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

List of Figures

List of Tables

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.

Research questions:

- How can a local deployment of an existing process simulation platform be migrated to a distributed environment?
- What impact on application performance does a distributed environment have?

1.2 Background

1.3 Summary of literature review

Chapter 2. Background

2.1 Project Alignment within Collaborative Work

2.2 Kubernetes

Kubernetes is a core component of the research and development described in this work. Kubernetes is a container management platform designed for handling the entire lifecycle of containerised workloads within distributed (multi-host) environments, from scheduling and operation, to termination. Some of the key terms related to Kubernetes and its constituent parts are defined.

2.2.1 Containerisation

Containers are units of software that contain all the necessary dependencies and configuration required for a piece of software to run. A containerisation approach to software development and deployment allows engineers to avoid an entire class of software error related to misconfiguration and missing dependencies. Containers have isolated filesystems that appear to the contained processes as a standard filesystem within an operating system.

2.2.2 Pod

Containers are a critical part of Kubernetes. However, within Kubernetes, the most granular runnable software unit is instead called a “pod”. A pod can manage several containers, which are treated as one element operated on by the cluster control plane, including initialisation, termination and scaling actions.

2.2.3 Deployment

A deployment object defines the container image to run within a pod, ports to expose, environment variables and secrets to consume, healthcheck endpoints to poll, and more. While specific pods are not guaranteed survival within Kubernetes, deployments will attempt to ensure that the defined pods remain available. For example, if a pod is killed or deleted, the control plane will detect that the inconsistency of the corresponding deployment's "desired" state with its actual state, and attempt to schedule a replacement pod as part of that deployment.

2.2.4 Service

Pods running within a cluster use private IP addresses allocated to them by the cluster control plane. As such, workloads that are intended for public access are, by default, not accessible from hosts outside of the cluster. Pods that attempt to communicate with one another need to know the IP address of the pod prior. A service object can handle both of these problems. ClusterIP, the base mode of operation for services, allows for a a single IP address and associated DNS name to be used to access a set of pods. Other operation modes are built on top of ClusterIP mode, which allow for external traffic to access software running on the cluster.

2.2.5 Ingress

An ingress is a centralised gateway for certain types of traffic arriving at a cluster, 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-manager[1], reducing the operations security workload and minimising the risk of misconfiguration.

[1]cert-manager: `https://cert-manager.io/`

Chapter 3. Methodology

3.1 Kubernetes distribution

With the intent of using Kubernetes as the distributed systems platform to build the Ahuora Digital Platform on top of, a specific distribution of Kubernetes still needs to be chosen. In general, engineers can choose between managed versions of Kubernetes, which are controlled by cloud providers (such as Amazon Web Services or Microsoft Azure), or self-managed versions, which require more up-front effort to provision a functioning cluster.

Managed versions of Kubernetes do not currently suit the needs of the platform at this time, as there are high ongoing costs associated with accessing a managed cluster service, as well as the usage of cloud-based virtual machines. Moreover, cloud service pricing can be less predictable and controllable than that of a physical environment where all the hardware part of the system is controlled directly. With a set of Raspberry Pis available for running a cluster, a Kubernetes distribution suited to a self-hosted approach is ideal. Another requirement for the distribution choice is for its installation and set-up process to be compatible with an airgapped networking environment (see 4.3.1).

There were three distributions of Kubernetes considered for selection: minikube[1], MicroK8s[2] and K3s[3]. minikube is oriented towards local Kubernetes cluster development, and enabling a quick cluster set-up process. MicroK8s presents itself as the easiest lightweight Kubernetes distribution to install, with an emphasis on simple addon integration for expansion of cluster behaviour and functionality. K3s specifically targets edge and small-scale computing environments, and is optimised for ARM CPU architectures.

minikube, while a useful tool, was discounted from consideration for the bare metal, multi-node Raspberry Pi cluster environment. Its reputation is that of a development tool, and not for running production workloads. MicroK8s is a more viable option for production environments compared

[1]minikube: `https://minikube.sigs.k8s.io/docs/`

[2]MicroK8s: `https://microk8s.io/`

[3]K3s: `https://k3s.io/`

to minikube, and this is also true of K3s. However, three key factors made K3s the final choice: its explicit curation towards an ARM64 CPU architecture, convenient packages for an airgapped installation, and an officially-supported Ansible installation template.

