

# **Integration of cloud computing paradigms for performant analysis of simulated process engineering applications**

A R&D report (Dissertation) submitted in partial fulfilment  
of the requirements for the degree  
of

**Bachelor of Engineering (Hons)**

at

**The University of Waikato**

by

Caleb Archer

Supervised by:

Mark Apperley, Tim Walmsley



THE UNIVERSITY OF  
**WAIKATO**  
*Te Whare Wānanga o Waikato*

2024

# Abstract

# Acknowledgements

# Declaration of Authorship

# Table of Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Project aim . . . . .	2
1.2	Background . . . . .	2
1.3	Summary of literature review . . . . .	2
<b>2</b>	<b>Project Alignment within Collaborative Work</b>	<b>3</b>
<b>3</b>	<b>Research and Development Approach</b>	<b>4</b>
3.1	Kubernetes cluster architecture . . . . .	5
3.2	Performance analysis experiment components . . . . .	5
3.2.1	Controlled environment . . . . .	5
3.2.2	Controlling variables . . . . .	6
3.2.3	Preliminary performance modelling . . . . .	6
3.2.4	Empirical comparison . . . . .	6
3.2.4.1	Grafana k6 load-testing . . . . .	6
3.2.4.2	Active test client and system monitoring . . . . .	7
3.2.5	Key metrics . . . . .	8
3.2.5.1	Response time . . . . .	8
3.2.5.2	Throughput . . . . .	8
3.2.5.3	Error rate . . . . .	9
3.2.6	Secondary metrics . . . . .	9
3.3	Performance analysis experiments . . . . .	9
3.3.1	Benchmark environment . . . . .	9
3.3.2	Experimental environment . . . . .	10
3.3.3	Test scenarios . . . . .	10
3.3.3.1	Pre-trial scenario set up . . . . .	11
3.3.3.2	Average-load testing . . . . .	12
3.3.3.3	Stress testing . . . . .	12
3.3.3.4	Spike testing . . . . .	12
3.3.3.5	Breakpoint testing . . . . .	13
3.3.4	Horizontal scaling policies . . . . .	13

3.3.4.1	Replica count . . . . .	14
3.3.4.2	Minimum replica count . . . . .	15
3.3.4.3	Resource allocation . . . . .	15
3.3.4.4	Target resource utilisation . . . . .	16
3.3.4.5	Scale-up pod addition limits . . . . .	16
<b>4</b>	<b>Implementation</b>	<b>17</b>
4.1	Raspberry Pi hardware and network provisioning . . . . .	18
4.2	Ansible playbook automation . . . . .	19
4.3	Isolated cluster access . . . . .	20
4.3.1	Network airgapping . . . . .	20
4.3.2	ZeroTier Virtual Private Network (VPN) usage . . . . .	21
4.3.3	Private container image registry mirror . . . . .	22
4.3.4	Cloudflare Tunnel ingress point . . . . .	23
4.4	Kubernetes manifest configuration . . . . .	23
4.4.1	Deployments . . . . .	23
4.4.2	Services . . . . .	23
4.4.3	Ingresses . . . . .	23
4.4.4	Authentication . . . . .	23
4.4.5	Database Stateful Set . . . . .	23
4.4.5.1	Postgres . . . . .	23
4.4.5.2	Connection Pooling . . . . .	23
4.5	Kubernetes deployment automation . . . . .	23
4.5.1	ArgoCD manifest synchronisation . . . . .	23
4.5.2	GitHub Actions platform release pipeline . . . . .	23
<b>5</b>	<b>Results and Discussion</b>	<b>24</b>
5.1	Outcomes . . . . .	25
5.1.1	Resource allocation . . . . .	25
5.1.2	Horizontal scaling policies . . . . .	25
5.1.2.1	Minimum replicas . . . . .	25
5.1.2.2	Target resource utilisation . . . . .	25
5.1.2.3	Variable load conditions (stabilisation window) . . . . .	25
5.1.3	Clustered vs. local PSE optimisation performance . . . . .	25
5.2	Impacts . . . . .	25
5.3	Threats to validity . . . . .	25
<b>6</b>	<b>Conclusion and Future Work</b>	<b>26</b>

