UNIVERSITY OF MELBOURNE

Auto-scaling an optimisation algorithm using Docker and Kubernetes on the NeCTAR Research Cloud

Student Name:
San Kho LIN

Supervisor:
Prof. Richard O. SINNOTT

Student ID	829463
Subject Code	COMP90019 Distributed Computing Project
Total Credit Point	25
Type of Project	Software Development Project

June 2018

Melbourne eResearch Group
School of Computing and Information Systems
Melbourne School of Engineering

Declaration of Authorship

I, San Kho LIN, declare that this thesis titled, "Auto-scaling an optimisation algorithm using Docker and Kubernetes on the NeCTAR Research Cloud" and the work presented in it are my own. I certify that:

- This thesis does not incorporate without acknowledgement any material previously submitted for a degree or diploma in any university; and that to the best of my knowledge and belief it does not contain any material previously published or written by another person where due reference is not made in the text.
- Where necessary I have received clearance for this research from the University's Ethics Committee and have submitted all required data to the Department.
- The thesis is 10850 words in length.

Signed By: *San Kho Lin*
Date: *4 June 2018*

Abstract

ATHENA is a strategic discrete event simulation, optimisation and analysis system for manpower planning. It is state-of-the-art, purpose-built distributed application. It is containerised and deployed into Cloud platform using Docker technologies. It is a compute intensive application, especially when an optimisation algorithm is activated in simulation. Scaling and auto-scaling of an optimisation algorithm is the center of this thesis research.

Infrastructure scaling on Cloud platform require provider specific API and technology. The containerisation deployment with Docker Swarm and Kubernetes technologies can scale across multi-clouds independently, without the need of Cloud Provider specific API.

Auto-scaling system can not meet SLO and QoS by simply relying on CPU utilisation metrics alone. Most web and mobile applications require auto-scaling based on *Requests Per Second* to handle traffic bursts and user load. With new **Custom Metrics API** feature introduced in Kubernetes, we can extend the API and expose ATHENA's metrics to Prometheus, then can be consumed by HPA controller and auto-scaled. For enterprise integration system like ATHENA, the auto-scaling could be triggered by the ActiveMQ job queue length exceeding some empirical threshold, therefore, **Threshold-based Reactive approach** with **Control Theory** auto-scaling technique can be explored.

Effective monitoring and metrics scraping is the heart of the pipeline for auto-scaler to work reliably well. Auto-scaling with Kubernetes is non-trivial task to setup for *Production Ready* in Private Cloud, such as NeCTAR Research Cloud.

Keywords: auto-scaling, scaling, container orchestration, containerisation, docker, kubernetes, openstack

Acknowledgements

I would like to thank my supervisor Professor Richard Sinnott for giving me opportunity to research on this challenging project. The door to Professor Sinnott's office was always open whenever I ran into a trouble spot or had a question about my research or writing. He consistently allowed this paper to be my own work, but steered me in the right direction whenever he thought I needed it.

I am also indebted to my colleagues who work together in the ATHENA project, their contribution and, influenced this thesis with their ideas and thoughtful criticisms.

I would like to thank my family for providing me with unfailing support and continuous encouragement throughout the years of study and through the process of researching and writing this thesis. This accomplishment would not have been possible without them.

This research was supported by use of the NeCTAR Research Cloud, a collaborative Australian research platform supported by the National Collaborative Research Infrastructure Strategy.

Contents

1 Introduction

Motivation: Today, in software engineering, the rise of Containerisation [1] and Cloud Computing [2] technologies have prominently changed the way we deliver software packaging and deployment. Certainly, application deployment is not quite like it used to be – concepts such as Containerised Application [3], Infrastructure-as-Code (IaC), DevOps [4] are some examples in the modern deployment literature. Advances in virtualization technologies have changed the Operating System (OS) landscape and studies [5] [6] show that container-based virtualizaton are the best fit for application process level isolation and, hypervisor-based full virtualization are better for infrastructure level islolation. Nevertheless, it is evident that the rise of virtualization technologies contributed the emergence of Cloud Computing. One profound benefit of Cloud Computing is giving virtually unlimited number of computing resources – Virtual Machine (VM) and, the elasticity to acquire and release of these resources. Software development is also advancing in more decomposable and serviceable components such as adopting Service-Oriented Architecture (SOA), Microservices and RESTful architecture. These advances are realisation of Distributed Systems – where it defines as *software components located at networked computers communicate and coordinate theirs actions only by passing messages* [7]. Studies [8] show that a **Scalable** cloud-based distributed application can be built and deployed it onto the Cloud platform. This also gives the indication that we can dynamically scale to meet with on-demands computing needs. The idea of **Auto-scaling** system is to give business:

1. to start with the minimal operational computing resource provision at deployment
2. to scale out the system to meet the usage demands by probing particular resource metrics such as CPU utilisation
3. to spin down the computing resources to maintain the minimal operational limit when no usage demand is required
4. to automate this scaling process 1-3 without human operator intervention

This elastic nature of dynamic scaling give *operational optimisation* which contribute business to streamline costs, increase productivity and improve profitability while maintaining the Service Level Objectives (SLO) and satisfying Quality of Service (QoS).

Research: To pursuit this motivation in real world setting, this project investigates the auto-scaling of ATHENA software stack on the Cloud platform. The investigation involves containerised application scaling using Docker Compose, Docker Swarm and Kubernetes orchestration.

The remainder of the thesis is organised as follows. Chapter 2 explain an overview of ATHENA system and its domain driver. Chapter 3 discuss ATHENA deployment, containerisation and scaling using Docker. Chapter 4 discuss the insightful research on Kubernetes, its auto-scaling and finding. Chapter 5 discuss the related studies and reviews. Chapter 6 give concluding remarks and identify areas for future work.

2 ATHENA

2.1 Background

ATHENA is a state-of-the-art, purpose-built system which is engineered by *Melbourne eResearch Group* at The University of Melbourne for the Australian Defence Science and Technology (DST) Group – Training Analysis for Workforce Planning[1] research centre. ATHENA is a strategic simulation and analysis system for manpower planning. The aim of the system is tackling workforce planning challenges of the Australian Defence Force (ADF) Training Continuum.

To begin with, the project purpose is to develop the *Dynamically Reconfigurable Agent-Based Discrete Event Simulator for Aircrew Training*. The project specification was originated in the ADF Helicopter Aircrew Training Continuum (HATC) system. The HATC[9] system is in a constant state of transience such that infrastructure consolidation – e.g. phasing out of Aircraft, rules about criteria for passing individual training components, and changes in policies that govern training in individual schools and squadrons – and, all these events occur rapidly. Changes like these are common in the system and, it inevitably causes perturbations in the flow of students through the training program. Therefore, it requires a framework designed to address the needs of ADF to perform **what-if scenario analysis**. To model such dynamic changes in the HATC complex system, while preserving a common simulation architecture, [9] identified the needs of reconfigurable simulation framework. Ever since, the HATC system evolved into ATHENA and, most recently in [10] discuss ATHENA as a more generic simulator for ADF manpower planning needs, such as not only for Pilot and Aircrew but also for Maritime Warfare Officer and Navy Submariner. These papers discuss the choice of Agent-Based Discrete Event Simulation (AB-DES) hybrid model and the details of the simulation modelling domain concept.

The remainder sections of this Chapter 2 focus more on the software engineering perspective and key excerpts from the ATHENA technical documentation v1.0 [11], an overview of the system, its architecture and major components, and highlight on Bag-of-Tasks (BoT) scaling potential for remote job execution.

2.2 System Overview

ATHENA provides facilities for simulation, visualisation and analysis. These three aspects come together to enable modelling, exploration and analysis of complex workforce and manpower resourcing scenarios.

Architecture: ATHENA is a multi-user, multi-tier web application. Major components include the user interface, a backend web service, databases, a message broker, and simulation workers. Conceptually, the system follows a traditional 3-tier web application, as depicted in Figure 2.1.

A user interface (UI), accessible via a web browser, provides access to the underlying modelling and simulation system. The user interface communicates with the backend system via REST[2] API[3] calls. These backend API calls are intercepted by an authentication and

[1] https://www.dst.defence.gov.au/research-facility/training-analysis-workforce-planning
[2] Representational State Transfer
[3] Application Programming Interface

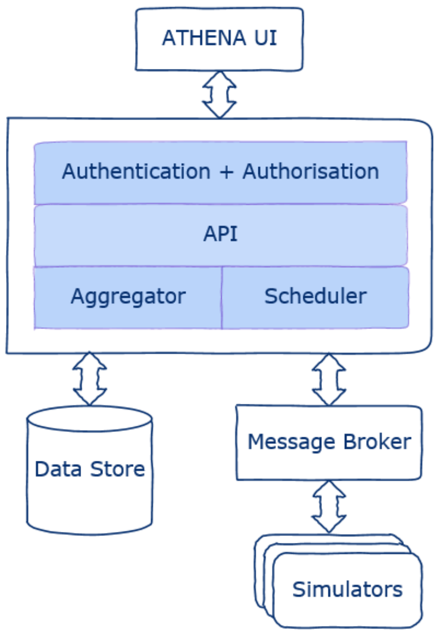


FIGURE 2.1: Conceptual view of the multi-tier component architecture of the ATHENA system

authorization layer, which perform user access control (ACL[4]) checks before despatching to various API endpoints. The backend system communicates with databases for the purpose of data persistence.

Apart from these traditional layers described above, the backend also communicates with a message broker for managing compute-intensive simulation tasks – Bag-of-Tasks job execution using Enterprise Integration pattern – Producer-Consumer Message-oriented architecture. These tasks are performed by a distributed set of workers, each hosting ATHENA's simulation engine.

Technology Stack: Figure 2.2 depicts ATHENA's technology stack. The backend service is a Spring framework-based modular application, written in Java. Primarily, the backend service provides logic related to the strategic aircrew training continuum, managing interfaces, business logic, data persistence, and distributed computation tasks. The data layer is composed of PostgreSQL RDBMS[5], which stores access control data, and 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 JMS (Java Message Service). Apache ActiveMQ manages reliable delivery of messages containing simulation requests, results, and status updates.

The primary user interface is a AngularJS JavaScript Single-Page Application (SPA) loaded in an HTML5 compliant web browser. The UI communicates with the backend application server using both the HTTP REST API and WebSocket channels.