3.2 Kubernetes cluster architecture

3.3 Performance analysis experiment components

3.3.1 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 exarcerbated 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 miiliseconds 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
```

3.3.2 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, has 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.

3.3.3 Empirical comparison

3.3.3.1 Grafana k6 load-testing

Grafana k6[1], 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

[1]Grafana k6: https://k6.io/

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 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.

3.3.3.2 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.

3.4 Load test experiments

3.4.1 Benchmark environment

A reference point to compare collected test metrics against is needed to obatin 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[1], which allows a set of containerised software components to be configured and run with 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.

3.4.2 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.

[1]Docker Compose: https://docs.docker.com/compose/

3.4.3 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. 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 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.

3.4.3.1 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

3.4.3.2 Post-trial metrics collection

For tests remotely run against the cluster, metrics internally generated and uploaded to Grafana Cloud by the cluster (via Grafana Alloy, see 4.4.7) 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.

Cluster-level metrics are retrieved and collected at one-minute intervals. This period may be limiting for observing fine-grained usage trends, and normally this interval would be decreased, but there is a financial limitation in place. Grafana Cloud provides 10,000 free metric series, and current usage within the cluster lies between 6,000 and 7,000 series in use. Decreasing the metrics collection interval by half would double platform usage, and extend into the paid tier, which is not viable at this time.

3.4.3.3 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 3.2: Base options set for k6 load tests

3.4.3.4 Load profile configuration

Load profile	**UOR Load (requests/second)**	**FS Load (requests/second)**
Average	15	1
Stress	30	2
Spike	80	8
Breakpoint	200	32

Table 3.3: Load profile values for API endpoints

3.4.3.5 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.

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

3.4.3.7 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.

3.4.3.8 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 deterioates 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.

3.4.4 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.

The `HorizontalPodAutoscaler` settings used in these tests will have the scale-down stabilisation window set to 30 seconds to allow the autoscaler to reset quickly inbetween tests. The default for this window is 300 seconds, which limits the scaling action instability (pod replicas rapidly increasing and decreasing unnecessarily), as the autoscaler will look at all CPU resource usage within the past 300 seconds and take the maximum utilisation value to inform scaling actions. Setting it to 30 seconds will still provide some stability, but avoid long cooldown periods.

3.4.4.1 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 3.4: 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

Table 3.5: Parameters for FS replica count tests

3.4.4.2 Resource allocation and target utilisation

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.

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.

These tests will assess the impact of the permutations of three CPU resource allocation settings and four target utilisation thresholds on response time and throughput. The number of Django API and IDAES replicas over time will be observed.

Load profiles used: Spike, breakpoint

Test iteration	**CPU allocation (millicores)**	**Average CPU utilisation target (%)**
1	500	25
2	500	50
3	500	75
4	500	90
5	1000	25
6	1000	50
7	1000	75
8	1000	90
9	2000	25
10	2000	50