# List of Figures



# List of Tables

3.1	API endpoints used for system testing . . . . .	10
3.2	Average load values for API endpoints . . . . .	12
3.3	Stress test load values for API endpoints . . . . .	13
3.4	Spike load values for API endpoints . . . . .	13
3.5	Breakpoint load values for API endpoints . . . . .	13
3.6	Parameters for UOR replica count tests . . . . .	14
3.7	Parameters for FS replica count tests . . . . .	14
3.8	Parameters for UOR minimum replica count tests . . . . .	15
3.9	Parameters for FS minimum replica count tests . . . . .	15
3.10	Parameters for resource allocation tests (UOR and FS) . . . . .	16
3.11	Parameters for resource utilisation tests (UOR and FS) . . . . .	16

# **Chapter 1**

## **Introduction**

## **1.1 Project aim**

The process systems engineering (PSE) field is critical to enabling detailed analysis and optimisation of industrial chemical processes. Despite this, modernisation of accessible optimisation software tools and techniques has slowed, with most software-based analysis performed via desktop applications constrained by the resource limits of local systems. More efficient optimisation techniques are available, but require software programming proficiency not held by the common process engineer. The cloud computing sector has significantly advanced the ability to run and deliver software in a distributed fashion, without requiring clients to procure and maintain physical hardware. Cloud-native software systems are better positioned to respond to dynamic usage and allocate computational resources efficiently based on need. There has been minimal convergence of the PSE and cloud computing fields.

This work details and evaluates the development of a cloud-native PSE software platform (the Ahuora Digital Platform) on a physical Kubernetes cluster, comparing the performance of the different software components running on the cluster versus a local deployment of said components.

## **1.2 Background**

## **1.3 Summary of literature review**

## **Chapter 2**

# **Project Alignment within Collaborative Work**

## **Chapter 3**

# **Research and Development Approach**

## 3.1 Kubernetes cluster architecture

## 3.2 Performance analysis experiment components

### 3.2.1 Controlled environment

Difficulties were encountered obtaining consistent results when conducting ad hoc performance analysis experiments. Because of the usage of a 4G router for providing internet access to the Raspberry Pi cluster, highly variable bandwidth and latency conditions were encountered. One performance test with increasing load up to sixty requests per second might encounter exponentially increasing response times half-way during the test, while another repetition would see this occur three-quarters through the test.

Another problem caused by relying on 4G mobile network infrastructure is the upper bandwidth limitation. The 4G router in use has (in its physical position in a lab) a peak download speed of ~50.0 Mbit/second (~6.25 Mbyte/second), and an upload speed of ~20.0 Mbit/second (~2.5 Mbyte/second). With increasing request rates during performance tests, a bandwidth bottleneck can be encountered, with the cluster unable to serve data because of a lack of sufficient capacity. Though perfect bandwidth and latency stability cannot be expected on the internet, mobile networks are especially prone to instability, and inhibit the ability to replicate a standard environment for applications deployed to “the cloud”, i.e. a datacentre with high bandwidth capacity. This is exacerbated by the usage of Cloudflare Tunnel on the cluster to enable external access, which provides a lower bound on maximum bandwidth than if the router was directly accessible via the internet.

To conduct consistent load-tests that provide a higher signal to noise ratio, a stable cloud infrastructure environment was simulated. Load-tests were performed from the ingress node, sending requests to the Kubernetes control node over the LAN (local area network) provided by the network switch. An Auckland-based data centre environment will be replicated, with an expected latency of 5 milliseconds from an in-country client device to the data centre. This latency will be simulated via the `tc` Linux command as depicted in Listings 3.1 and 3.2. The `tc` command allows the traffic travelling via a particular network interface to have artificial constraints added, such as with bandwidth and latency. Prior to every test, 5 milliseconds of latency was added, and then removed after each test.

