

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

1	Introduction	1
1.1	Project aim	2
1.2	Background	2
1.3	Summary of literature review	2
2	Project Alignment within Collaborative Work	3
3	Research and Development Approach	4
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	Post-trial metrics collection	12
3.3.3.3	Base k6 load test configuration	12
3.3.3.4	Average-load testing	14
3.3.3.5	Stress testing	14
3.3.3.6	Spike testing	14

3.3.3.7	Breakpoint testing	15
3.3.4	Horizontal scaling policies	15
3.3.4.1	Replica count	16
3.3.4.2	Minimum replica count	16
3.3.4.3	Resource allocation	17
3.3.4.4	Target resource utilisation	18
3.3.4.5	Scale-up pod addition limits	18
4	Implementation	19
4.1	Raspberry Pi hardware and network provisioning	20
4.2	Ansible playbook automation	21
4.3	Isolated cluster access	22
4.3.1	Network airgapping	22
4.3.2	ZeroTier Virtual Private Network (VPN) usage	23
4.3.3	Private container image registry mirror	24
4.3.4	Cloudflare Tunnel ingress point	24
4.4	Platform migration to Kubernetes cluster	25
4.4.1	Django API	26
4.4.2	Front-end	27
4.4.3	IDAES service	27
4.4.4	Public gateway	28
4.4.4.1	Front-end ingress	28
4.4.4.2	Django API ingress	29
4.4.4.3	Authentication ingress	29
4.4.5	Authentication	29
4.4.6	Postgres database	29
4.4.6.1	Postgres	29
4.4.6.2	Connection Pooling	29
4.4.7	Monitoring	29
4.5	Kubernetes deployment automation	29
4.5.1	ArgoCD manifest synchronisation	29
4.5.2	GitHub Actions platform release pipeline	29
5	Results and Discussion	30
5.1	Outcomes	31
5.1.1	Resource allocation	31
5.1.2	Horizontal scaling policies	31
5.1.2.1	Minimum replicas	31
5.1.2.2	Target resource utilisation	31

5.1.2.3	Variable load conditions (stabilisation window)	31
5.1.3	Clustered vs. local PSE optimisation performance	31
5.2	Impacts	31
5.3	Threats to validity	31
6	Conclusion and Future Work	32

List of Figures

List of Tables

3.2	API endpoints used for system testing	11
3.3	Base options set for k6 load tests	13
3.4	Average load values for API endpoints	14
3.5	Stress test load values for API endpoints	14
3.6	Spike load values for API endpoints	15
3.7	Breakpoint load values for API endpoints	15
3.8	Parameters for UOR replica count tests	16
3.9	Parameters for FS replica count tests	16
3.10	Parameters for UOR minimum replica count tests	17
3.11	Parameters for FS minimum replica count tests	17
3.12	Parameters for resource allocation tests (UOR and FS)	18
3.13	Parameters for resource utilisation tests (UOR and FS)	18

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 k6¹, 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 the number of iterations being completed by VUs, as well as the number of VUs that are stalled in

¹Grafana k6: <https://k6.io/>

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 executor utilised in testing the Ahuora Digital Platform has been limited to the ramping arrival rate executor, which can model both flat and variable load profiles.

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 at 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 Compose¹, which allows a set of containerised software components to be configured and run with

¹Docker 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 entrypoint. All tested software components will be run on the K3s Kubernetes cluster. The core software components interacted with are the same as in the benchmark environment, though the Kubernetes environment introduces a central proxy layer through which all API traffic travels through, and is authorised by.

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 3.2. Software versions used are: Django API 0.0.17, IDAES worker 0.0.10 and Postgres 16.4.

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. All load profiles will have a 30-second “graceful stop” period, which allows requests already in the queue to complete, preventing the processing of requests from

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 3.2: API endpoints used for system testing

the end of one trial affecting the results of the immediately subsequent trial.

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 Post-trial metrics collection

For tests remotely run against the cluster, metrics internally generated and uploaded to Grafana Cloud by the cluster are pulled by a custom k6 test summary generator, obtaining metrics for container CPU, memory, disk and network usage, as well as general cluster metrics reporting pod and container counts. This data is used in conjunction with test results to find associations between test variables and develop a better model of system performance influences.

3.3.3.3 Base k6 load test configuration

k6 option	Value	Usage explanation
insecureSkipTLSVerify	true	The TLS certificate presented to clients by the cluster is self-signed. External traffic from the internet accesses the platform via Cloudflare, which presents its own trusted TLS certificate. Since experimental load tests will be conducted over the local network, the k6 HTTP client by default will fail because of TLS verification failure. The IP address locally associated with the domain on the ingress node can be trusted.
noConnectionReuse	true	Every newly started request should simulate the action of an independent user, and therefore consume the expected resources in establishing a new connection to the platform backend.
discardResponseBodies	true	Since the response body is not being used in test logic, memory consumption by the testing client can be reduced and thereby reduce any undue influence the testing client may have on test results.
maxRedirects	0	It is expected that all requests made will result in a direct response. Any HTTP redirects may indicate, in the case of cluster-bound requests, that the test client is not authenticated and has been redirected to a sign-in page. In any case, redirects would distort test results. Redirects should cause a request iteration to report failure.