Figure 2.3 shows a 10 year simulation result, the Scenario view which is the Sankey diagram[6] (*upper north*) – visualisation of layered flow structure for the flow of the training continuum (*left to right*). The Scenario Timeline (*bottom south*) shows the statistical data of trainees or instructors over the whole simulation time span. The result also shows that the

[4]Access Control List
[5]Relational Database Management System
[6]`https://en.wikipedia.org/wiki/Sankey_diagram`

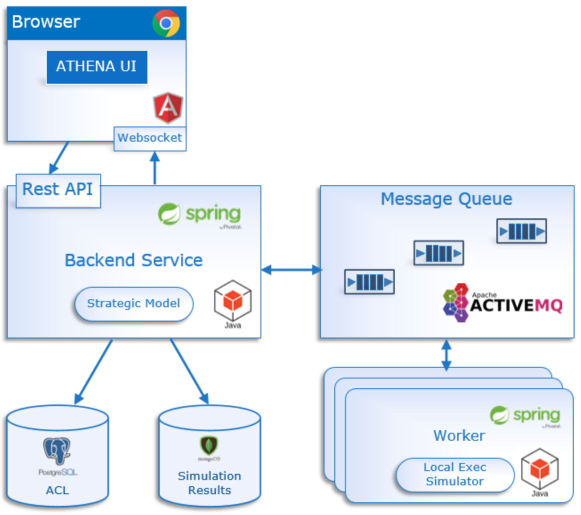


FIGURE 2.2: ATHENA technology stack and their respective interactions

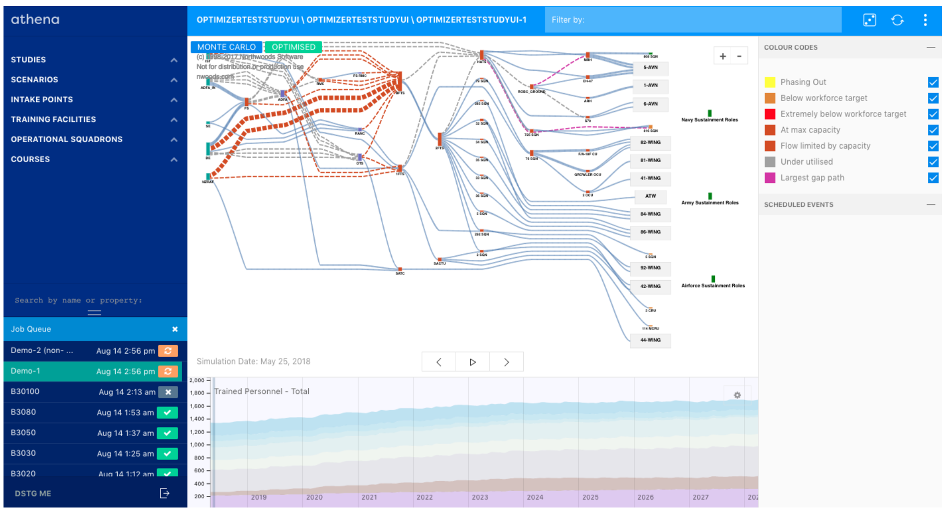


FIGURE 2.3: ATHENA Scenario View

simulation have used **Monte Carlo** simulation and recruitment **optimisation algorithm** by labelling at the top left corner of the Sankey diagram.

2.3 Remote Job Execution and Bag-of-Tasks Scaling Potential

Simulation jobs 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 aiming at offering high-performance and reliable simulations. Figure 2.4 depicts this process.

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

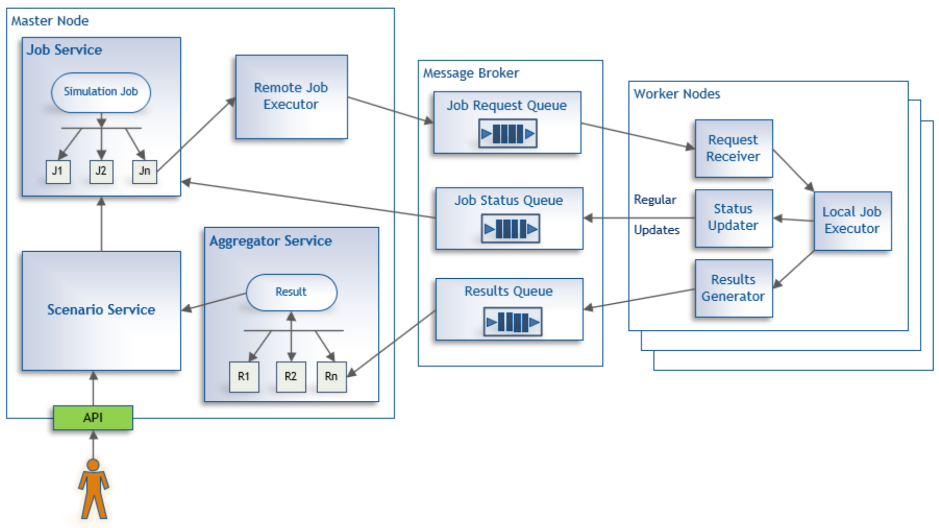


FIGURE 2.4: ATHENA Process of remotely executing simulation jobs

with the correct inputs and creates one sub-scenarios 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 with 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**, interpret the job definition, read job inputs, and perform local execution 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 in order to provide progress updates to the end-user. Each individual simulation repetition produces one result instance, or intermediate result. These are send 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 execution is reliable in the sense that the failure of a worker does not cause the entire simulation to fail, as failed jobs are simply retried by another available worker.

For further details, readers are recommended to read the complete documentations of the ATHENA system in [11].

3 Deployment: Containerisation and Scaling

3.1 Overview

This chapter discuss as the evolution of ATHENA deployment approaches, address the scaling aspect and improvement done on each iteration. The fundamental understanding of Docker technologies[1] is assumed.

Concept: The following diagram 3.1 depicts the one logical deployment of the ATHENA system with no specifics on how to layout this onto computing resources e.g. it could be on one machine or broken down into multiple machines.

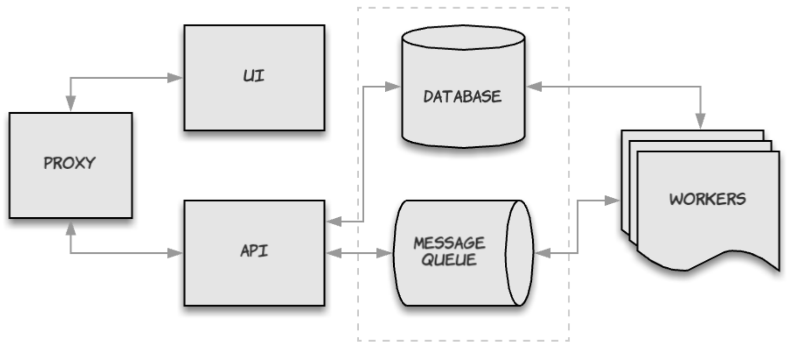


FIGURE 3.1: ATHENA Logical Deployment Stack

Contrary to typical lab or experimental setup that use for benchmarking or comparison purpose, the ATHENA deployment is, somewhat, in a real world setting and live deployments where users are actively using the system. The development and integral releases are pushing into the deployed sites regularly. Therefore, the deployment and operation consideration mainly favour stable steps and reliable approaches. Deployments are regularly snapshot and backed up. The server instance decommissioning are carefully planned and recycled when the machine vertical scaling is needed to replace the already provisioned environment.

Infrastructure: The Australian National eResearch Collaboration Tools and Resources (NeCTAR) is the federally funded research cloud platform where the underlying datacenters span across Australia capital cities. At present, it supports 6000+ virtual machines using around 30,000+[2] vCPUs across Australia.

ATHENA uses NeCTAR as the main Infrastructure-as-a-Service (IaaS) Cloud provider. NeCTAR infrastructure is implemented and managed using the OpenStack cloud computing framework. The ATHENA deployment is leveraging across two tenancies with total number of 300 vCPU cores allocated for the project. The typical choice of instance size is between **m1** and **m2** flavours of **large**, **xlarge** [3] ranging between 4 cores to 16 cores. The choice of base image is *NeCTAR Ubuntu 16.04 LTS (Xenial) amd64*.

[1] https://docs.docker.com/engine/docker-overview/
[2] https://status.rc.nectar.org.au/growth/infrastructure/
[3] https://support.ehelp.edu.au/support/solutions/articles/6000055380-resources-available-to-you

The ATHENA source code version control is hosted on Microsoft Visual Studio Team Services (VSTS)[4]. The deployment is mainly driven by Git Workflow[5], VSTS's build and release pipelines. The VSTS's **Continuous Integration** (CI) and **Continuous Delivery** (CD) [4] are setup to push to the target deployment environments. The terms **DEV**, **STAGING** and **PROD** in Figure 3.2 represent the development, staging and production deployment environments of the ATHENA software stack. Build-agents are the VSTS build slaves for compiling and building the ATHENA artifacts.

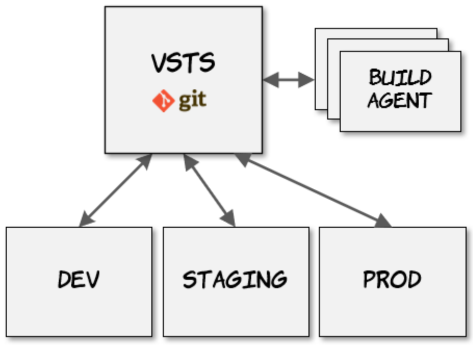


FIGURE 3.2: ATHENA VSTS and Git Driven Deployment

3.2 Containerised Deployment and Scaling

Initially, the ATHENA stack was deployed in a good old-way of using distro packaging tool `apt-get` and running using `systemd/init.d` daemon services. This has performed through SSH terminal session on the node, e.g:

```
apt install mongo
systemctl start mongod
```

IaC Automation Tool: Nonetheless, this were later change to use Ansible[6] as IaC automation tool. This improve the scripted knowledge of what has made changes to the server over the system deployment life cycle. The Ansible Playbook scripts were written according to server role such as `db-server.yaml`, `api-server.yaml`, so on. The pros of Ansible are agentless and, its only dependency is SSH which comes in default with most Linux distribution. The cons side of Ansible is that it requires a bit more care to script it in idempotent way. Otherwise the side-effect is detrimental e.g the use of the following task would wipe `vdd` block device each time Ansible script run.

```
shell (or command): mkfs.xfs -f /dev/vdd
```

For a quick comparison, Chef[7] and SaltStack[8] tools were evaluated as Ansible alternative. The SaltStack can operate in both agent and non-agent mode – whereas in agentless, it operates as light weight as Ansible. Chef requires an agent to be installed on all the nodes in the cluster fleet which is an overhead. On the other hand, Chef gives the most controlled

[4]https://www.visualstudio.com/team-services
[5]https://git-scm.com/book/en/v2/Git-Branching-Branching-Workflows
[6]https://www.ansible.com
[7]https://www.chef.io
[8]https://saltstack.com

environment such as retaining the guarantee system state. A more in-depth comparison is off interest for this paper, but the next contender would be SaltStack. The justification is solely based in favour of the KISS principle – ***K**eep **I**t **S**imple and **S**traightforward*.

Vertical Scaling Issue: Furthermore, in the beginning, the whole ATHENA stack Figure 3.1 was deployed onto the single environment i.e. one VM instance. The pros with this approach is almost zero network latency among software components communication. But it needs a powerful machine. The highest flavour we can request on NeCTAR is *m1.xxlarge: 16 cores, 64GB RAM*, although the real-world experience shows that this is somewhat difficult to get allocation from NeCTAR, which is understandable for a shared research facility. Most importantly, this approach has the potential vertical scaling limit as if requires more resources in one box.

Stateful and Stateless: The first step of scale out approach is to identify the **State** nature of the software stack and split between stateful and stateless components. In addition, the components are built into the docker images (i.e. dockerised) and run as containers. Figure 3.3 depicts the arrangement of this approach – stateful database containers on `db-server` node and, pseudo-stateless API, stateless UI and Worker containers on another node. The scaling is a bit easier now. However, this is still a single node container scaling for the worker's Bag-of-Tasks remote job execution discussed in Chapter 2.

```
docker-compose -f docker-compose.yaml -f staging.yaml  --project-name=stg \
   --scale worker=2
```

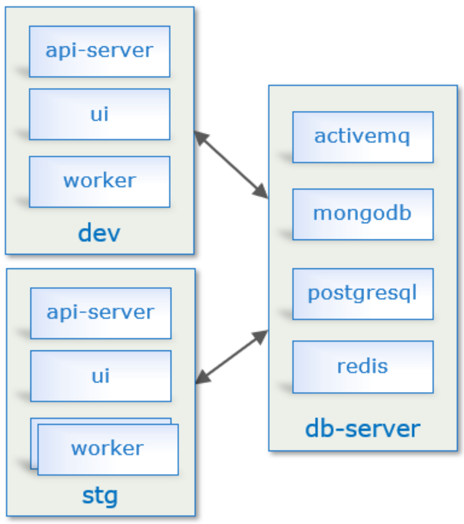


FIGURE 3.3: ATHENA containerised and split by stateful and stateless components on different nodes

Dedicated Workers: As discussed in Chapter 2, the ATHENA Worker's remote job execution and computation can become very CPU intensive when one of optimisation algorithms is activated in the simulated scenario. Further attempt to improve Worker's computation throughput, we can split the worker containers into a pool of worker nodes as shown in Figure 3.4. This solution is expensive but a good compromise if dedicated worker service is needed for QoS purpose.