### Listing 3.1: Pre-test latency simulation command

```
sudo tc qdisc add dev eth0 root netem delay 5ms
```

### Listing 3.2: Post-test latency simulation teardown

```
sudo tc qdisc del dev eth0 root netem
```

## 3.2.2 Controlling variables

As with any valid experiment, any one load-test carried out assesses the performance differences observed after modifying one variable. For example, the number of replicas for a Kubernetes deployment, in one set of experiments, have been varied, while all other configuration fields have been kept the same. In some experiments, an entire block of configuration (such as the presence of a `HorizontalPodAutoscaler` in a `Deployment`) have been independently assessed, and everything else controlled.

## 3.2.3 Preliminary performance modelling

## 3.2.4 Empirical comparison

### 3.2.4.1 Grafana k6 load-testing

Grafana k6<sup>1</sup>, or k6, is a load-testing tool for assessing the performance of local and remote software systems. It is a script-based tool, where developers use JavaScript to define their testing logic, scenarios, and metrics, and then these tests are run using a Go-based custom JavaScript runtime to minimise test performance overheads.

k6 tests are built around the concept of a “Virtual User” or VU, which represents one unit of work that carries out one iteration of the defined test logic. The test logic may carry out a simple action such as making a request to a website at some URL, and waiting for the response. During a test, k6 displays

---

<sup>1</sup>Grafana k6: <https://k6.io/>

the number of iterations being completed by VUs, as well as the number of VUs that are stalled in waiting for their respective request responses. VUs are run concurrently within a runtime thread pool, with each VU requiring an allocation of memory depending on the work each test iteration does.

Test scenarios can be defined in a number of ways. One test may utilise a “constant arrival rate” executor, which attempts to send a fixed number of test iterations for every unit of time (seconds, minutes, etc.) using the available pool of VUs. Another could use a “ramping arrival rate” executor, which can be configured to start an increasing, decreasing, or even constant number of test iterations per unit of time, at different stages of the test.

There are several other scenario executors that can be used within k6, but these have the possibility of skewing test results. For example, the “constant VUs” executor tries to use a set number of VUs to launch as “as many iterations as possible”. However, if the executor detects that the time required to complete an iteration increases beyond a certain limit (i.e. the request response time is increasing), it will decrease the iteration start rate, and thereby reduce the number of requests being started. When the goal is to test how a system responds under specific conditions, this style of execution is unfavourable, given that it will dynamically change the test conditions. As such, the executors utilised in testing the Ahuora Digital Platform have been limited to the constant and ramping arrival rate executors.

#### **3.2.4.2 Active test client and system monitoring**

As performance tests were carried out, the Grafana monitoring tool was used to gain insight into how the tests were impacting the different parts of the cluster, including at the container level, node level, and the overall cluster. Apart from this, it was important to ensure that the intended loads could actually be generated by the client sending the requests to the system. If the client encounters CPU or memory limits when attempting to generate load, then test results will be affected by the performance of the client, rather than the system, and prevent any meaningful insights being obtained. To ensure that client-side test throttling is not taking place, the `htop` Linux command was used to observe client CPU and memory usage. The thresholds of 90% CPU usage (across all cores) and 80% memory usage were chosen as causes for concern for test validity.



### **3.2.5 Key metrics**

There are three key metrics that will be used to assess system performance impacts: the HTTP response time, throughput and the error rate. Within the context of self-adaptive systems, as described by Weyns [1], there are two levels of a system that can be optimised. Improvements can be made to the managed system, or to the managing system. For the Ahuora Digital Platform, the optimisation explored here will focus on optimisation of the managing system, or the Kubernetes cluster. These metrics will serve to inform steps towards such optimisation.

#### **3.2.5.1 Response time**

The response time for an individual request is the total length of time taken to receive a response from a destination endpoint. This includes the duration of all steps involved in the request, including the time taken to establish an initial connection, perform a TLS handshake, wait for the response, and receive the data. As a metric, it allows an analyst to observe how a system responds over time to increasing load, with a worsening response time indicating that the system is being throttled or is reaching a performance ceiling. From a developer perspective, the response time is important to track, as an increasing response time worsens the experience for users of a product, who may perceive a slowly responding application as a reflection of poor software quality.