Table 3.3: Base options set for k6 load tests

3.3.3.4 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 3.4: Average load values for API endpoints

3.3.3.5 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 3.4). The goal is to determine what performance degradation (if any) happens when the system experiences load that is still within an expected range.

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

Table 3.5: Stress test load values for API endpoints

3.3.3.6 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	80
FS	8

Table 3.6: Spike load values for API endpoints

3.3.3.7 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 a maximum of ten minutes. To prevent tests for running far beyond the breakpoint unnecessarily, a threshold will be defined for each breakpoint load profile variant, which will mandate that the number of virtual users (VUs) does not exceed 100. If this threshold is exceeded, then the test will cease.

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

Table 3.7: 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 3.8: 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 3.9: 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 3.10: Parameters for UOR minimum replica count tests

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

Table 3.11: 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 3.12: 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 3.13: 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

repository¹, 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.

¹ 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.

ZeroTier¹ 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.

¹ZeroTier 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*¹ 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.

4.3.4 Cloudflare Tunnel ingress point

The 4G router used does not have a publicly routable IP address assigned to it at this time, let alone one that is static. However, it is still necessary to make the platform available to users over the internet. While using the VPN to achieve user access is possible, this does not expand well beyond a limited pool of users, who already would need to gain access to a VPN that is intended to be private and restricted to developer use.

Cloudflare has a tunnel² service as part of their Zero Trust product range. The service allows devices with internet access to create a tunnel connection with Cloudflare using a background service called Cloudflared, which can receive traffic proxied through Cloudflare, and forward it to the intended

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

²Cloudflare Tunnel: <https://www.cloudflare.com/products/tunnel/>

internal service.

Within the cluster, the Cloudflared service worker runs on the control node, and forwards HTTPS traffic to the handling service at port 443. With Cloudflare serving as the DNS provider, traffic sent to Ahuora-controlled domains (such as `api.ahuora.org.nz` or `ahuora.org.nz`) can be internally forwarded over the tunnel, and subsequently sent to the Kubernetes cluster and handled by the corresponding service. Cloudflare then handles the responsibility of providing a trusted HTTPS connection to users over the internet. This allows the Ahuora Digital Platform to be made accessible over the internet despite the absence of a publicly accessible router.

There are some downsides to using Cloudflare's tunnel service. Compared to direct traffic access, the available egress bandwidth is reduced, and with already-limited bandwidth capacity due to the usage of a mobile network (instead of a datacentre with fibre infrastructure), the ability to serve external users is partially diminished. To compound this, the latency when interacting with the Ahuora Digital Platform is substantially increased. Another small downside is the tunnel service taking some of the available compute resources when running on the control node, increasing with the amount of traffic.

4.4 Platform migration to Kubernetes cluster

The Ahuora Digital Platform includes several constituent software components that operate together to serve the platform. These include the Django API, the React-based front-end, the IDAES PSE (process systems engineering) solver service, and a Postgres database for housing data accessed and managed through the API. Each of these components needs to be deployed by the Kubernetes cluster.

Within a Docker-based environment, the individually runnable software unit is known as a container, which has an isolated filesystem with all the necessary dependencies for the target software to run. In Kubernetes, the most granular runnable software unit is instead called a "pod". A pod can manage several containers, and are treated as one element operated on by the cluster control plane, including initialisation, termination and scaling actions.

A Deployment object defines the container image to run, ports to expose, environment variables and secrets to consume, healthcheck endpoints, and more.

4.4.1 Django API

The Django API was one of the first core deployed platform elements. The API is managed as a deployment object, and assigned the “django-api” label to allow a Kubernetes service to target the pods belonging to the corresponding deployment.

Since the API relies on access to a Postgres database to perform any data-related operation, it requires access to a set of credentials that can authenticate its database access. It would not be appropriate to explicitly define these credentials within the internal settings of the API. First, keeping credentials available in code is a substantial security risk, where anyone who has read access either to the git repository of the API, or the API’s published container image, would have knowledge of the credentials required to access the database. Though the database is not externally accessible outside of the cluster, this does broaden the scope for access in case of an internal breach of security. Second, static credentials are inflexible and difficult to manage. Entirely new versions of the API would have to be published and synchronised to the cluster deployment if any credential rotation was performed. As such, Kubernetes’ native secret management system is used to inject database credentials as environment variables.

Other elements of configuration that should be dynamically assigned have also made use of environment variables for configuration injection, instead of static assignment. This includes the private service URL for IDAES, the DNS name of the database, the permitted host names that can be used to access the API, and the runtime mode.