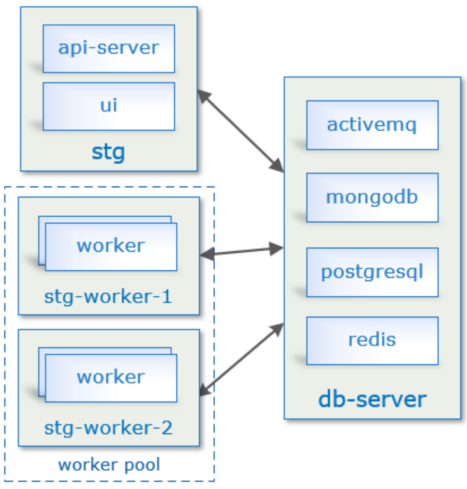


FIGURE 3.4: ATHENA dedicated workers on different nodes to form pool of workers

In fact, this dedicated workers approach is the recommended deployment for Production environment in the ATHENA Deployment Guide v1.0 [11]. It requires network and resource provision as shown in Figure 3.5.

Note that, only ports 80 and 443 are exposed through the proxy-server to the external (public) network. The rest of the system resources form a trusted private network (subnet) among themselves and, is protected behind the firewall for all the ingress network traffic. The private network `10.1.0.0/16` has setup and, a fixed static IP addressing is configured on each of the host systems. The host systems may have 2 NICs (network interface cards) typically eth0, eth1 and, have chosen 1 NIC for connecting this private internal network. The recommended network is Gigabit Ethernet (1GbE) or higher speed inter-network connection is assumed.

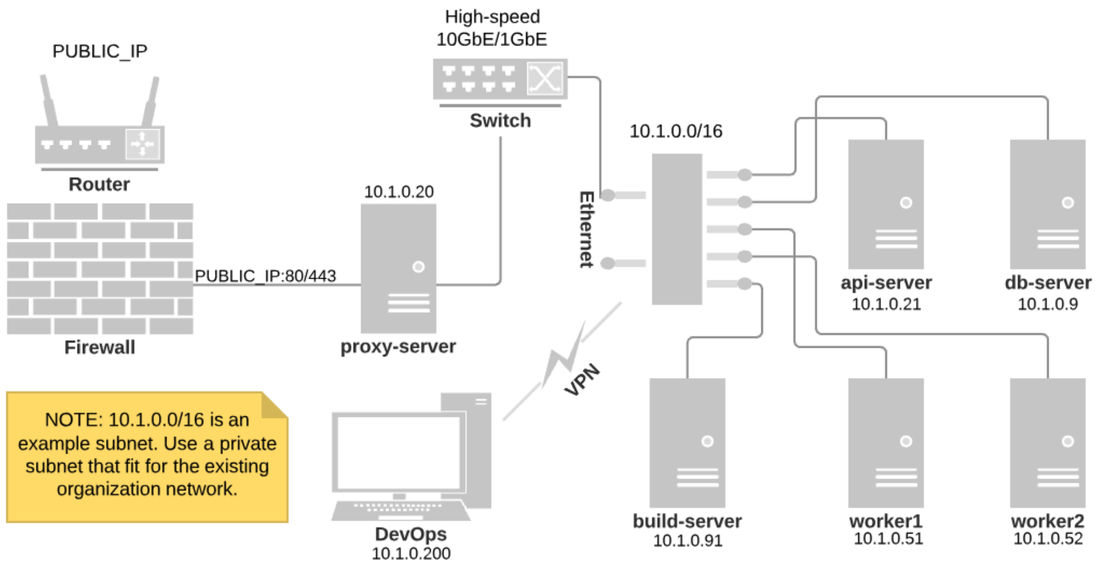


FIGURE 3.5: ATHENA Network Diagram

In this arrangement, we can horizontally scale out the system well. We can add more database nodes for MongoDB Sharding and Replication[9] to handle the terabytes (TB) of simulation data and, to retain these simulation data in the system for operational need. We can add more worker nodes into the dedicated worker pool for higher throughput. In ATHENA version 1.0, the api-server is still pseudo-stateless, in such that the Aggregator component as shown in Figure 2.1 is required to aggregate all the results from the simulators computation. This has a potential bottleneck on the api-server. There is a plan to decompose the Aggregator component in a future ATHENA releases.

3.3 Docker Swarm Mode

To this point, we have identified how we can scale the ATHENA system at different layers. However, the system operator still requires to configure and manage these computing resources manually. A good estimated time to add a new worker with the prepared IaC automation scripts; plus NeCTAR VM instance spin up time takes around 20 to 30 minutes. And the operator will have to spin down the instances where there no need of computation demands and keep the system in minimal operational state. This entails the need for self-managed Autonomic Computing [12] and, the notion of autonomous cluster management and job execution platform from a pool of computing resources – also refer as **Orchestrations** in a more recent deployment literature.

Since we have containerised the entire ATHENA software components, the next best thing is to switch into docker swarm mode. Turning into the docker swarm mode is relatively straight forward. On the chosen node, initialize the manager (master) and add (swarm) worker nodes to form docker swarm cluster. Service scaling is easy, however, the scaling instruction need to manually set it on the swarm manager node through CLI or docker-compose stack deployment `service.[srv_name].deploy.replicas:5` nomenclature.

```
docker swarm init --advertise-addr 192.168.99.100
docker swarm join --token SWMTKN-1-[tonken-scrap] 192.168.99.201:2377
docker swarm join --token SWMTKN-1-[tonken-scrap] 192.168.99.202:2377
docker node ls
ID                             HOSTNAME         STATUS          AVAILABILITY
[nxor..] *   swarm-master         Ready          Active          Leader
[ke35...]    swarm-worker1        Ready          Active
[u2wx...]    swarm-worker2        Ready          Active
docker service scale athena_worker=5
athena_worker scaled to 5
```

For the interest of the thesis length, the details review of how Docker Swarm works[10] is left to its documentation. This thesis takes advantage of the related studies made within the *Melbourne eResearch Group*, of which studies [13] benchmarks on the performance and qualitative comparison of Docker Swarm and Kubernetes. There are also internet articles available on qualitative comparison of these two technologies e.g. Platform9[11].

Base on these literatures, we can conclude that the technical merit of both technologies is closely matched, however, in the context of the research presenting in this paper, the *built-in Auto-scaling feature* in Kubernetes has ruled out for a probable auto-scaling solution for the ATHENA system. The justification to *why Kubernetes* decision is somewhat empirical choice based on maturity of the software. It is also a pragmatic observation over adoption of Kubernetes trend by many Cloud providers and key players, e.g. Google, IBM, Azure, AWS, RedHat, and to name few. The technical insight in Chapter 4 reinforces this hypothesis.

[9] https://docs.mongodb.com/manual/
[10] https://docs.docker.com/engine/swarm/how-swarm-mode-works/nodes/
[11] https://platform9.com/blog/kubernetes-docker-swarm-compared/

4 Kubernetes: Insight and Auto-scaling

4.1 Overview

This chapter focus on the critical review and details setup of Kubernetes (k8s or Kube) cluster for deploying the auto-scalable ATHENA system. The Kubernetes key components are introduced, however, a more details and complete reference left it to the k8s official documentation [14].

With reference to the official website[1], Kubernetes is the *production grade container orchestration system that design for the deployment, scaling, management, and composition of application containers across clusters of hosts*. It is a robust container-management system that create a virtual abstraction layer on top of Cloud platform for deploying and maintaining scalable distributed systems. This abstraction enable users to deploy their applications consistently between different Cloud providers.

Kubernetes is the third incarnation of Google own container-management systems – whereas the k8s is built upon its predecessors Borg and Omega [15] systems over a decade experience of production run at scale. At Google with Borg system, [16] reported that it *runs hundreds of thousands of jobs, from many thousands of different applications, across a number of clusters each with up to tens of thousands of machines. . .*

4.2 Looking at Kubernetes Design

From Borg-Omega to Kubernetes, the Google software engineers learned lessons [15] on how best to tackle with the container-management at design[2] and, bring the architecture to next level[3] to make it easy for developers to write the distributed applications and services that run in cloud and datacenter environments. Looking at Kubernetes, we can observe that the design of the system become more generic and abstraction compare to its predecessors. It goes beyond more than container-management. Since the process of containerisation encapsulates the application, Kubernetes introduces *Application-Oriented Infrastructure* (AOI) with two motivations:

- abstracting away details of machine and operating system from application developer and deployment
- *single application process per container* Design Pattern entails that managing containers mean managing applications, therefore, it shifts the Kube API from machine-oriented to application-oriented and, improves application deployment and introspection

Objects Management: To model a generalised system is a challenging goal for every distributed system designer. To realise this generalisation, at very abstract, Kubernetes is just a **Objects Management** API service. The Kubernetes API does not perform the functional aspect of the system, but the objects its managed deliver these functionality. In Kubernetes, we can define the type of what kind, which does something. The shape of the Kubernetes Object can be expressed in YAML format. The mandatory fields for a Kubernetes Object are summarised in Table 4.1.

[1]https://kubernetes.io

[2]https://kubernetes.io/blog/2015/06/the-distributed-system-toolkit-patterns/

[3]https://kubernetes.io/blog/2016/06/container-design-patterns/

Field	Description
apiVersion	Which version of the Kubernetes API using to create this object.
kind	What kind of object you want to create.
metadata	Data that helps uniquely identify the object, including a name string, UID, and optional namespace.
spec	To Describe the desired state of the object. Precise format of the object spec is different for every Kubernetes object, and contains nested fields specific to that object.
status	Provides read-only information about the current state of the object, is supplied and updated by the Kubernetes system. Like spec, format is vary by object type.

TABLE 4.1: Kubernetes Object Mandatory Fields

Work Out: To understand how Objects Management work in Kubernetes API, we will go through a quick exercise. Consider we have Kubernetes and its API service up and running. Consider we like to create an object of type, say, `FireBall`. Create a YAML as follows.

```
apiVersion: apiextensions.k8s.io/v1beta1
kind: CustomResourceDefinition
metadata:
  name: fireballs.api.sankholin.com
spec:
  group: api.sankholin.com
  version: v1
  scope: Namespaced
  names:
    plural: fireballs
    singular: fireball
    kind: FireBall
```

And create this `FireBall` type in Kube API:

```
kubectl create -f CRD_Fireball.yaml
kubectl get crd
kubectl api-versions
```

And now we can prescribe our `FireBall` type in YAML as follows. Notice of *kind* field.

```
apiVersion: "api.sankholin.com/v1"
kind: FireBall
metadata:
  name: my-fireball-object
spec:
  image: sankholin.com/fireball-container-image:latest
```

Finally, create an instance of `FireBall` custom Kubernetes object.

```
kubectl create -f my-fireball.yaml
kubectl get fireball
kubectl get fireball -o yaml
```

Note that `my-fireball-object` custom object contains custom field `spec.image`, which can be arbitrary JSON field and, for example, this will pull and deploy the containerised *Fire-Ball* application image from *Container Image Registry*. The use of *FireBall* nomenclature is intentional and the exact behaviour of what *FireBall* type might do is left it for reader interpretation – e.g. a cluster of Game engine. This is to show that how Kube API handle custom resources in a generic way through `CustomResourceDefinition`. Kube Objects are persistent entities (i.e. CRUD[4]) which represent the state of the cluster. In this exercise, we are using `CustomResourceDefinition` which is one of the built-in resource type of Kube API which purpose is to be able to create a *Custom Resources* inside Kube system. Table 4.2 summarise the commonly use Kubernetes built-in Resources[5] (or *kind*).