Response times will be assessed using an exponentially-weighted moving average (EWMA), along with comparisons at different moving percentiles. Use of a rolling median would appear either too unstable with a small window, or lag behind in showing recent trends. Using an EWMA allows recent data to be paid more attention, and enables better correlation analysis to be conducted.

#### **3.2.5.2 Throughput**

The throughput of a system is the number of requests that are processed per second. This metric can be used to identify the peak capacity of the system, as well as identify the thresholds or points at which system performance degrades. As opposed to the response time, which is concerned with the performance of individual requests, throughput is a metric that provides insight only with respect to behaviour of the overall system. One goal within a performance optimisation context is to maximise the peak throughput of a system to serve more users concurrently.

### **3.2.5.3 Error rate**

Along with speed-oriented metrics like the response time and throughput, it is critical to keep track of the error rate, or the proportion of HTTP requests that failed. Error rates can increase under overloaded system conditions, and can provide early indication that a sustainable throughput threshold has been passed. Another reason to record errors is to ensure that they do not cause invalid observations to be made about test data. In some situations, failed requests may have lower response times than their successful counterparts, which may mislead one to think the system is more responsive than it actually is.

### **3.2.6 Secondary metrics**

Both CPU and memory usage will provide additional insight into system performance influences. For example, the observation of high consumption of allocated CPU time within a pod at the same time as request rate instability would possibly indicate the pod is reaching computation limits. These metrics will be recorded at the container level as well as the node level.

## **3.3 Performance analysis experiments**

### **3.3.1 Benchmark environment**

A reference point to compare collected test metrics against is needed to obtain meaningful insights with respect to performance improvements. A series of benchmarks were created to compare various Kubernetes cluster configuration impacts against. As one of the purposes of this project is to analyse how transitioning to a cloud-based system can affect the performance of a process engineering simulation application, all benchmarks assess this system running entirely on one system, with all interprocess communication taking place locally.

These benchmarks will run on the ingress node, which does not participate in the Kubernetes cluster, but can access cluster nodes over the network. This local deployment is set up with Docker Compose<sup>1</sup>, which allows a set of containerised software components to be configured and run with

---

<sup>1</sup>Docker Compose: <https://docs.docker.com/compose/>

ease on a local system. Docker configures a private network shared by each container defined in the configuration manifest, and enables DNS-based communication between containers. The manifest used in benchmarks defines four containers: the Django API server, a PostgreSQL database, the PgBouncer database connection pooler, and the IDAES mathematical solver service. The API server uses PgBouncer as its database endpoint, which manages the reuse of connections made to the database.

### 3.3.2 Experimental environment

Load tests will be run from the ingress node against the cluster, with traffic directed at the control node serving as the public endpoint.

### 3.3.3 Test scenarios

Various load profiles are used to assess system performance and dynamics. This allows for observation of whether the system is appropriately tuned or capable of responding to both flat and variable load profiles, as well as handling extended load periods. Two core API endpoints available on the Django API server are used for each test scenario, as described in Table 3.1. Software versions used are: Django API 0.0.17, IDAES worker 0.0.10 and Postgres 16.4.

Test type	Endpoint	Description
Unit operation retrieval (UOR)	/api/unitops/unitops/?flowsheetOwner={id}	Retrieves all unit operations belonging to a particular flowsheet.
Flowsheet solving (FS)	/api/solve/idaes/	Serialises a flowsheet and sends an internal request to the IDAES service to solve the flowsheet's parameters.

Table 3.1: API endpoints used for system testing

The unit operation retrieval endpoint interacts with the Django API server and the PostgreSQL

database. The flowsheet solving endpoint interacts with both of these, as well as the IDAES service. The specific load profiles used against either endpoint will differ, as these endpoints have dissimilar baseline response times. The time required to make a single unit operation retrieval request is between 20 and 40 milliseconds, while a solve request may take 500 to 1000 milliseconds.

Flat load profiles will have initial ramp-up and ramp-down periods between their core profiles to provide corresponding system warm-up and cooldown periods. These periods will be short, fixed at 5% of the total test duration each.

To avoid individual tests skewing the overall results, each test scenario will be run three times, with the resulting data averaged across the three trials.