11	2000	75
12	2000	90

Table 3.6: Parameters for resource allocation and target utilisation tests (UOR and FS)

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) mnemomics.

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[1], 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.

ZeroTier[1] 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.

[1]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*[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 tunnel[2] 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

[1]Container image registry: https://github.com/mcronce/oci-registry

[2]Cloudflare 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. Alongside this, the latency when using the tunnel instead of allowing direct connections is higher, causing slower interactions with the Ahuora Digital Platform. 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.

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.

WSGI, or Web Server Gateway Interface, is a standard that allows dedicated web servers to interact with a Python application, absolving the target application of the responsibility of HTTP request management. During tests, it was discovered that Granian, the web server used to run the Django API WSGI application, was not providing graceful closure of active connections when the container received a SIGTERM (terminate) signal from the cluster. This meant active requests made to the terminating API instance were simply being closed, instead of being allowed to finish within the default graceful shutdown period of 30 seconds provided by the cluster. From the perspective of the testing client, these requests failed to complete. To resolve this, a "preStop" lifecycle command was added to the Django API container, which simply sleeps for five seconds. When any preStop command is defined, the cluster will wait for that command to finish before signalling the container with SIGTERM, providing a window for requests to finish.

Another problem found through testing was a phenomenon where, after every autoscaling interval where more pod replicas were created, the number of HTTP requests waiting to be processed and the average response time suddenly spiked, and then quickly came down. It was determined that Granian lazily loads the target WSGI application; it does not load the application until the first request. The load time can range between 500 and 1000 milliseconds. As such, the response time for any requests arriving during this lazy initialisation period will be much higher than subsequent requests, explaining the brief spike in request queue length and response time. To mitigate this issue, a startup probe was added, which polls the `/api/status/` endpoint for a successful response. Until the probe succeeds once, the pod will not be considered ready, and the Django service will not forward any traffic to the pod until it becomes ready. This way, external requests will not be affected by the initialisation time of Django API 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 Vite[1] 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.

[1]Vite: https://vitejs.dev/

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.

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 entrypoint 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.

A 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 Traefik[1] reverse proxy. Traefik uses the ingresses defined for each Ahuora

[1]Traefik: `https://traefik.io/traefik/`

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 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. If the front-end is unauthorised, when it tries to make an API call, it will not be able to detect API authorisation failure, as the redirection would cause the sign-in page to “load” successfully, with an HTTP 200 response. The client would instead error because of an expected content type (HTMl instead of JSON). Returning an authorisation error resolves these problems.

4.4.4.3 Authentication ingress

Unlike the prior ingress definitions, the authentication ingress does not use any middleware or block clients from accessing the service. The authentication service should be available to any client at all times to allow for them to attempt the authentication flow. This ingress simply forwards traffic to the internal authentication service.