Kind	Description
Pod	The basic building block of Kubernetes – the smallest and simplest unit in the Kubernetes object model.
Service	Service is an abstraction which defines a logical set of Pods and a policy by which to access them - sometimes called a micro-service.
ReplicaSet	Next-generation Replication Controller. A ReplicaSet ensures that a specified number of pod replicas are running at any given time.
Deployment	Deployment controller provides declarative updates for Pods and ReplicaSets. You describe a desired state in a Deployment object, and the Deployment controller changes the actual state to the desired state at a controlled rate.
StatefulSets	StatefulSet is the workload API object used to manage stateful applications.
DaemonSet	A DaemonSet ensures that all (or some) Nodes run a copy of a Pod. As nodes are added to the cluster, Pods are added to them. As nodes are removed from the cluster, those Pods are garbage collected.
Job	A job creates one or more pods and ensures that a specified number of them successfully terminate.
CronJob	A Cron Job manages time based Jobs, run: Once or Repeatedly at a specified point in time.

TABLE 4.2: Kubernetes Built-in Resources

Note that Table 4.2 is just introductory Kubernetes resources that need to get started work on with the Kubernetes in most use cases. For the interest of thesis length, it is recommended to read Kubernetes documentation [14] for how these built-in Kubernetes resources work in specific to their functionality.

4.3 Tackling at Production Ready Setup

Kubernetes is a much more complex system to up and run with – it is quite a daunting task for setting up *Production Ready* Kubernetes deployment for user's own on-premise, bare-metal or Private Cloud infrastructure. It has steep learning curve. Its breadth of functionality grows daily, documentation is outdated quickly and is enough to overwhelm even the most advanced users. Every layer of Kubernetes have a couple of different ways to setup and options to pick. Nevertheless, Kubernetes composes of components for **Master** role and

[4]Create, Read, Update, Delete

[5]Resource as in RESTful architecture

components for the **Node** role. Table 4.3 and Table 4.4 summarise this. These components are fundamental executables for up and running the Kubernetes cluster.

Component	Description
kube-apiserver	exposes the API and the front-end for the Kubernetes control plane
kube-scheduler	monitor newly created pods and selects a node for them to run on
kube-controller-manager	control loop that watches the shared state of the cluster through the apiserver and makes changes attempting to move the current state towards the desired state
etcd	consistent and highly-available key value store backing for all cluster data

TABLE 4.3: Components for Kubernetes Master Role

Hard Way: For straight forward approach, we could spin up one node and install all the components mention in Table 4.3 to run as Kube Master role. And spin up one or more nodes to install all the components mention in Table 4.4 to run as Kube Node role. Since Kubernetes is written in GoLang[6], we could download the source – compile, build – or, download pre-compiled binary and run executables in foreground or, wrap into `systemd` daemon services. This would work in principle, at least. However so, this has to deal with more low level handling and, generally *"goto"* setup for a Kube developer.

Component	Description
kubelet	An agent that runs on each node in the cluster. It makes sure that containers are running in a pod.
kube-proxy	enables the Kubernetes service abstraction by maintaining network rules on the host and performing connection forwarding
container-runtime	the software that is responsible for running containers – supports several runtimes: **Docker**, **rkt**, **runc** and any Open Container Initiative (OCI) runtime specification implementation

TABLE 4.4: Components for Kubernetes Node Role

Streamline Toolkit: Up and running in *Hard Way* is not an ideal for *Production Ready* setup. For this purpose, Kube community has diverse range of streamline solution for many different setup that are specifics to Public Cloud Provider, Linux distro, and so on. Table 4.5 summarise the recommended toolkit selection for setting up Kube cluster.

Toolkit	Description
kubeadm	bootstrap a best-practice Kubernetes cluster in an easy, reasonably secure and extensible way. **Kubernetes Certified**.
Kubespray	install a Kubernetes cluster hosted on GCE, Azure, OpenStack, AWS, or Baremetal. Ansible.
kops	helps you create, destroy, upgrade and maintain production-grade, highly available, Kubernetes clusters from the command line for AWS

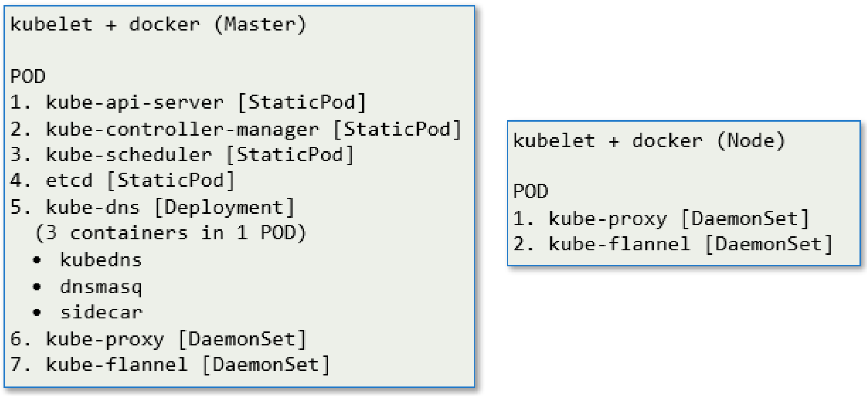
TABLE 4.5: Toolkits for Streamline Kubernetes Cluster Setup as of v1.10

[6]https://golang.org

The kubeadm is chosen to bootstrap the Kube cluster for the purpose of ATHENA deployment. However, also note that, Kubernetes documentation offer dozens of different ways[7] to setup Kube cluster. This research finds that **kubeadm** is the most appropriate toolkit for setting up *Production Ready* Kubernetes cluster. The justification for *why kubeadm* is because it is the Kubernetes **Certified Solution** as of v1.10, its abstraction to IaC automation tool and it can streamline the upgrade of the Kube cluster reliably well as shown below.

```
root@athena-kube-master1:~# kubeadm upgrade plan
[preflight] Running pre-flight checks.
[upgrade] Making sure the cluster is healthy:
(scrap)
Components that must be upgraded manually after you have upgraded the \
   control plane with 'kubeadm upgrade apply':
COMPONENT   CURRENT      AVAILABLE
Kubelet     4 x v1.9.4   v1.10.2

Upgrade to the latest stable version:
COMPONENT            CURRENT   AVAILABLE
API Server           v1.9.4    v1.10.2
Controller Manager   v1.9.4    v1.10.2
Scheduler            v1.9.4    v1.10.2
Kube Proxy           v1.9.4    v1.10.2
Kube DNS             1.14.7    1.14.7
Etcd                 3.1.11    3.1.11

You can now apply the upgrade by executing the following command:
        kubeadm upgrade apply v1.10.2
Note: Before you can perform this upgrade, you have to update kubeadm to v1.10.2.

---------------------------------------------------------------------

root@athena-kube-master1:~# apt-get install --only-upgrade kubeadm
(scrap)
root@athena-kube-master1:~# kubeadm upgrade apply v1.10.2
(scrap)
```

Figure 4.1 shows the result of Kubernetes components installed into Master and Node role respectively using kubeadm.



```
kubelet + docker (Master)

POD
1. kube-api-server [StaticPod]
2. kube-controller-manager [StaticPod]
3. kube-scheduler [StaticPod]
4. etcd [StaticPod]
5. kube-dns [Deployment]
  (3 containers in 1 POD)
  • kubedns
  • dnsmasq
  • sidecar
6. kube-proxy [DaemonSet]
7. kube-flannel [DaemonSet]
```

```
kubelet + docker (Node)

POD
1. kube-proxy [DaemonSet]
2. kube-flannel [DaemonSet]
```

FIGURE 4.1: Kubernetes components on Master and Node using kubeadm

[7] https://kubernetes.io/docs/setup/pick-right-solution/

It is interesting to observe that at the basic bootstrapping for **a Kube Node** means a machine installed with **just kubelet component and container runtime engine**. After kubelet and Docker container runtime engine installed, up and running as a `systemd` daemon service, a Pod can be deployed. Therefore, Figure 4.1 shows that api-server, controller-manager, scheduler, etcd components are running as Pods on a node with Master role; and similarly kube-proxy is running as Pod on both Master and Node roles.

Recall Table 4.2 that a Pod is the built-in resource type (a *kind*) and the basic unit of Kubernetes Object model. In this case, the kube-api-server, a GoLang application is containerised and deployed as a Pod. The `kube-apiserver.yaml` manifests as follows.

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    component: kube-apiserver
    tier: control-plane
  name: kube-apiserver
  namespace: kube-system
spec:
  containers:
  - command:
    - kube-apiserver
    image: gcr.io/google_containers/kube-apiserver-amd64:v1.10.2
    (scrap)
```

However, note that, kubeadm created the kube-apiserver component as a special Pod type called Static Pod. From k8s documentation[8], [14] states that *Static Pods are managed directly by kubelet daemon on a specific node, without the API server observing it. Kubelet automatically creates so-called mirror pod on the Kubernetes API server for each static pod, so the pods are visible there, but they cannot be controlled from the API server*. From this, we can also observe that kubelet can directly bootstrap a Pod, and it shares the Kubernetes objects management along with the kube-apiserver to sudden extent.

4.3.1 Add-ons

In addition to the Kubernetes Master and Node components (Tables 4.3, 4.4), we can also observe that there are kube-dns on Master role (installed by part of `kubeadm init`[9]) and, kube-flannel running on both Master and Node role. These components are called **Add-ons**[10] which extend the functionality of Kubernetes. Table 4.6 summarise components that are deployed as part of the bootstrapping Kubernetes cluster.

Add-ons	Description
kube-dns	The internal implementation for Kubernetes DNS service.
kube-flannel	Flannel is an overlay network provider that can be used with Kubernetes Pod network.

TABLE 4.6: Additional compulsory Kubernetes add-ons components

[8]https://kubernetes.io/docs/tasks/administer-cluster/static-pod/
[9]https://kubernetes.io/docs/reference/setup-tools/kubeadm/kubeadm-init/
[10]https://kubernetes.io/docs/concepts/cluster-administration/addons/

To bootstrap the full-fledge *Production Ready* Kubernetes cluster, there are compulsory add-ons such as like these. Note though that, Kubernetes documentation [14] offer dozens of Add-ons; some of which are competing technologies and alternatives for the particular functionality e.g. to use **Weave Net** instead of **Flannel** for Pod Networking.

4.3.2 Pod Networking

From Table 4.6, there is a notion of **Overlay Network**. In Kubernetes, the concept of Pod is to encapsulates containers. A Kube Pod object holds one or more containers and, introduces **IP-per-Pod** for network model. It implies IP addressing at the Pod scope. Therefore, containers within a Pod share their network namespaces including their IP address. The Pod networking requirement [14] states that nodes and containers can communicate each others without Network Address Translation (NAT). Therefore, the IP that a container sees itself as is the same IP that others see it as. In a nutshell, Kubernetes network model imposes barring usage of intentional network segmentation policies such as NAT. Kubernetes documentation offers dozens of different ways[11] for implementing this networking requirement – Overlay Network is one of them. The implementation usually offer as *Add-ons*, however, it is a compulsory step. Medium articles by google-cloud[12], ApsOps[13] illustrated details on how Overlay Network is used for Kubernetes Pod networking requirement.

For the context of *compute auto-scaling research* presenting in this thesis, the further investigation on performance comparison of Pod Networking with competing technologies left out for future work.

4.3.3 Service

Service Proxies: In Kubernetes, Pods are ephemeral and mortal. This is due to the fact that the Kube cluster can replicate Pod (destroy and re-create new) for dynamically scale up or down, self-healing and self-managing purposes. As the Pod get destroyed or recycled, the Pod IP address may change. This is not the desire effect for the application developer to keep track of these IP addresses. Service is an network abstraction layer to retain the consistent endpoint within the cluster. Service can be also seen as frontend service proxies for other Kubernetes objects such as Pod, Deployment, ReplicaSet, etc. For the implementation, kube-proxy is responsible for creating a form of Virtual IP in 3 different proxy modes – userspace, iptables, IPVS – for Services of type other than ExternalName. For publishing services, it offers 4 different types as summarise in Table 4.7.

Type	Description
ClusterIP	Exposes the service on a cluster-internal IP. **Default type**.
NodePort	Exposes the service on each Node's IP at a static port.
LoadBalancer	Exposes the service externally using a cloud provider's load balancer.
ExternalName	Maps the service to the contents of the externalName field (e.g. foo.bar.example.com), by returning a CNAME record with its value.

TABLE 4.7: Kubernetes service types for publishing service endpoints

[11] https://kubernetes.io/docs/concepts/cluster-administration/networking/
[12] https://medium.com/google-cloud/understanding-kubernetes-networking-pods-7117dd28727
[13] https://medium.com/@ApsOps/an-illustrated-guide-to-kubernetes-networking-part-2-13fdc6c4e24c

Service Discovery: For Service Discovery purpose, Kubernetes runs internal DNS. This comes in as Add-ons, however, compulsory. Kubeadm deployed kube-dns[14] Pod for this purpose. For example, the following `athena-ui.yaml` service manifest creates a service endpoint for the athena-ui.