### **3.3.3.1 Pre-trial scenario set up**

Though these experiments do not interact with nor test the Ahuora Digital Platform's user interface, a React-based front-end, it is still necessary to construct a solvable flowsheet that can be used to retrieve unit operations from, and make solve requests. This flowsheet needs to be consistent between test scenarios and their trials. At the same time, the flowsheet needs to be deleted after usage during a trial to avoid an excessive and duplicated presence for users of the live platform when viewing available flowsheets. As such, it was necessary to construct pre and post-trial flowsheet set up and teardown scripts.

These Node.js-based scripts use the existing Django API to create (and delete) a flowsheet, unit operation, and all their corresponding properties and elements of configuration. The flowsheet constructed consists of a single pump with two material streams, one serving as input to the pump, and another as output. The specific steps involved in creating the flowsheet are:

1. Create flowsheet object
2. Add water compound to flowsheet
3. Add Helmholtz Water property package to flowsheet
4. Create pump unit operation
5. Set pump efficiency to 0.5
6. Set pump outlet pressure to 200

7. Set input stream temperature to 80
8. Set input stream pressure to 100
9. Set input stream molar flow to 1000
10. Set input stream vapor fraction to 0
11. Set input stream water compound amount to 1

### 3.3.3.2 Average-load testing

This load profile will test the system against a flat load profile, or a fixed number of requests per second. The intent is to assess how the system performs under what can be called an “average” load. The Ahuora Digital Platform is not publicly available as of writing, and so the average load will have to be assumed at some value, as we do not have actual usage data that can be used to inform an appropriate average load value. Tests will last ten minutes.

Test type	Load (requests/second)
UOR	15
FS	1

Table 3.2: Average load values for API endpoints

### 3.3.3.3 Stress testing

The stress testing load profile will assess the system at an “above-average” load, which will be set at 200% of the previously outlined average load values (Table 3.2). The goal is to determine what performance degradation (if any) happens when the system experiences load that is still within an expected range.

### 3.3.3.4 Spike testing

Spike tests will test how the system responds to a rapidly increasing, very high load (that may be overwhelming) that then rapidly decreases. These tests will last for four minutes.

Test type	Load (requests/second)
UOR	30
FS	2

Table 3.3: Stress test load values for API endpoints

Test type	Load (requests/second)
UOR	80
FS	8

Table 3.4: Spike load values for API endpoints

### 3.3.3.5 Breakpoint testing

These load profiles will linearly increase the number of requests made against the system per second to some extremely high threshold. This will allow the “breakpoint” of the system to be identified, or the point at which system performance either rapidly deteriorates or completely collapses. These tests will run for ten minutes.

Test type	Load (requests/second)
UOR	200
FS	32

Table 3.5: Breakpoint load values for API endpoints

## 3.3.4 Horizontal scaling policies

Within Kubernetes, an operator can define the scaling policies associated with a specific workload. Scaling policies define how many replicas of a workload should be running, based on either static values, or dynamic metrics such as CPU, network or memory usage. As a workload operates, the cluster control plane will monitor the resource utilisation of the workload, and perform scaling operations

(both scaling up and down) based on the corresponding scaling policies. The following experiments will assess different scaling policy elements and their impact on the selected key metrics.

### 3.3.4.1 Replica count

On a Kubernetes Deployment object, the “replicas” field statically defines the number of pod replicas that should be running on the cluster. This test will assess how the number of active replicas influences system throughput and response time.

**Load profiles used:** Average, stress, spike, breakpoint

Test iteration	Django replicas
1	1
2	2
3	4
4	8

Table 3.6: Parameters for UOR replica count tests

Test iteration	Django replicas	IDAES replicas
1	1	1
2	2	2
3	4	4
4	8	8
5	1	8
6	2	8
7	4	8
8	8	8

Table 3.7: Parameters for FS replica count tests

### 3.3.4.2 Minimum replica count

The minimum replica count is a property of the *HorizontalPodAutoscaler*, which defines the scaling policies to be adhered to by the control plane's autoscaling controller. The minimum replica count, as plainly indicated, sets a requirement for the smallest number of pod replicas that are allowed to run, regardless of whether the target resource utilisation threshold is being met.