4.4.5 Authentication

As mentioned prior, a central solution for platform access management was needed to prevent platform components from taking on repeated security responsibilities. A third-party authentication service called OAuth2 Proxy was deployed to the cluster for managing validation of all access to the platform, and the corresponding authentication flows.

OAuth2 Proxy allows for integration with a range of providers that implement OAuth-compliant authentication protocols. This includes entities such as Google, Microsoft, and GitHub. For the Ahuora Digital Platform, Google was chosen to initially integrate with as an authentication provider, though additional providers will be enabled in future. To accomplish this, an OAuth application had to be created within a new Google Cloud account. OAuth2 Proxy then had be to provided with Google Cloud service account credentials to permit OAuth API calls. These secrets were injected into the service using the Kubernetes secret construct.

To specify the users permitted to access the platform, a list of authorised emails was created and injected into the service. When a user attempts to sign in with a Google account, the verified account details are returned by Google to OAuth2 Proxy, where the email is then checked against the list, and denied or allowed access depending on whether it is present on the list.

4.4.6 Database

The platform relies on the usage of a Postgres database for storing all data generated by interactions with the platform. The database, however, needs to be deployed in a different fashion to the other platform components.

4.4.6.1 Postgres provisioning

With respect to state management in cloud-native software, there are two broad classes that applications belong in: stateless and stateful. The Django API, front-end and IDAES service are all stateless applications; they do not (directly) rely on the persistence of data to host or network storage to operate. However, the needed Postgres database is stateful, and does require a consistent view of the data it writes and reads to storage. This demands the usage of a different Kubernetes construct: the StatefulSet.

StatefulSets[1] ensure that deployed pods are provided with consistent identities that stays with them across rescheduling or failure events. Pods are allocated consistent storage provided by a PersistentVolume, which abstracts storage provisioning details away from the dependent (storage may be sourced from the host, the network, or a cloud provider, for example). Pods are accessed via individual virtual IP addresses pointed to by a "headless" service, which does not use a single IP address to control traffic.

The platform's Postgres database is deployed using such a StatefulSet, with one pod replica. This pod constructs a PersistentVolumeClaim to make a storage request from a PersistentVolume tied with a 5 GiB storage block on a specific cluster node. Without NAS (network-attached storage) available, it was necessary to assign a PersistentVolume to a specific node, otherwise pod restarts could cause the storage to be allocated randomly to another node within the cluster, and any previously available data would be missing.

The secret management approach is the same as for other platform components; the Postgres password is injected as an environment variable from a Secret resource.

4.4.6.2 Connection pooling

Within Postgres, there is a limit to the number of connections to the database that can be active at any given time. By default, this value is 100, though it can be modified as a start-up argument. As soon as this limit is exceeded, the database will return errors to clients attempting to create these excess connections. This hard limit would require complex retry logic for every point in application code that a SQL query is made against the database, as there is always a risk that an error may be returned under periods of increased load.