With the deployment being able to manage any number of Django API replicas, it is necessary to provide load-balancing that can abstract away the responsibility of service selection from internal clients. Kubernetes allows one to define a “Service” object which takes on this responsibility. While a service has several potential modes of operation, the one appropriate for the Django API is ClusterIP. A ClusterIP service targets a series of pod endpoints based on a pod selector (matching a label(s) defined on the selector with labels found on pods). This service is then assigned a single private IP address that can be used from within the cluster to access pods without needing any knowledge of their individual addresses. This service can then distribute traffic to the target pods.

4.4.2 Front-end

The platform is primarily interacted with via a React-based front-end client, which in turn makes calls to the Django API for retrieving and persisting data created by the user. During development, the front-end is accessed via a developer-oriented static web server, accessing each of the individual component files one-by-one when navigating to a page. This approach does not bode well for what is meant to be a production version of the application, with the total load time being potentially tens of seconds long.

To create an efficient production version of the front-end to run on the cluster, it was necessary to alter the container image build process to construct an optimised static build. The Vite¹ build tool is used to produce minimised HTML, JavaScript and CSS files. These files are then transferred to an Nginx web server image, where Nginx then ultimately serves the files at runtime. The combination of an optimised static files bundle and usage of a purpose-build web server allows for a proper production version of the front-end to be deployed.

Within this new build process, it was also necessary to introduce build variables that allow for values to be injected into the client at build-time. In particular, the base API endpoint that the front-end uses to make API calls requires this injection step, as it varies depending on whether it is in a development or production environment. The development environment will refer to localhost, while in production this must be the API's assigned FQDN (fully qualified domain name), api.ahuora.org.nz.

With these elements in place, the front-end could be deployed to the cluster, and provided a ClusterIP service to enable internal access as with the Django API.

4.4.3 IDAES service

A separate application was created to handle receiving flowsheet solve requests using the IDAES PSE framework. This IDAES service is used internally by the Django API, but can be independently scaled up and down separate to the API. This IDAES worker was deployed (once more) using a Kubernetes deployment object, along with a ClusterIP service pointing to this deployment.

¹ Vite: <https://vitejs.dev/>

4.4.4 Public gateway

Initially, each of the public-facing platform components, the API and front-end, were to be exposed using a load balancer service type. This was the configuration for some time while the platform was only exposed over the ZeroTier VPN during early testing phases. However, it became apparent that a centralised public endpoint was needed to manage access to the platform. Requiring individual platform components to implement their own access management was not going to serve as a suitable long-term strategy, especially as the stakes raise over time with respect to an increasing need for data security and integrity.

Kubernetes has a concept known as an “Ingress”, which is a centralised gateway for certain types of traffic, most often HTTP/S. Such a gateway can make use of the host name provided in received requests to redirect them internally to a handling service, if one exists. The ingress gateway can introduce middleware for various purposes, including authentication management, redirection and CORS (Cross-Origin Resource Sharing) handling. Such middleware can then intercept requests for processing dynamically altering requests (and responses) prior to forwarding. As well as this, ingresses can integrate with automatic TLS certificate provisioning via systems like cert-manager¹, reducing the operations security workload and minimising the risk of misconfiguration.

An series of ingresses have been created for the Ahuora Digital Platform, controlling access to the API, front-end and the authentication service. Requests including `ahuora.org.nz` as the host are forwarded to the front-end service, and likewise, those including `api.ahuora.org.nz` to the API service, and `auth.ahuora.org.nz` to the authentication service. Any HTTP (port 80) or HTTPS (port 443) traffic sent to the external IP addresses of cluster is monopolised by the ingress controller, which in K3s, is the designated to be the Traefik² reverse proxy. Traefik uses the ingresses defined for each Ahuora platform service to perform traffic forwarding.

4.4.4.1 Front-end ingress

The front-end uses one custom middleware chain, consisting of the “oauth-forwarder” and “oauth-signin”. The former is responsible for sending requests to the authentication service to check for

¹cert-manager: <https://cert-manager.io/>

²Traefik: <https://traefik.io/traefik/>

valid credentials, and the latter uses the preceding HTTP response to decide whether to redirect the user to the authentication service’s sign-in page.

4.4.4.2 Django API ingress

Requests made using the API FQDN are similarly processed by middleware, but instead by “django-cors” (the aforementioned CORS handler) and then “oauth-forwarder”. The CORS handler defines the allowed HTTP methods, headers, cookies, host names and source origins that can be used while interacting with the API.

The Django API does not use the same authentication middleware chain used by the front-end, as an unauthorised API call should simply be rejected instead of redirected, and API clients are not capable of interacting with a sign-in page in any case.

4.4.4.3 Authentication ingress

4.4.5 Authentication

4.4.6 Postgres database

4.4.6.1 Postgres

4.4.6.2 Connection Pooling

4.4.7 Monitoring

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.