```
apiVersion: v1
kind: Service
metadata:
  labels:
    app: athena-ui
  name: athena-ui
spec:
  ports:
    - port: 80
  selector:
    app: athena-ui
```

The following screenshot shows ATHENA's components and their service proxies setup.

```
root@athena-kube-master1:~# kubectl get svc
NAME                TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)   AGE
athena-analytics    ClusterIP   10.103.197.212   <none>        80/TCP    61d
athena-api-server   ClusterIP   10.101.72.108    <none>        80/TCP    54d
athena-ui           ClusterIP   10.103.117.69    <none>        80/TCP    82d
```

In this example, we can say that, the `athena-ui.yaml` service manifest created a DNS
`athena-ui.default.svc.cluster.local`
that resolve to the Service Endpoint IP address `10.103.117.69`.

Exposing Service Endpoint: We can reach to the service's **ClusterIP**, from all nodes within cluster, e.g:

```
root@athena-kube-master1:~# curl -s --user (scrap):(scrap) 10.101.72.108/api/version
{"version":"1.0.1-SNAPSHOT"}
root@athena-kube-n1:~# curl -s --user (scrap):(scrap) 10.101.72.108/api/version
{"version":"1.0.1-SNAPSHOT"}
root@athena-kube-n2:~# curl -s --user (scrap):(scrap) 10.101.72.108/api/version
{"version":"1.0.1-SNAPSHOT"}
root@athena-kube-n3:~# curl -s --user (scrap):(scrap) 10.101.72.108/api/version
{"version":"1.0.1-SNAPSHOT"}
```

Kubernetes offers few different possibilities – **NodePort, LoadBalancer, HostNetwork, HostPort, Ingress** – to expose the service endpoint to outside world, i.e. Public network. For production setup, cloud provider's LoadBalancer or Ingress[15] should be used, however, on NeCTAR, every VM instances get Public IP address. In this case, setting up a simple HAProxy **Layer-4/Layer-7** routing deliver the similar effect.

[14] https://kubernetes.io/docs/tasks/administer-cluster/dns-custom-nameservers/
[15] https://kubernetes.io/docs/concepts/services-networking/ingress/

```
root@athena-kube-n1:~# ip a|grep eth0|grep inet
    inet 115.146.85.147/22 brd 115.146.87.255 scope global eth0

root@athena-kube-n1:~# tail -6 /etc/haproxy/haproxy.cfg
frontend loadbalancer
        bind *:80
        default_backend athena

backend athena
        server api1 10.101.72.108
```

4.3.4 Volume and Storage

Kubernetes also offer dozens of different ways[16] for setting up volume and storage. Likewise in Docker container, volume in Kubernetes are ephemeral i.e. data will be lost if the Pod restart has happened. Table 4.8 list some basic volume that can be used by Kubernetes Pod, StatefulSets, etc.

Type	Description
emptyDir	An emptyDir volume is first created when a Pod is assigned to a Node, and exists as long as that Pod is running on that node.
hostPath	A hostPath volume mounts a file or directory from the host node's filesystem into your Pod.
nfs	An nfs volume allows an existing NFS (Network File System) share to be mounted into your Pod.

TABLE 4.8: Kubernetes basic volume storage

The production setup should explore a better solution space in cluster file system such as setting up generic GlusterFS, CephFS, or cloud provider specific OpenStack Cinder, AWS ElasticBlockStore or a more performance iSCSI, Fibre Channel, and parallel distributed clustered file system such as IBM GPFS or LustreFS.

For the context of *compute auto-scaling research* presenting in this thesis, the further investigation on scaling of volume and data storage left out for future work.

4.4 Instrumentation and Metrics

Kubernetes core components (Tables 4.3, 4.4) are instrumented and exposed at `/metrics`. By default, metrics are in Prometheus format[17]. When making HTTP GET request to `/metrics` on the component being monitored, it expects to return a set of line delimited metrics. The following is an example of kube-apiserver component metrics.

```
curl -s 127.0.0.1:8001/metrics | grep process_cpu_seconds_total
# HELP process_cpu_seconds_total Total user and system CPU time spent in seconds.
# TYPE process_cpu_seconds_total counter
process_cpu_seconds_total 167305.1
```

[16] https://kubernetes.io/docs/concepts/storage/volumes/

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

Among these core components, kubelet is one of the most important components in Kubernetes. Kubelet makes sure containers run inside Pod, runs container probe and, reports the status back to the kube-apiserver. Kubelet also comes with an embedded cAdvisor[18] instance which collects, aggregates, processes and exports metrics such as CPU, memory and network usage of running containers. Kubelet exposes cAdvisor metrics at `/metrics/cadvisor`.

```
curl -s 127.0.0.1:10255/metrics/cadvisor|grep container_cpu_usage_seconds_total
container_cpu_usage_seconds_total{container_name="athena-worker", (scrap)"} 218.975
```

Kubelet Summary API also expose aggregated node and Pods metrics as follows.

```
curl -s http://localhost:10255/stats/summary | less
{ "node": {
   "nodeName": "athena-kube-n3",
   "systemContainers": [
    {
     "name": "runtime",
     "startTime": "2018-06-07T06:51:55Z",
     "cpu": {
      "time": "2018-06-09T13:47:57Z",
      "usageNanoCores": 39373255,
      "usageCoreNanoSeconds": 79051470783486
     },
     "memory": {
      "time": "2018-06-09T13:47:57Z",
      "usageBytes": 5535948800,
      "workingSetBytes": 1090809856,
      "rssBytes": 103976960,
      "pageFaults": 18562442,
      "majorPageFaults": 558
     },
    (scrap)
```

Before Kubernetes v1.9, the popular stack is Heapster + InfluxDB for consuming these metrics and storing them as timeseries data. However, as of v1.10, Heapster is deprecated[19] in favour for an overhaul with Metrics API[20] at Kubernetes Instrumentation and monitoring architecture[21]. And the Metrics Server is the official[22] successor to replace Heapster. Prometheus is used for the monitoring and scraping various Kubernetes components metrics. In particular, **Prometheus Operator**[23] **+ kube-prometheus**[24] stack is chosen empirically based on different trials. Prometheus is primarily a pull style monitoring platform. The justification for *why Prometheus* is mainly based on this metrics polling approach for better performance outcome. This essentially does not impose stress and load on the monitored application. Furthermore, Prometheus offer many third-party metrics exporters[25] such as the JMX exporter which can export from a wide variety of JVM-based applications (e.g. ActiveMQ, Kafka) and, timing and gauging HTTP Requests/Responses (e.g. Apache, Nginx,

[18] https://github.com/google/cadvisor
[19] https://github.com/kubernetes/heapster/blob/master/docs/deprecation.md
[20] https://brancz.com/2018/01/05/prometheus-vs-heapster-vs-kubernetes-metrics-apis/
[21] https://github.com/kubernetes/community/blob/master/contributors/design-proposals/instrumentation/monitoring_architecture.md
[22] https://kubernetes.io/docs/tasks/debug-application-cluster/core-metrics-pipeline/
[23] https://github.com/coreos/prometheus-operator
[24] https://github.com/coreos/prometheus-operator/tree/master/contrib/kube-prometheus
[25] https://prometheus.io/docs/instrumenting/exporters/

HAProxy) metrics and workload.

Prometheus Operator: It[26] creates, configures, and manages Prometheus monitoring instances and, introduces "monitoring" namespace and additional resources Alertmanager and ServiceMonitor to Kubernetes API through `CustomResourceDefinition` (section 4.2). The ServiceMonitor uses Kube's Label and Selector to intercept Kubernetes Services (section 4.3.3) and their exposed Prometheus format metrics (i.e. `/metrics`) in evasive way.

```
root@athena-kube-master1:~# kubectl get crd
NAME                                    AGE
alertmanagers.monitoring.coreos.com     20d
prometheuses.monitoring.coreos.com      20d
servicemonitors.monitoring.coreos.com   20d
root@athena-kube-master1:~# kubectl get namespaces
NAME          STATUS    AGE
monitoring    Active    20d
root@athena-kube-master1:~# kubectl api-versions|grep monitoring
monitoring.coreos.com/v1
```

Monitoring Setup: The Metrics Server (Heapster replacement) setup is straight forward. We can create it from the provided manifests as follows.

```
root@athena-kube-master1:~# git clone \
                https://github.com/kubernetes-incubator/metrics-server.git
(scrap)
root@athena-kube-master1:~# cd metrics-server
root@athena-kube-master1:~/metrics-server# kubectl create -f deploy/1.8+/
(scrap)
```

Once Metrics Server is up and running, we can observe the cluster wide metrics and monitoring that is scraped from Kubelet Summary API.

```
root@athena-kube-master1:~# kubectl api-versions|grep metrics
metrics.k8s.io/v1beta1
root@athena-kube-master1:~# kubectl top node
NAME                   CPU(cores)   CPU%     MEMORY(bytes)   MEMORY%
athena-kube-master1    216m         1%       928Mi           1%
athena-kube-n1         71m          0%       1011Mi          2%
athena-kube-n2         64m          0%       5729Mi          11%
athena-kube-n3         109m         0%       2786Mi          5%
root@athena-kube-master1:~# kubectl top pod
NAME                                  CPU(cores)   MEMORY(bytes)
athena-analytics-5ddd588cd6-mv7ff     0m           705Mi
athena-api-server-dc74db874-mk8fp     1m           5462Mi
athena-ui-7f77f68b86-dxf8s            0m           3Mi
athena-worker-79fb4ff6bb-6dns4        1m           1179Mi
athena-worker-79fb4ff6bb-hkjml        1m           1212Mi
---
kubectl get --raw "/apis/metrics.k8s.io/v1beta1/nodes" | python -m json.tool
(scrap)
kubectl get --raw "/apis/metrics.k8s.io/v1beta1/pods" | python -m json.tool
(scrap)
```

[26]https://coreos.com/operators/prometheus/docs/latest/user-guides/getting-started.html

The prometheus operator **Clustering Monitoring** documentation[27] provides Kubernetes manifests (i.e. YAMLs) for stetting up the **Prometheus Operator + kube-prometheus monitoring stack**. However, since the stack comprises of multiple components, it is better to use **Helm** for the setup.