To handle this problem, there is a technique known as "connection pooling" that is used. Connection pooling involves software that manages the use and recycling of connections to a database instance (or several). There may be a fixed pool of connections that are initially established with the target database, and when a client wants to create a new connection, they are provided with an existing connection from the pool. Once the client has finished using a connection, it is returned to the pool. A pool may be configured so that excess connection attempts are queued instead of rejected, creating

[1]StatefulSet: `https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/`

better query resilience by default, even if it introduces a waiting period.

Connection pooling can be performed at the application level (within the same process as the SQL clients) or the software level (a separate system that is purposes for connection pooling and management). Because of the replication of Django API instances that may take place within the platform, having a dedicated Postgres connection pooler system is more appropriate. Configuration of pooling rules can be centralised and made independent of SQL clients, creating consistent connection management regardless of the number of clients. A software system called PgBouncer is used to fill this role.

Instead of the Django API connecting directly to the Postgres database, it connects to PgBouncer instead, which in turn uses a fixed number of connections to query the database. PgBouncer can allow many more clients to connect to it than there are actual connections available. While the average query response time will increase when there there is a high ratio of clients to connections, this is managed gracefully.

4.4.7 Monitoring

Within any distributed computing system, it is critical to maintain good knowledge of how the system is operating at several levels of granularity, whether that at the level of an individual application, a participating node, or the system overall. This involves having access to data reporting resource usage, health statuses and various logs that work together to describe the state of the system. An absence of these things makes it far more challenging to diagnose the system when (not if) things go wrong.

To effectively monitor the Ahuora Digital Platform, as well as the wider cluster, a monitoring package called k8s-monitoring-helm was deployed. This package includes several software components, including Grafana Alloy, Kube State Metrics, and Node Exporter. Kube State Metrics and Node Exporter are responsible for collecting cluster and node-level metrics respectively, while Grafana Alloy aggregates these and all other available metrics along with recorded cluster events and logs, filtering them based on a predefined set of Prometheus rules. Every minute, this aggregation and filtering process is performed, and the resulting dataset is uploaded to Grafana Cloud for storage and viewing.

Some custom Prometheus filtering rules were created to allow filesystem activity on the specific storage devices used within the cluster to be captured, as they were initially ignored by Grafana Alloy because of the default ruleset.

4.5 Kubernetes deployment automation

Every non-trivial cloud-native software system can be better managed when automation is introduced to the management of critical processes. One of these processes is the deployment and release process of software to a system. The Ahuora Digital Platform is made up of and relies on several custom and third-party software components. The reliable management of these components necessitates the usage of automation through release pipelines and upstream version checks.

4.5.1 Argo CD manifest synchronisation

Argo CD[1] is a "continuous delivery tool for Kubernetes". It targets a GitOps approach to Kubernetes cluster configuration and deployment, where all configuration for a cluster and its deployed software is tracked and managed through source-control. Changes to configuration in source-control should result in corresponding changes on the live cluster.