**Load profiles used:** Spike, breakpoint

Test iteration	Minimum replica count (Django)
1	1
2	2
3	4

Table 3.8: Parameters for UOR minimum replica count tests

Test iteration	Minimum replica count (Django and IDAES)
1	1
2	2
3	4

Table 3.9: Parameters for FS minimum replica count tests

### 3.3.4.3 Resource allocation

CPU and memory resource requests are set on deployment manifests for containers within a pod. In this context, each pod has one container. These resource requests indicate to the cluster the minimum amount of CPU time that needs to be made available to the pod. The pod may not use all of its resource request, but this allows the cluster pod scheduler to decide how to allocate unscheduled pods across the CPU resource pool available. The unit for a CPU resource request is the millicore, which is one thousandth of one physical CPU core.

**Load profiles used:** Spike, breakpoint



Test iteration	CPU allocation (millicores)
1	500
2	1000
3	2000

Table 3.10: Parameters for resource allocation tests (UOR and FS)

#### 3.3.4.4 Target resource utilisation

This metric is used by the cluster to make scaling decisions. Using the CPU resource request usage across all pod replicas within a deployment, an average utilisation percentage is taken, and compared with the target utilisation percentage defined in the corresponding *HorizontalPodAutoscaler* scaling policy. If the average utilisation exceeds the target by a certain threshold, then more pods are scheduled and run to achieve the target. Likewise, utilisation falling below the target by the same threshold will result in pods being removed.

**Load profiles used:** Spike, breakpoint

Test iteration	Average CPU utilisation target (%)
1	25
2	50
3	75
4	90

Table 3.11: Parameters for resource utilisation tests (UOR and FS)

#### 3.3.4.5 Scale-up pod addition limits

**Load profiles used:** Spike, breakpoint

# **Chapter 4**

## **Implementation**

## 4.1 Raspberry Pi hardware and network provisioning

For ease of access and cost minimisation purposes, a set of eight Raspberry Pi 5 computers was obtained to run the Kubernetes cluster on. Each device has an active cooler component installed to effectively cool the CPU (Central Processing Unit) and prevent system throttling. The usage of these devices allows for the construction of a physically compact computing cluster at low cost.

A headless (sans desktop interface) version of Raspberry Pi OS (operating system) was loaded onto eight corresponding SD cards, which each device uses as primary storage. The headless version of the OS strips the resource consumption of the desktop user interface, which is not required, as most interaction with each device will be automated over the network, requiring no more complication than a remote CLI (Command-Line Interface) provides.

To allow the Raspberry Pi devices to communicate and form a computing cluster, a network switch was used. The switch has eight ethernet ports, and each Pi connects to the switch via CAT-6 ethernet cables. On initial start-up, the Pi devices did not have IP addresses assigned, and could only be identified by their MAC addresses, which are not suitable for higher-level communication protocols, which rely on IP addresses as part of the low-level IP protocol (Internet Protocol). Along with this, the devices needed access to the internet, and as such, a network router was required. Because of the network security concerns held by campus ITS (Information Technology Services), it was not appropriate to connect the cluster to the primary university network in order to gain internet access. Instead, a 4G Teltonika RUTX11 cellular router was procured, which could connect to the 2degrees-managed university 4G network, and thereby obtain internet access. With only eight ethernet ports on the switch, cluster nodes (Pi devices) requiring internet access were connected to the router over Wi-Fi. Two of the eight nodes have been configured to access the internet this way.

After this, the nodes still needed IP addresses assigned for the network switch interface. A DHCP (Dynamic Host Configuration Protocol) server called DNSMasq was set up on one of the Wi-Fi connected nodes (called the *ingress* node). DHCP allows for devices on a network to be automatically assigned IP addresses from an available address pool, avoiding the tedious process of manually configuring network interfaces on each individual device. In this context, it was necessary to assign static (unchanging) IP addresses to each device, so DNSMasq was configured to assign specific addresses based on the MAC address of each device. The ingress node was assigned 192.168.100.101, with the

rest numbered from 192.168.100.121 to 192.168.100.127. DNSMasq was also used as a DNS (Domain Name Server) server to enable name-based communication over the network, instead of being forced to recall specific IP addresses.