```
root@athena-kube-master1:~# helm repo add coreos \
         https://s3-eu-west-1.amazonaws.com/coreos-charts/stable/
"coreos" has been added to your repositories
root@athena-kube-master1:~# helm repo list
NAME          URL
stable        https://kubernetes-charts.storage.googleapis.com
local         http://127.0.0.1:8879/charts
coreos        https://s3-eu-west-1.amazonaws.com/coreos-charts/stable/
root@athena-kube-master1:~# helm install coreos/prometheus-operator \
        --name prometheus-operator --namespace monitoring
NAME:   prometheus-operator
(scrap)
```

Helm: Helm.sh[28] is the package manager for Kubernetes. It similar to what APT package management tool to Ubuntu/Debian.

4.5 Horizontal Pod Autoscaler

According to [14], the Horizontal Pod Autoscaler (HPA) dynamically adjusts the number of Pod replicas in a Deployment based on observed CPU utilisation. It is implemented as a Kubernetes API resource and a controller. To walkthrough, the ATHENA worker deployment manifest as follows.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: athena-worker
spec:
  selector:
    matchLabels:
      app: athena-worker
  replicas: 2
  template:
    spec:
      containers:
      - name: athena-worker
        image: athena-nexus.eresearch.unimelb.edu.au/athena/athena-worker:latest
(scrap)
```

To deploy this:

```
kubectl create -f athena-worker.yaml
```

To scale the deployment:

```
kubectl scale deployment athena-worker --replicas=5
```

[27] https://github.com/coreos/prometheus-operator/blob/master/Documentation/user-guides/cluster-monitoring.md
[28] https://helm.sh

Recall (Chapter 3) that this is the similar scaling we did in the plain vanilla Docker-Compose and Swarm stack deployment.

```
docker-compose -f docker-compose.yaml -f staging.yaml  --project-name=stg \
   --scale worker=2

docker service scale athena_worker=5
```

However, with Kubernetes HPA, we can now **autoscale** the ATHENA worker deployment as follows.

```
kubectl autoscale deployment athena-worker --min=1 --max=6 --cpu-percent=80
```

This command[29] tells that auto scale a deployment "athena-worker", with the number of Pods between 1 and 6, target CPU utilization at 80%.

```
root@athena-kube-master1:~# kubectl get hpa
NAME            REFERENCE                  TARGETS    MINPODS    MAXPODS    REPLICAS
athena-worker   Deployment/athena-worker   0%/80%     1          6          1
php-apache      Deployment/php-apache      0%/50%     1          10         1
```

Instead of command line, we can also describe the HPA manifest as follow.

```
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  name: athena-worker
  namespace: default
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: athena-worker
  minReplicas: 1
  maxReplicas: 6
  targetCPUUtilizationPercentage: 80
```

And, as usual, create the HPA Kubernetes resource object.

```
kubectl create -f athena-worker-hpa.yaml
```

The Horizontal Pod Autoscaler feature was introduced in Kubernetes v1.1. The first version of HPA scale the Pod based on observed CPU utilization and memory usage. In Kubernetes 1.6, the Custom Metrics API was introduced that enables HPA access to arbitrary metrics through REST API. In Kubernetes 1.7, the API server aggregation layer allows third party applications to extend the Kubernetes API by registering themselves as API *Add-ons*. Note that, the concept about API server aggregation[30] is similar to `CustomResourceDefinition` (section 4.2), but offer more flexible implementation option.

Figure 4.2 depict the full orchestration of Prometheus metrics monitoring stack with HPA. In Core Metrics pipeline, Metrics Server scrape node and container metrics from Kubelet cAdvisor. In Monitoring pipeline, Prometheus operator collect metrics through its ServiceMonitor and store in Prometheus. And the Grafana dashboard can visual these

[29] https://kubernetes.io/docs/reference/kubectl/kubectl/kubectl_autoscale.md
[30] https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/

monitoring metrics. Furthermore, with Prometheus Adapter (an API server aggregation implementation), the Prometheus metrics can be consumed by HPA.

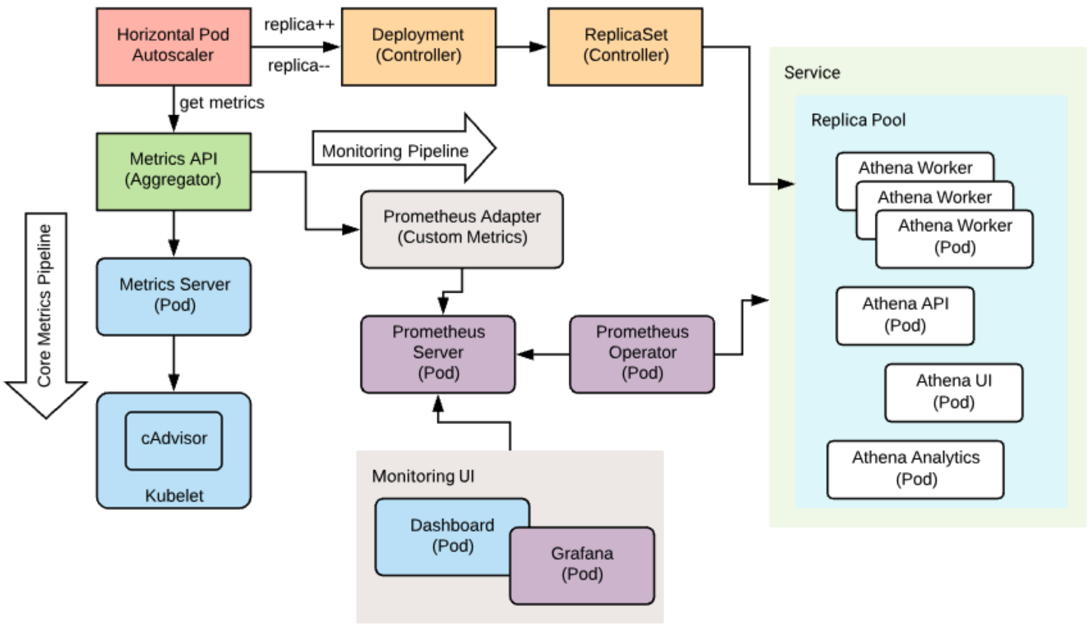


FIGURE 4.2: Kubernetes Cluster: HPA and Monitoring for ATHENA

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 Spring Boot application like 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 Actuator[31] and Micrometer[32].

```
curl -s https://athena-dev.eresearch.unimelb.edu.au/prometheus
# HELP jvm_memory_used_bytes The amount of used memory
# TYPE jvm_memory_used_bytes gauge
jvm_memory_used_bytes{area="heap",id="PS Survivor Space",} 2.3983848E7
jvm_memory_used_bytes{area="heap",id="PS Old Gen",} 1.11927144E8
jvm_memory_used_bytes{area="heap",id="PS Eden Space",} 7.72185792E8
(scrap)
# HELP http_server_requests_duration_seconds Timer of servlet request
# TYPE http_server_requests_duration_seconds summary
http_server_requests_duration_seconds_count{method="GET",uri="/login",} 101.0
http_server_requests_duration_seconds_sum{method="GET",uri="/login",} 0.8731 218.975
(scrap)
```

[31]https://docs.spring.io/spring-boot/docs/current/reference/html/production-ready.html
[32]http://micrometer.io

HPA Autoscaling Algorithm: Based on [14], the HPA implementation of autoscaling algorithm works as follow.

1. Implemented as a **Control Loop** - 30 seconds by default, configurable.
2. Periodically queries Pods to collects their CPU utilization
3. Compares the arithmetic mean of the Pods' CPU utilization with the target.
4. Adjusts the replicas of the Scale if needed to match the target, preserving condition:
   - MinReplicas <= Replicas <= MaxReplicas
   - CPUUtilization (C) = recent CPU usage of a Pod (average across the last 1 minute) / CPU requested by the Pod (*spec.containers[].resources.requests.cpu*[33])
   - TargetNumOfPods = ceil(sum(CurrentPodsCPUUtilization) / TargetCPUUtilizationPercentage (T))

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

5. Scale-up can only happen if there was no rescaling within the last 3 minutes. Due to temporarily CPU fluctuation during start/stop.
6. Scale-down will wait for 5 minutes from the last rescaling. Due to temporarily CPU fluctuation during start/stop.
7. Any scaling will only be made if: avg(CurrentPodsConsumption) / TargetCPUUtilizationPercentage drops below 0.9 or increases above 1.1 (10% tolerance).

[14] states that algorithm approach has two benefits:

- Autoscaler works in a conservative way in such that increasing rapidly the number of Pods is important when user load is deteced. However, lowering the number of pods is not that urgent.
- Autoscaler avoids thrashing, i.e. it prevents rapid execution of conflicting decision if the load is not stable.

[33] `https://kubernetes.io/docs/tasks/configure-pod-container/assign-cpu-resource/`

4.6 Result and Finding

Though Kubernetes is complex system to master thoroughly, when in operation, it is found that a very robust and flexible system. There also observable notice in overhead such as service proxies network layer.

During the experimental runs, it is noticed that the auto-scaler doesn't react immediately to usage spikes. By default the metrics 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 is also found that auto-scaling based on observed CPU utilisation alone is not sufficient enough for ATHENA worker, due to the dynamic of how the simulation would run: *simulation alone* or *simulation + optimisation algorithm*. CPU utilisation scaling works better with optimisation algorithm on. However, simulation run alone with Monte Carlo job executed really quick. Therefore, it missed the scaling observation period delay time and consequently, it missed the HPA replication trigger cycle and there was only 1 worker Pod instance working on jobs. This can be tuned accordingly to the target threshold and CPU utilisation feedback control loop, however, it is still not a perfect solution. In addition of CPU utilisation, observing job queue metrics should be implemented for a better work load balancing.

Figure 4.3 depicts the architectural view of ATHENA deployment with Kubernetes cluster. Docker Registry is a container image registry hosted using Nexus repository manager.

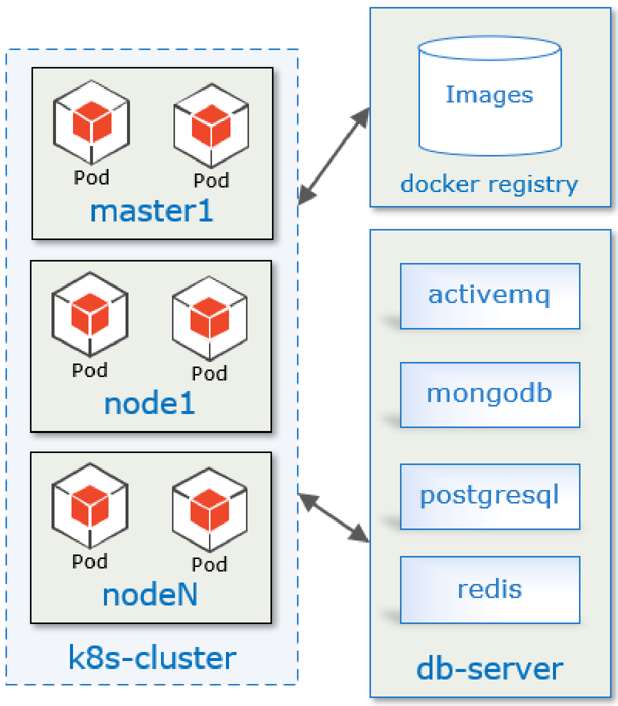
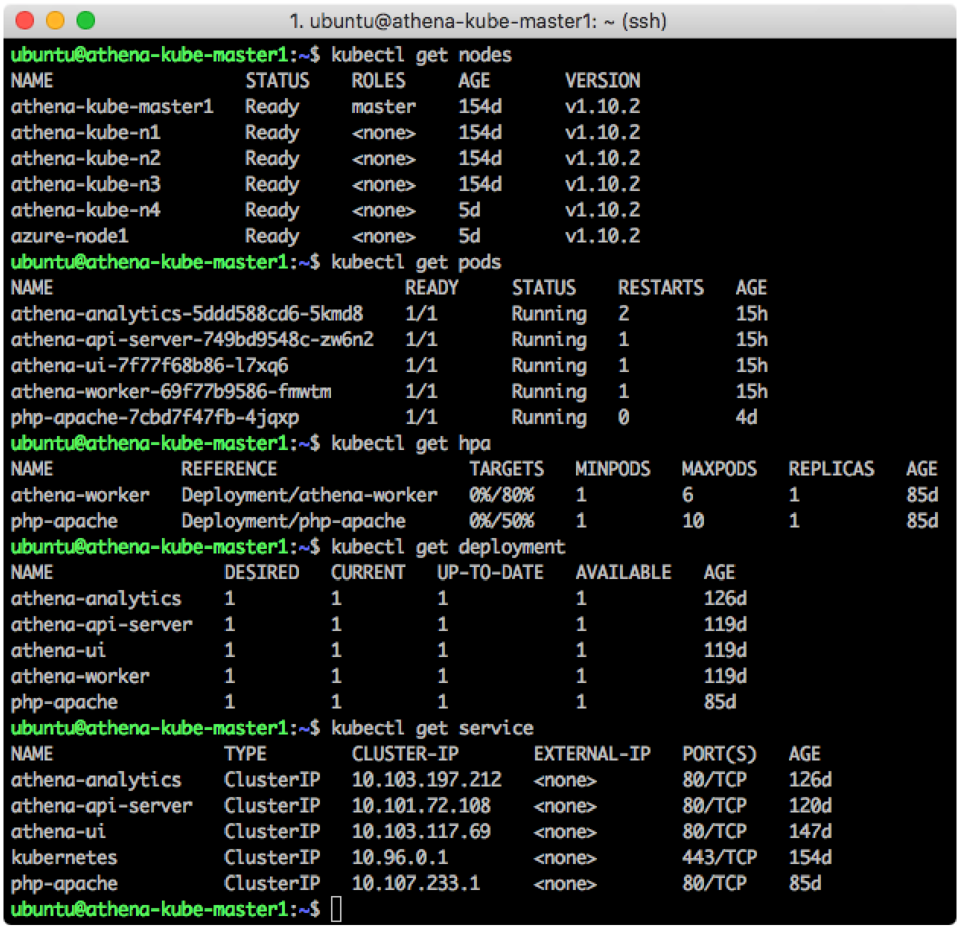