Argo CD is used within the Ahuora Digital Platform to keep track of changes made to the deployment definitions for the Django API, front-end, IDAES worker service and PostgreSQL database. Each deployment and service manifest (as well as other Kubernetes objects) is stored within a git repository purposed for cluster configuration. Argo CD was provided with credentials to read the contents of the repository, where every three minutes, it retrieves any upstream changes and compares them to the state it has cached locally. Modifications to tracked manifests are applied to the cluster through a synchronisation. This also allows changes that have not been persisted through source-control to be reverted automatically, ensuring consistency between the live state of the cluster and the repository.

The core resource used by Argo CD is the `Application`. An `Application` consists of all the resources related to a specific system, such as the core Ahuora Digital Platform components.

[1]Argo CD: `https://argo-cd.readthedocs.io/en/stable/`

In addition to the core Ahuora `Application`, another was defined for the `k8s-monitoring-helm` system bundle, but instead specifying its resources as the corresponding upstream Helm chart that templates all the manifests and resource definitions for the package.

4.5.2 GitHub Actions platform release pipeline

While Argo CD is responsible for tracking changes made to manifests, including version changes of container image dependencies, a fully automated release pipeline still requires version changes to be automatically applied to manifests. To establish this, several GitHub Actions workflows were created, involving both the core Ahuora platform and cluster configuration repositories.

4.5.2.1 Container image build processes

A generic container image build and publishing process was defined as a GitHub Actions workflow. This workflow consists of three "jobs" or stages.

The first is responsible for version-tagging the image with the Semantic Versioning[1] (Semver) format (e.g. 0.1.2) based on the prefix of the latest git commit. If a Semver tag has already been generated previously, the appropriate version component will be incremented by one.

The next stage builds a container image using Docker, adding a container tag based on the previously created git tag. As the Raspberry Pis making up the cluster use an ARM64 CPU architecture, the images are built for ARM64. The images are built using instructions provided in Dockerfiles that exist within the project directory of each software system. After completion of the build, the image is uploaded to Docker Hub for storage, within an Ahuora-controlled Docker Hub account.

The final stage makes an API call to trigger a workflow defined in the cluster configuration repository, making use of credentials provided by a custom GitHub App that has access to both repositories (workflows in one repository typically do not have any access to those in others).

This generic workflow is then re-used by workflows specific to the Django API, front-end and IDAES service. Each of these concrete workflows provides four parameters to the generic workflows: the name of the branch to check out, the prefix to add to version tags (e.g. "django-api"), the name to use for the published image, and the directory to find the corresponding Dockerfile definition. To

[1]Semantic Versioning: `https://semver.org/`

avoid issues with concurrent image tagging and publishing, a concurrency rule is added for each of these workflows that prevents multiple runs at once for the same workflow on the same branch.

4.5.2.2 Automated manifest version updates

In response to new image versions published to Docker Hub, Kubernetes manifests should be automatically updated to use these new versions, which can then be identified by Argo CD for pulling downstream.

An application called Renovate[1] exists for this very purpose. Renovate can check for updates to depedencies for many different package types, including Kubernetes manifests. In the cluster configuration repository, Renovate settings were added to require a Renovate bot to check for upstream version changes to manifests in the Ahuora manifest directory (belonging to the Ahuora Argo CD `Application`). When changes are detected, Renovate will create a pull request for each version change. Docker Hub credentials had to be provided to Renovate to check for image version changes, as each of the Docker Hub image repositories is marked as private.