One issue with managing an airgapped (internet-isolated) computer network is the potential for dates and times to become desynchronised, especially following a power outage or any instance where nodes are powered off. Computers typically synchronise with the actual time via time servers accessible over the internet, but perform no such synchronisation without internet access. This problem was encountered during the development process, where logs and metrics that were expected from each node on the cluster were missing. Because of prior cases where some nodes had been powered off for an extended period of time (several weeks), and a lack of hardware clocks on Raspberry Pi models, the date and time on these nodes lagged by more than a month. This caused the logs and metrics to be rejected by the monitoring tool to which they are uploaded. To resolve this, an NTP (Network Time Protocol) server (ntpsec) was deployed on the ingress node, and all airgapped nodes were configured to use the ingress node as their time server. Since the ingress node can retrieve the actual time over the internet, it can provide the actual time to its airgapped clients.

## 4.2 Ansible playbook automation

To manage the bulk of device and system-level configuration of the cluster, a configuration automation tool called Ansible was heavily used throughout the development process. Ansible allows a developer to define a *playbook*, which consists of a set of tasks that will be run in sequence on a target host (a remote device), or a group of hosts. These tasks may be defined as arbitrary shell commands, but they often provide a higher level of abstraction, where a developer can easily specify the parameters from a restricted set, rather than having to remember an exact series of (potentially confusing) mnemonics.

Playbooks can also include *roles*, which include their own sets of tasks, but are focused on grouping related tasks together, and allowing parameters to be easily re-used amongst said tasks. If made analogous to an imperatively written programming language, a playbook is like a program, and a role is akin to a class or module.

The Ansible playbooks used in configuring the system were adapted from the official K3s Ansible

repository<sup>1</sup>, with many modifications. Included playbooks are: *reboot*, for restarting all Kubernetes cluster nodes; *registry*, for setting up a container image registry mirror; *reset*, for removing configuration and components installed via the site playbook; *site*, for performing the entire cluster software installation and configuration process; and *upgrade*, for updating installed cluster software to a new version.

- *airgap*: Configures hosts for an air-gapped environment.
- *k3s\_server*: Configures K3s control (master) nodes.
- *k3s\_agent*: Configures K3s agents (worker nodes).
- *k3s\_deployments*: Configures third-party software to be deployed at installation time.
- *k3s\_upgrade*: Performs K3s upgrading process.
- *ntp\_time\_server*: Installs an NTP time server and configures clients to use it.
- *prereq*: Performs any prerequisite configuration before cluster start-up.
- *raspberrypi*: Performs configuration on all remote hosts specific to Raspberry Pi systems.

## 4.3 Isolated cluster access

### 4.3.1 Network airgapping

As a step towards better cluster security, access to the internet for nodes running Kubernetes is heavily restricted. No nodes beyond the control node are connected to the 4G router, and are limited to local network communication. In the case of maliciously crafted or modified software that may inadvertently be deployed on the cluster (or directly to the base hosts), their ability to exfiltrate information or otherwise communicate with the outside world has been minimised. The K3s control node requires internet access in order to perform tasks such as push collected logs and metrics, check for configuration updates in the associated cluster GitHub repository and install plugins.

---

<sup>1</sup> K3s Ansible GitHub repository: <https://github.com/k3s-io/k3s-ansible>

### 4.3.2 ZeroTier Virtual Private Network (VPN) usage

With these network limitations in place, it was still necessary to have a method to remotely access and manage nodes within the cluster, while maintaining security requirements. While it was possible to log in to the cluster via an authenticated laptop connected to the cluster 4G router, this was certainly the least convenient option, especially when working from a remote location, or even an office desktop using a separate network. On the other hand, exposing an SSH (Secure Shell) server port to the outside world would have provided convenience, but less security confidence.