FIGURE 4.3: Kubernetes Cluster for ATHENA Deployment: Architecture

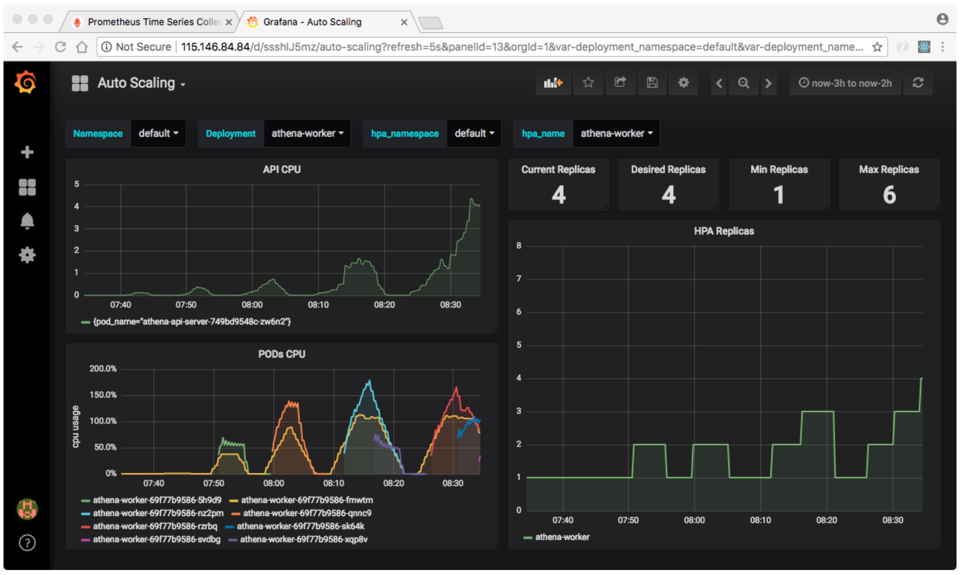
Figure 4.4 shows the terminal screenshot of Kubernetes cluster up and running the ATHENA software stack. And Figure 4.5 shows the screenshot of Grafana[34] monitoring dashboard for observing auto-scaling of ATHENA worker – where the Replica trend shows the discrete steps of worker Pods scaling up/down, during a simulation run.

[34] https://grafana.com



```
1. ubuntu@athena-kube-master1: ~ (ssh)
ubuntu@athena-kube-master1:~$ kubectl get nodes
NAME                 STATUS    ROLES     AGE       VERSION
athena-kube-master1  Ready     master    154d      v1.10.2
athena-kube-n1       Ready     <none>    154d      v1.10.2
athena-kube-n2       Ready     <none>    154d      v1.10.2
athena-kube-n3       Ready     <none>    154d      v1.10.2
athena-kube-n4       Ready     <none>    5d        v1.10.2
azure-node1          Ready     <none>    5d        v1.10.2
ubuntu@athena-kube-master1:~$ kubectl get pods
NAME                                  READY     STATUS    RESTARTS   AGE
athena-analytics-5ddd588cd6-5kmd8     1/1       Running   2          15h
athena-api-server-749bd9548c-zw6n2    1/1       Running   1          15h
athena-ui-7f77f68b86-l7xq6            1/1       Running   1          15h
athena-worker-69f77b9586-fmwtm        1/1       Running   1          15h
php-apache-7cbd7f47fb-4jqxp           1/1       Running   0          4d
ubuntu@athena-kube-master1:~$ kubectl get hpa
NAME            REFERENCE                  TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
athena-worker   Deployment/athena-worker   0%/80%    1         6         1          85d
php-apache      Deployment/php-apache      0%/50%    1         10        1          85d
ubuntu@athena-kube-master1:~$ kubectl get deployment
NAME                DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
athena-analytics    1         1         1            1           126d
athena-api-server   1         1         1            1           119d
athena-ui           1         1         1            1           119d
athena-worker       1         1         1            1           119d
php-apache          1         1         1            1           85d
ubuntu@athena-kube-master1:~$ kubectl get service
NAME                TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)   AGE
athena-analytics    ClusterIP   10.103.197.212   <none>        80/TCP    126d
athena-api-server   ClusterIP   10.101.72.108    <none>        80/TCP    120d
athena-ui           ClusterIP   10.103.117.69    <none>        80/TCP    147d
kubernetes          ClusterIP   10.96.0.1        <none>        443/TCP   154d
php-apache          ClusterIP   10.107.233.1     <none>        80/TCP    85d
ubuntu@athena-kube-master1:~$ 
```

FIGURE 4.4: Kubernetes Cluster for ATHENA Deployment: Terminal



Not Secure | 115.146.84.84/d/ssshlJ5mz/auto-scaling?refresh=5s&panelId=13&orgId=1&var-deployment_namespace=default&var-deployment_name...

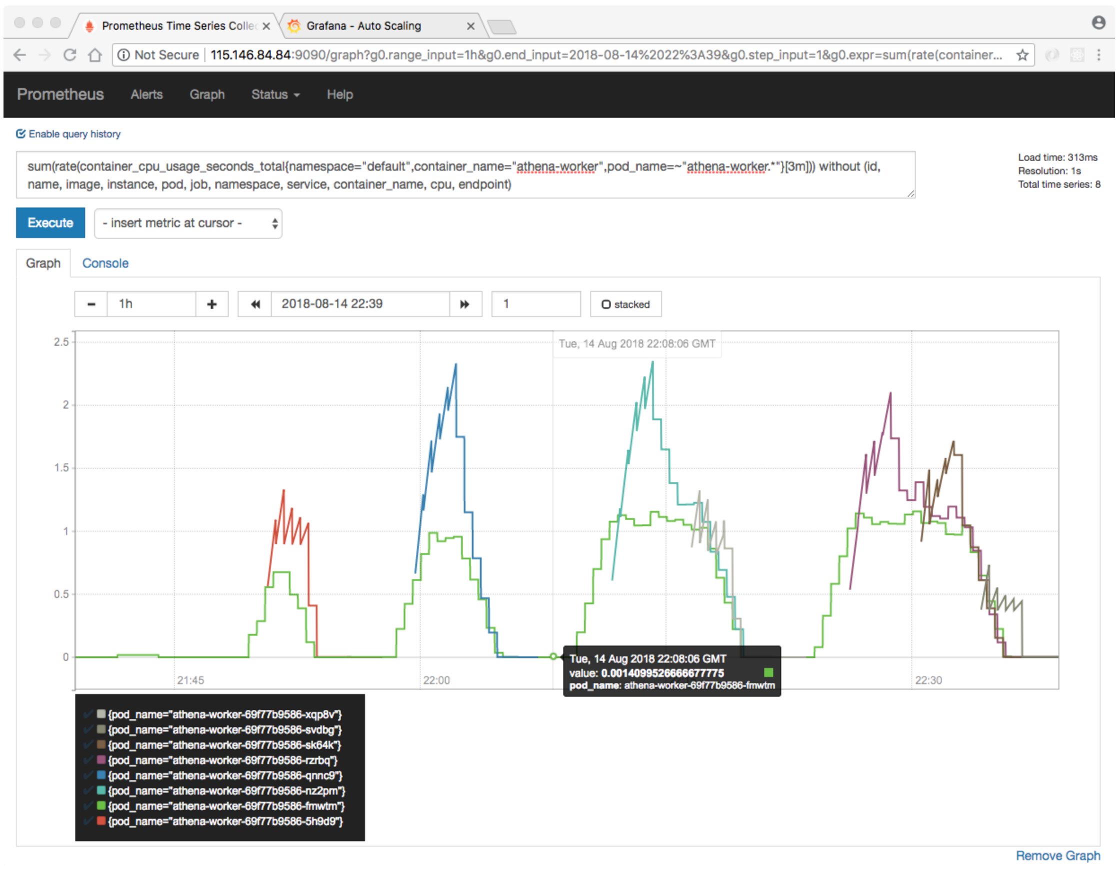
Namespace default · Deployment athena-worker · hpa_namespace default · hpa_name athena-worker

Current Replicas 4 · Desired Replicas 4 · Min Replicas 1 · Max Replicas 6

Prometheus Time Series Collec
Grafana - Auto Scaling
Auto Scaling
now-3h to now-2h
API CPU
5
4
3
2
1
0
07:40
07:50
08:00
08:10
08:20
08:30
{pod_name="athena-api-server-749bd9548c-zw6n2"}
PODs CPU
cpu usage
200.0%
150.0%
100.0%
50.0%
0%
athena-worker-69f77b9586-5h9d9
athena-worker-69f77b9586-fmwtm
athena-worker-69f77b9586-nz2pm
athena-worker-69f77b9586-qnnc9
athena-worker-69f77b9586-rzrbq
athena-worker-69f77b9586-sk64k
athena-worker-69f77b9586-svdbg
athena-worker-69f77b9586-xqp8v
HPA Replicas
8
7
6
5
4
3
2
1
0
athena-worker

FIGURE 4.5: Kubernetes Cluster for ATHENA Deployment: Grafana

Figure 4.6 shows the screenshot of the Prometheus dashboard along with the Prometheus query and result graphing from Prometheus time-series database.