Initially, Renovate was added as a third-party GitHub app to the repository. This allowed Renovate to manage the dependency check scheduling as well as secret injection, reducing the need to create a custom GitHub Actions workflow. However, when the need for triggering a dependency check arose to allow for full release pipeline integration, the lack of a trigger mechanism called for a move to manual usage of Renovate.

The new workflow is scheduled to run every hour, but it can also be triggered via an API request, as previously mentioned in 4.5.2.1. Since several images can be built and published concurrently, it is possible that several workflow runs could be triggered. Only one Renovate dependency check should attempt to update manifest versions at a time, lest duplicate pull requests are created and automatic merges are halted. A concurrency group is used here to prevent concurrent runs.

The final step in enabling full pipline automation was to configure the repository (and Renovate) to allow for automatic merges. By default, a developer has to explicitly request for a pull request to be merged into the target branch. With automatic merges, a tool, such as Renovate, can mark a pull request as auto-mergeable, where then GitHub will manage the merge process without further interaction.

[1]Renovate: `https://docs.renovatebot.com/`

With all this in place, it is possible for new versions of platform software to be merged in the Ahuora platform repository, have the corresponding container images built, update the referenced versions in their dependent Kubernetes manifests, and then update the cluster to use the latest versions. The entire process following a merge is now automated.

Chapter 5. Results and Discussion

5.1 Performance analysis

5.1.1 Key metrics

There are three key metrics that will be used to assess system performance impacts: the HTTP response time, throughput, and 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 inner managed system, or to the outer 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.

5.1.1.1 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.

5.1.1.2 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.

5.1.1.3 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.

5.1.2 Secondary metrics

5.1.2.1 Container-level resource utilisation

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 or container at the same time as observed request rate instability would possibly indicate the pod is reaching computation limits.

5.1.2.2 Cluster-level resource allocation and utilisation

Especially when auto-scaling policies are in place, it is necessary to track the number of deployed pods for each active cluster deployment. This metric can then be used to calculate the sum of all provisioned resources on the cluster, such as the total requested CPU. With this information, the level of over or under-provisioning can be compared across different workloads, and across workload configuration variations. If a pod requests 2000 millicores of CPU, but only uses 100 millicores on average, then this is a clear sign of severe over-provisioning. Likewise, requesting 500 millicores but quickly reaching an average of 90% CPU utilisation would indicate under-provisioning. With auto-scaling policies in place, poorly set CPU or memory requests can rapidly multiply into cluster-wide resource allocation issues, such as additional pods being unschedulable.

5.1.3 Summary statistics

Using the data generated within tests, a number of summary statistics for each test result can be derived to assist with immediate comparison of deployment configurations.

- **Average (mean) request completion rate**: The mean number of requests that were successfully completed per second.
- **Maximum request completion rate**: The peak request completion rate across the test duration.
- **Maximum request start rate**: The peak number of requests started by the k6 test client per second.
- **Average (median) response time**: The median time taken for a request to receive a response.
- **Overall error rate**: The proportion of requests across the test that failed due to an HTTP error or client timeout.
