on public clouds to meeting the robust, enterprise level services and infrastructure demands on private and often mission critical systems. Through close work with the DSTG many of these non-technical issues are being overcome. There are numerous other potential end users of these systems including defence forces outside of Australia and other application domains where manpower planning is required.

A. Acknowledgments

The authors would like to thank the National eResearch Collaboration Tools and Resources (NeCTAR - www.nectar.org.au) for the (free) access to and use of the Cloud resources that were used in this paper. Acknowledgments are also made to the Australian Research Data Services (RDS - www.rds.edu.au) project for the (free) data storage systems that were utilised in this paper.

REFERENCES

[1] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "A view of cloud computing," *Commun. ACM*, vol. 53, no. 4, pp. 50–58, Apr. 2010.

[2] M. J. Scheepers, "Virtualization and containerization of application infrastructure: A comparison," in *21st Twente Student Conference on IT*, vol. 1, no. 1, 2014, pp. 1–7.

[3] D. Merkel, "Docker: Lightweight linux containers for consistent development and deployment," *Linux J.*, vol. 2014, no. 239, 2014.

[4] M. Httermann, *DevOps for Developers*, ser. Expert's voice in Web development. Apress, 2012.

[5] Z. Kozhirbayev and R. O. Sinnott, "A performance comparison of container-based technologies for the cloud," *Future Generation Comp. Syst.*, vol. 68, pp. 175–182, 2017.

[6] J. Che, C. Shi, Y. Yu, and W. Lin, "A synthetical performance evaluation of openvz, xen and kvm," in *2010 IEEE Asia-Pacific Services Computing Conference*, Dec 2010, pp. 587–594.

[7] R. O. Sinnott and W. Voorsluys, "A scalable cloud-based system for data-intensive spatial analysis," *Int. J. Softw. Tools Technol. Transf.*, vol. 18, no. 6, pp. 587–605, Nov. 2016.

[8] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, pp. 41–50, Jan 2003.

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

[10] L. Thai, B. Varghese, and A. Barker, "A survey and taxonomy of resource optimisation for executing bag-of-task applications on public clouds," *CoRR*, vol. abs/1711.08973, 2017.

[11] A. N. Toosi, R. O. Sinnott, and R. Buyya, "Resource provisioning for data-intensive applications with deadline constraints on hybrid clouds using aneka," *Future Generation Comp. Syst.*, vol. 79, pp. 765–775, 2018.

[12] T. Lorido-Botran, J. Miguel-Alonso, and J. A. Lozano, "A review of auto-scaling techniques for elastic applications in cloud environments," *J. Grid Comput.*, vol. 12, no. 4, pp. 559–592, 2014.

[13] T. M. Truong, A. Harwood, and R. O. Sinnott, "Predicting the stability of large-scale distributed stream processing systems on the cloud," in *Proceedings of the 7th International Conference on Cloud Computing and Services Science - Volume 1: CLOSER,*, INSTICC. SciTePress, 2017, pp. 603–610.

[14] H. Arabnejad, C. Pahl, P. Jamshidi, and G. Estrada, "A comparison of reinforcement learning techniques for fuzzy cloud auto-scaling," in *Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, ser. CCGrid '17. Piscataway, NJ, USA: IEEE Press, 2017, pp. 64–73.

[15] J. Nickoloff. (2016, Nov) Evaluating container platforms at scale. Retrieved from Medium, June 2018. [Online]. Available: https://medium.com/on-docker/evaluating-container-platforms-at-scale-5e7b44d93f2c

[16] B. Quetier, V. Neri, and F. Cappello, "Scalability comparison of four host virtualization tools," *Journal of Grid Computing*, vol. 5, no. 1, pp. 83–98, Apr. 2007.

[17] R. Morabito, J. Kjllman, and M. Komu, "Hypervisors vs. lightweight virtualization: A performance comparison," in *2015 IEEE International Conference on Cloud Engineering*, March 2015, pp. 386–393.

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

[19] M. G. Xavier, M. V. Neves, F. D. Rossi, T. C. Ferreto, T. Lange, and C. A. F. D. Rose, "Performance evaluation of container-based virtualization for high performance computing environments," in *2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, 2013, pp. 233–240.

[20] M. G. Xavier, I. C. D. Oliveira, F. D. Rossi, R. D. D. Passos, K. J. Matteussi, and C. A. F. D. Rose, "A performance isolation analysis of disk-intensive workloads on container-based clouds," in *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, March 2015, pp. 253–260.

[21] V. Nguyen, M. Shokr, A. Novak, and T. Caelli, "A reconfigurable agent-based discrete event simulator for helicopter aircrew training," in *Proceedings of the 2016 ISMOR Conference*, 2016.

[22] V. Nguyen, A. Novak, M. Shokr, and K. Pash, "Aircrew manpower supply modeling under change: An agent-based discrete event simulation approach," in *2017 Winter Simulation Conference (WSC)*, Dec 2017, pp. 4070–4081.

[23] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, omega, and kubernetes," *Queue*, vol. 14, no. 1, pp. 10:70–10:93, Jan. 2016.

[24] K. Developers. (2018) Kubernetes documentation. V1.10, Access June 2018. [Online]. Available: https://kubernetes.io/docs/