Prometheus Time Series Collec
Grafana - Auto Scaling
Not Secure
115.146.84.84:9090/graph?g0.range_input=1h&g0.end_input=2018-08-14%2022%3A39&g0.step_input=1&g0.expr=sum(rate(container...
Prometheus
Alerts
Graph
Status
Help
Enable query history
Execute
- insert metric at cursor -
Graph
Console
1h
2018-08-14 22:39
1
stacked
2.5
2
1.5
1
0.5
0
21:45
22:00
22:30
Tue, 14 Aug 2018 22:08:06 GMT
Remove Graph

```
sum(rate(container_cpu_usage_seconds_total{namespace="default",container_name="athena-worker",pod_name=~"athena-worker.*"}[3m])) without (id,
name, image, instance, pod, job, namespace, service, container_name, cpu, endpoint)
```

Load time: 313ms
Resolution: 1s
Total time series: 8

Tue, 14 Aug 2018 22:08:06 GMT
value: 0.0014099526666677775
pod_name: athena-worker-69f77b9586-fmwtm

```
{pod_name="athena-worker-69f77b9586-xqp8v"}
{pod_name="athena-worker-69f77b9586-svdbg"}
{pod_name="athena-worker-69f77b9586-sk64k"}
{pod_name="athena-worker-69f77b9586-rzrbq"}
{pod_name="athena-worker-69f77b9586-qnnc9"}
{pod_name="athena-worker-69f77b9586-nz2pm"}
{pod_name="athena-worker-69f77b9586-fmwtm"}
{pod_name="athena-worker-69f77b9586-5h9d9"}
```

FIGURE 4.6: Kubernetes Cluster for ATHENA Deployment: Prometheus

The following is an example of Prometheus query that summing over `container_cpu_usage_seconds_total` metrics at rate of 5 minutes averaging on all ATHENA worker Pods.

```
sum(rate(container_cpu_usage_seconds_total{namespace="default",\
        container_name="athena-worker",pod_name=~"athena-worker.*"}[5m])) \
        without (id, name, image, instance, pod, job, namespace, service, \
        container_name, cpu, endpoint)
```

The following is an example of Prometheus query that find the maximum of `kube_hpa_status_current_replicas` metrics – the current max replica created by the HPA controller.

```
max(kube_hpa_status_current_replicas{hpa="athena-worker",namespace="default"}) \
        without (instance, pod)
```

5 Related Studies

The concept of *Auto-Scaling system* is not new. The early research dated back to IBM's **MAPE-K** architecture discussion in Autonomic Computing – computing systems that can manage themselves given high-level objectives from administrators [12]. The paper [12] identified the key autonomic elements – Monitor, Analyse, Plan, Execute, Knowledge – the reference model for autonomic control loop for achieving the self-managed – self-configuration, self-optimisation, self-healing, self-protection – autonomic computing system.

In [17], it further identified the challenges of auto-scaler in each of the MAPE phases, and created taxonomy for auto-scaling web application in clouds. The [17] presented very broad spectrum of auto-scaling surveys – some of the approaches such as scaling indicator and different type of metrics, resource estimation with rule and threshold based approaches, multi-tier application scaling and, container-based auto-scaler in Kubernetes are investigated in this thesis.

Although not specific about auto-scaling, [18] is another good pointer on survey and taxonomy of resource optimisation for executing Bag-of-Task applications on Public Clouds – which give some indication on tackling with ATHENA BoT worker remote job executions. The [19] is another influential article which discussed dynamic provision with Aneka that can be able to burst out to Public Cloud from on-premise Private Cloud (therefore, Hybrid Cloud) to off load the deadline-driven data intensive application and Big data processing. Some of which, for example, Kubernetes Pod Networking with Overlay Network can go across multi Clouds; therefore, it could configure *Quality of Service for Pods*[1] and design burstable, auto-scalable system for compute or data processing systems.

Auto-scaling techniques discussed in [20] gives future exploration on auto-scaler algorithm design. According to [20], Kubernetes HPA auto-scaler can classify into reactive approach where HPA implementation is a threshold-based reactive controller to automatically adjust the required resources to the application demand i.e. Control Loop or Control Theory as stated in literature. Other approaches presented are Threshold-based rules which are simple and straightforward to set scale up and scale down thresholds based on observed CPU load or average responses. Reinforcement Learning (RL) is Machine Learning based, a type of automatic decision-making approach with no priori knowledge or system model, but it may take the method to converge to an optimal policy. Queuing Theory is highly Mathematical influence approach that require to model VM, containers or application tier as a queue of requests. Time-series analysis is proactive auto-scaling approach that use the past history of a time-series to predict future values - which is very promising technique to explore and investigate with the Prometheus time-series database stack that already used in this thesis research.

[1]`https://kubernetes.io/docs/tasks/configure-pod-container/quality-service-pod/`

6 Conclusion

6.1 Conclusion

In order to produce a reliable auto-scaling system, it is important to understand the target application well, in this case, ATHENA system as discussed in Chapter 2. It is important step to identify which part of system component can be scaled such as ATHENA Worker. It also requires to consolidate the Stateless and Stateful of the system, so that dynamic scaling is possible, as discussed in Chapter 3. It is also crucial to get insight of the foundation platform, Kubernetes as discussed in Chapter 4, to configure properly, get up and running the *Production Ready* cluster.

Infrastructure scaling on cloud platform required provider specific API and technology. The containerisation deployment with Docker Swarm and Kubernetes technologies can scale across multi-clouds independently, without the need of Cloud Provider specific API.

Auto-scaling system can not meet SLO and QoS 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 traffic bursts and user load. With new **Custom Metrics API** feature introduced in Kubernetes, we can extend the API and expose ATHENA's metrics to Prometheus, then can be consumed by HPA controller and auto-scaled. For enterprise integration system like ATHENA, the auto scaling could be triggered by the ActiveMQ job queue length exceeding some empirical threshold, therefore, **Threshold-based Reactive approach** with **Control Theory** auto-scaling technique can be explored.

Effective monitoring and metrics scraping is the heart of the pipeline for auto-scaler to work reliably well. Auto-scaling with Kubernetes is non trivial task to setup for *Production Ready* in Private Cloud, such as NeCTAR Research Cloud.

6.2 Future Work

For the future work, instrumenting ATHENA with Prometheus and exposing the right metrics can fine tune the auto-scaling to better handle bursts, high availability and QoS. We can also explore and design auto-scaling algorithm with different techniques such as Time-series Analysis for Kubernetes, as well as implementing Docker Swarm based auto-scaler.

We can also explore burstable infrastructure level scaling with OpenStack for NeCTAR and other Public Cloud Providers. Kubernetes has introduced Cloud Manager controller object to explore in this direction. For the infrastructure scaling, we can observe the metrics already produced by Kubernetes and Prometheus; and auto-scale nodes based on these metrics.

There are potential comparison and benchmarking studies that can be done within Kubernetes eco-system, such as benchmarking different Pod Networking options, comparison for reliable cluster file systems for Kubernetes, and so on. Highly Available (HA) Kubernetes cluster involves multiple Kube masters, therefore, multi-cells cluster; which is interesting space to explore as well.

As for ATHENA, the API backend need to decompose the aggregator component to become a better stateless application, so that it can be scaled well. And to explore ATHENA database stack to deploy it onto Kubernetes cluster using StatefulSet and explore clustered file system store, e.g. Cinder, CephFS, for dynamic volume provision.

7 Appendix

Source Code Listing

https://dstgrp.visualstudio.com/Daenerys/DaenerysTeam/_git/deployment
https://github.com/victorskl/auto-scaling-thesis
https://github.com/victorskl/kube-node
https://github.com/victorskl/docker-prometheus-grafana
https://github.com/victorskl/docker-private-registry
https://github.com/victorskl/docker-nexus-oss
https://github.com/victorskl/docker-mongodb
https://github.com/victorskl/docker-activemq
https://github.com/victorskl/docker-postgresql
https://github.com/victorskl/docker-redis
https://github.com/victorskl/docker-java-beatapp
https://github.com/victorskl/docker-cplex
https://github.com/victorskl/spring-boot-tute
https://github.com/victorskl/wharfie
https://github.com/victorskl/nectar-os-scripts

Demo

** Need to be connected on UniMelb VPN*
https://115.146.84.84 – Kubernetes Dashboard
http://115.146.84.84 – Grafana Dashboard
http://115.146.84.84:9090 – Prometheus Dashboard
http://115.146.84.84:9093 – Prometheus Operator Alertmanager
http://115.146.85.147 – ATHENA UI (Kube Deployment)
http://115.146.85.102/api/swagger – ATHENA Data Analytics

Bibliography

[1] Mathijs Jeroen Scheepers. "Virtualization and Containerization of Application Infrastructure: A Comparison". In: 2014.

[2] Michael Armbrust et al. "A View of Cloud Computing". In: *Commun. ACM* 53.4 (Apr. 2010), pp. 50–58. ISSN: 0001-0782. DOI: 10.1145/1721654.1721672.

[3] Dirk Merkel. "Docker: Lightweight Linux Containers for Consistent Development and Deployment". In: *Linux J.* 2014.239 (Mar. 2014). ISSN: 1075-3583.

[4] M. Httermann. *DevOps for Developers*. Expert's voice in Web development. Apress, 2012. ISBN: 9781430245704. URL: https://books.google.com.au/books?id=JfUAkB8AA7EC.

[5] Zhanibek Kozhirbayev and Richard O. Sinnott. "A performance comparison of container-based technologies for the Cloud". In: *Future Generation Comp. Syst.* 68 (2017), pp. 175–182.

[6] J. Che et al. "A Synthetical Performance Evaluation of OpenVZ, Xen and KVM". In: *2010 IEEE Asia-Pacific Services Computing Conference*. 2010, pp. 587–594. DOI: 10.1109/APSCC.2010.83.

[7] George Coulouris et al. *Distributed Systems: Concepts and Design*. 5th. USA: Addison-Wesley Publishing Company, 2011. ISBN: 0132143011, 9780132143011.

[8] R. O. Sinnott and W. Voorsluys. "A Scalable Cloud-based System for Data-intensive Spatial Analysis". In: *Int. J. Softw. Tools Technol. Transf.* 18.6 (Nov. 2016), pp. 587–605. ISSN: 1433-2779. DOI: 10.1007/s10009-015-0398-6.

[9] V. Nguyen et al. "A Reconfigurable Agent-Based Discrete Event Simulator for Helicopter Aircrew Training". In: *Proceedings of the 2016 ISMOR Conference*. 2016.

[10] V. Nguyen et al. "Aircrew manpower supply modeling under change: An agent-based discrete event simulation approach". In: *2017 Winter Simulation Conference (WSC)*. 2017, pp. 4070–4081. DOI: 10.1109/WSC.2017.8248116.

[11] ATHENA Developers. *ATHENA v1.0 Technical Documentation, User Guide, Administration Guide, Deployment Guide*. Tech. rep. Melbourne eResearch Group, University of Melbourne, June 2018.

[12] J. O. Kephart and D. M. Chess. "The vision of autonomic computing". In: *Computer* 36.1 (2003), pp. 41–50. ISSN: 0018-9162. DOI: 10.1109/MC.2003.1160055.

[13] Yao Pan et al. "A Performance Comparison of Cloud-based Container Orchestration Tools". In: *submitted to Journal of Concurrency and Computation: Practice and Experience* (December 2017).

[14] Kubernetes Developers. *Kubernetes Documentation*. Tech. rep. v1.10, access June 2018. URL: https://kubernetes.io/docs/.

[15] Brendan Burns et al. "Borg, Omega, and Kubernetes". In: *ACM Queue* 14 (2016), pp. 70–93. URL: http://queue.acm.org/detail.cfm?id=2898444.

[16] Abhishek Verma et al. "Large-scale cluster management at Google with Borg". In: *Proceedings of the European Conference on Computer Systems (EuroSys)*. Bordeaux, France, 2015.

[17] Chenhao Qu, Rodrigo N. Calheiros, and Rajkumar Buyya. "Auto-scaling Web Applications in Clouds: A Taxonomy and Survey". In: *CoRR* abs/1609.09224 (2016). arXiv: 1609.09224.

[18] Long Thai, Blesson Varghese, and Adam Barker. "A Survey and Taxonomy of Resource Optimisation for Executing Bag-of-Task Applications on Public Clouds". In: *CoRR* abs/1711.08973 (2017). arXiv: 1711.08973. URL: http://arxiv.org/abs/1711.08973.

[19] Adel Nadjaran Toosi, Richard O. Sinnott, and Rajkumar Buyya. "Resource provisioning for data-intensive applications with deadline constraints on hybrid clouds using Aneka". In: *Future Generation Comp. Syst.* 79 (2018), pp. 765–775. DOI: 10.1016/j.future.2017.05.042.

[20] Tania Lorido-Botran, Jose Miguel-Alonso, and Jose A. Lozano. "A Review of Auto-scaling Techniques for Elastic Applications in Cloud Environments". In: *J. Grid Comput.* 12.4 (Dec. 2014), pp. 559–592. ISSN: 1570-7873. DOI: 10.1007/s10723-014-9314-7.