ZeroTier<sup>1</sup> is a VPN (Virtual Private Network) service that allows users to connect devices of varying types to a virtual network that appears to behave the same way as a physical network switch. IP addresses from private subnets are assigned dynamically to each authorised device, which can then communicate with other devices on the network from anywhere in the world, as long as the device has internet access. ZeroTier provides a set of free root nodes that facilitate the establishment of connections between devices, which then continue to transmit data over a direct P2P (peer-to-peer) connection. Traffic between devices is end-to-end encrypted, meaning that in-flight data cannot be intercepted and interpreted by actors in the middle of a connection, even ZeroTier themselves (when a P2P connection cannot be established and has to be relayed via ZeroTier nodes).

The cluster ingress node is the core device connected to a ZeroTier VPN. Other devices that need to remotely access the cluster do so first by joining the VPN, and then creating an SSH connection to the ingress node via the ingress node's VPN-allocated IP address. Following this, the rest of the physical cluster network can be accessed over SSH via the ingress node.

With this strategy, any services that need to be exposed to developers but not to the wider internet can be utilised via the VPN, providing a simple, secure and convenient management context. During early development of the Ahuora Digital Twin platform, there was no authentication system in place, but stakeholders of the platform needed to be able to test it without having to set up a manual deployment. To enable testing, the control node of the cluster was added to the VPN, and then the device to perform the testing from, where it could then access the front-end of the platform securely.

---

<sup>1</sup>ZeroTier VPN: <https://www.zerotier.com/>

### 4.3.3 Private container image registry mirror

Because of the internet access restrictions on the cluster, Kubernetes Pods attempting to retrieve container images from external sources (such as Docker Hub) will repeatedly fail to deploy. This presents a problem, as some form of external access is required, but it is not acceptable to provide broad internet access to all cluster nodes. In this scenario, it is necessary to use some form of limited proxying solution, where requests can be made to a service local to the network, which has access to the internet, and can pull container images from external image registries on behalf of clients.

A container image mirror service called *oci-registry*<sup>1</sup> was used to achieve this functionality. The ingress node was configured to run *oci-registry* and expose it to the physical network. All nodes within the Kubernetes cluster are likewise configured to make image pull requests to the ingress node. When a request is made, *oci-registry* checks if the requested image is present in its cache. If it is not present, it retrieves it from the requested source; otherwise, it is served from the cache. In some cases, the requested image may be corrupted during transit or storage, causing dependent pods to enter a failure loop, either from detecting image corruption, or attempting to run the image and encountering a segmentation fault from invalid memory addressing. To handle this, it was necessary to enable cache integrity checking whenever stored images were retrieved: if the hash of an image does not match the expected value, the registry mirror will retrieve the upstream image copy again.

---

<sup>1</sup>Container image registry: <https://github.com/mcronce/oci-registry>

#### **4.3.4 Cloudflare Tunnel ingress point**

### **4.4 Kubernetes manifest configuration**

#### **4.4.1 Deployments**

#### **4.4.2 Services**

#### **4.4.3 Ingresses**

#### **4.4.4 Authentication**

#### **4.4.5 Database Stateful Set**

##### **4.4.5.1 Postgres**

##### **4.4.5.2 Connection Pooling**

### **4.5 Kubernetes deployment automation**

#### **4.5.1 ArgoCD manifest synchronisation**

#### **4.5.2 GitHub Actions platform release pipeline**



## **Chapter 5**

### **Results and Discussion**

## **5.1 Outcomes**

### **5.1.1 Resource allocation**

### **5.1.2 Horizontal scaling policies**

#### **5.1.2.1 Minimum replicas**

#### **5.1.2.2 Target resource utilisation**

#### **5.1.2.3 Variable load conditions (stabilisation window)**

### **5.1.3 Clustered vs. local PSE optimisation performance**

## **5.2 Impacts**

Efficiency, accuracy, feasibility (examples of demonstrable impacts).

## **5.3 Threats to validity**

## **Chapter 6**

### **Conclusion and Future Work**

# Bibliography

- [1] D. Weyns, *Engineering Self-Adaptive Software Systems – An Organized Tour*. Sep. 2018.