- **Performance degradation threshold**: The point at which the response time exceeds a certain level (100 milliseconds for UOR tests, and 1000 milliseconds for FS tests). This statistic allows for an approximation of the request start and completion rates where system performance begins to degrade rapidly. This threshold will not be calculated for average load or stress load tests.
- **Maximum started and completed request ratio**: The largest ratio between the number of requests that have started and the number that have completed. A high ratio indicates that the system is failing to process as many requests as it is receiving. This will not be calculated for breakpoint load tests.
- **Maximum cluster CPU request proportion**: The peak proportion of available cluster CPU used by a deployment. This will be calculated for the Django and IDAES pods only, and only for the resource allocation and utilisation tests.

5.1.4 Rolling window statistics

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.

5.2 Unit operation retrieval (UOR) experiment results

5.2.1 Benchmarks

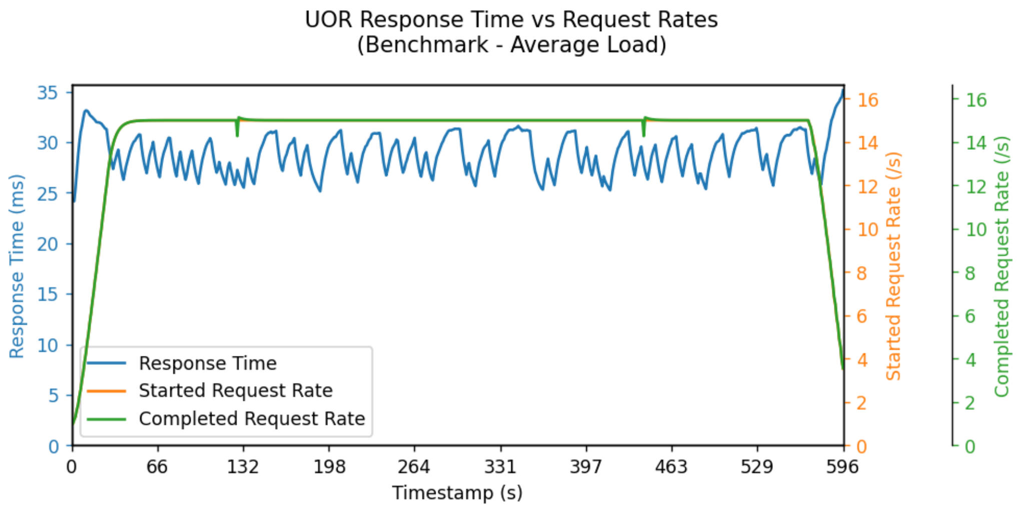


Fig. 5.1: UOR response time vs. request rate graph - average load benchmark

Both response time and request rate degradation can be observed in the local spike test (Fig. 5.3). After reaching ~40 requests per second, the response time rapidly increases, and the request completion rate starts to diverge from the request start rate. At 45 requests per second (RPS), the response time average increases past 100 milliseconds, and reaches tens of thousands of milliseconds as the request rate continues to increase. During the test, the request start rate is also seen degrading (between 135 and 189 seconds). This is because of the virtual user limit of 2000 set on the k6 test client, which was configured to prevent system resource starvation by the test client. The response time does not appear to recover towards the end of the test. Approximately 0.036% of requests in the spike tests failed, while neither the average or stress load tests had any failed requests.

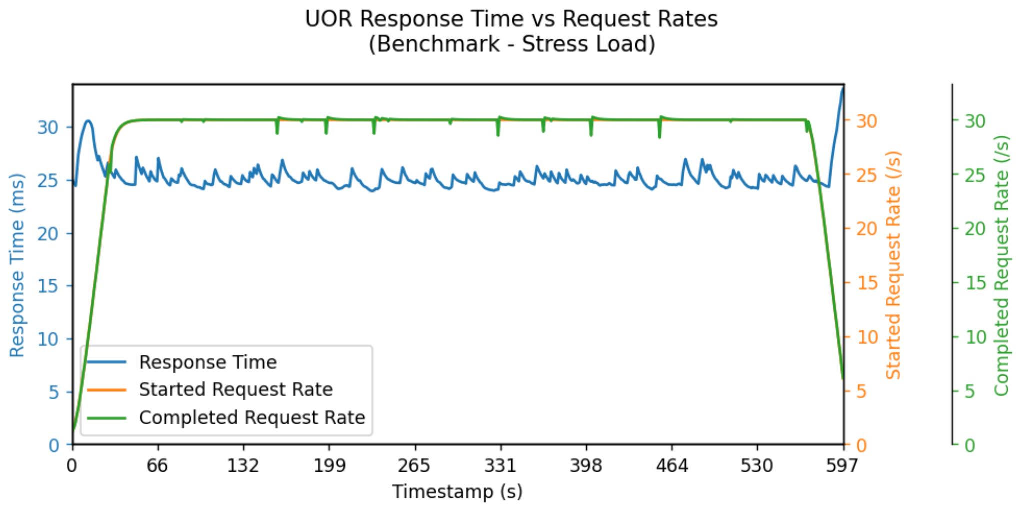


Fig. 5.2: UOR response time vs. request rate graph - stress load benchmark

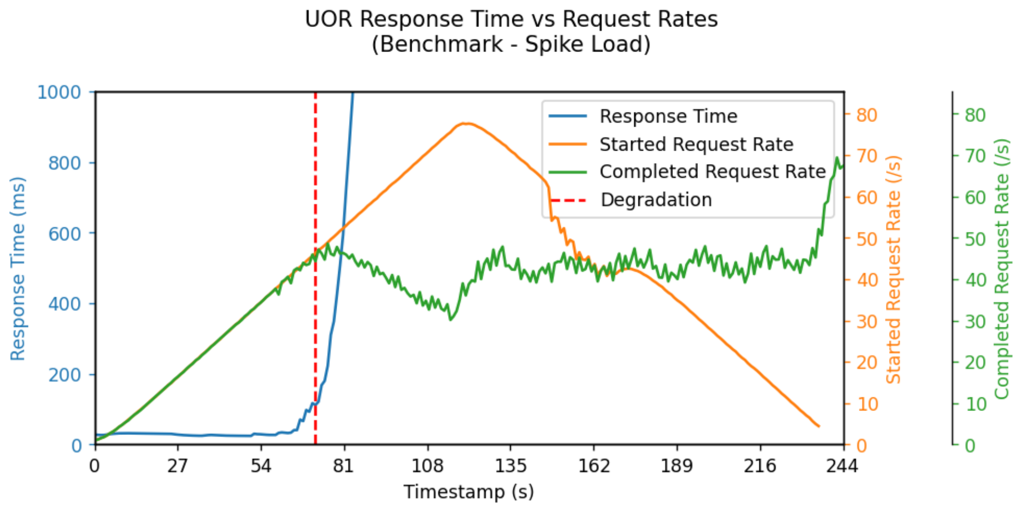


Fig. 5.3: UOR response time vs. request rate graph - spike load benchmark

The breakpoint test (Fig. 5.4) also identifies this same 45 RPS threshold, beyond which the average response time exceeds 100 milliseconds, and the request completion rate also diverges. No requests failed in the breakpoint tests, but this in part due to the early termination threshold used by breakpoint tests.

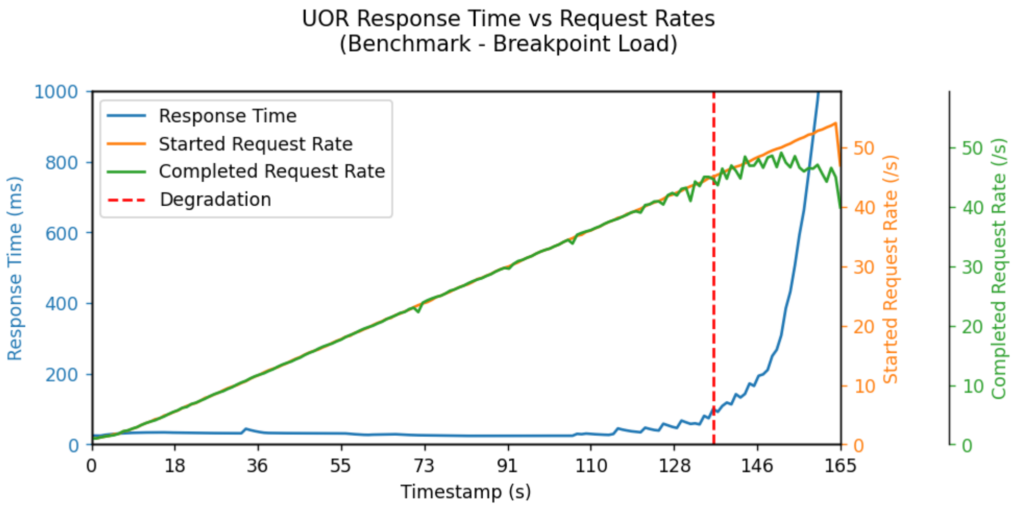


Fig. 5.4: UOR response time vs. request rate graph - breakpoint load benchmark

5.2.2 Replica count

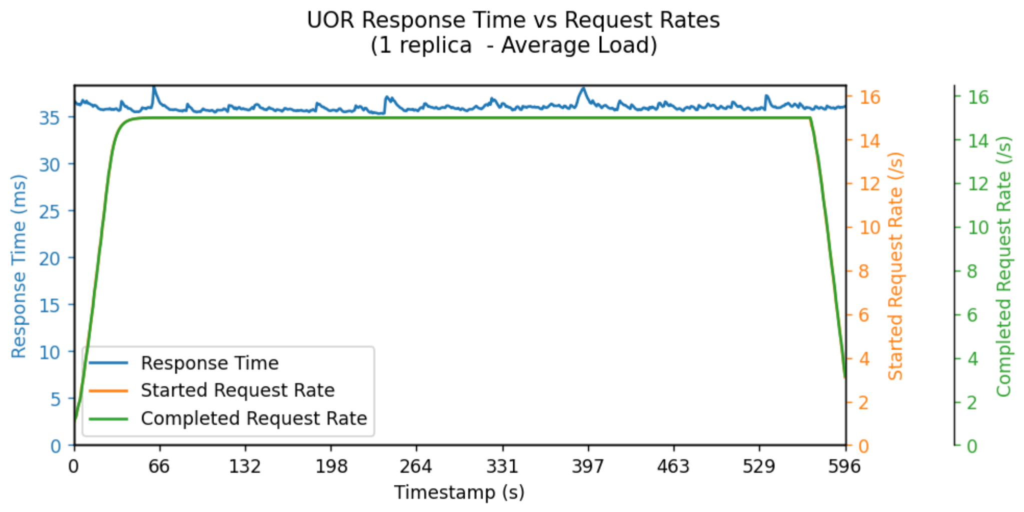
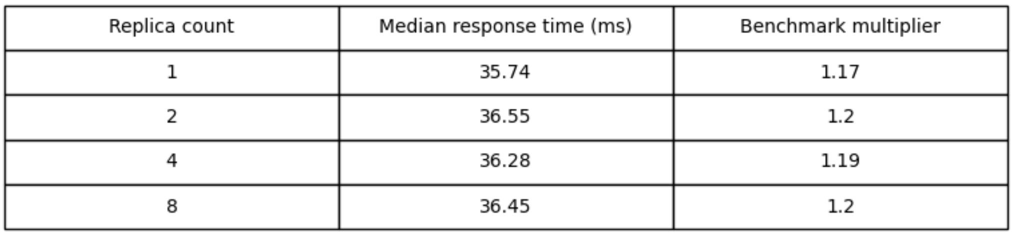


Fig. 5.5: UOR response time vs. request rate graph - breakpoint load with one replica

Using an average load profile against a Django deployment configuration with one replica allows some differences to be observed between the average load benchmark and this cluster-bound test. While the response time of the benchmark sits between 25 and 30 milliseconds, the single replica test sees a median average response time of 35.74 milliseconds (Fig. 5.5).

This higher response time average is consistent with any number of replicas (as seen in Fig. 5.6) when average load is applied. The simple explanation for this difference is the use of TLS in the



Replica count	Median response time (ms)	Benchmark multiplier
1	35.74	1.17
2	36.55	1.2
4	36.28	1.19
8	36.45	1.2

Fig. 5.6: Table of median response times by replica count, average load

HTTPS connections used between the test client and the cluster, which may take between 5 and 10 milliseconds per request. Unencrypted HTTP is used to access the local deployment, so less work is required to establish a connection. As well as this, the 5 millisecond artificial delay added for cluster-bound requests partially accounts for this discrepancy. A similar range of averages is observed within the replica count stress tests, albeit with a larger gap between the benchmark and replica count median response times (due to the lower average achieved by the benchmark in stress tests compared with the average load tests.)

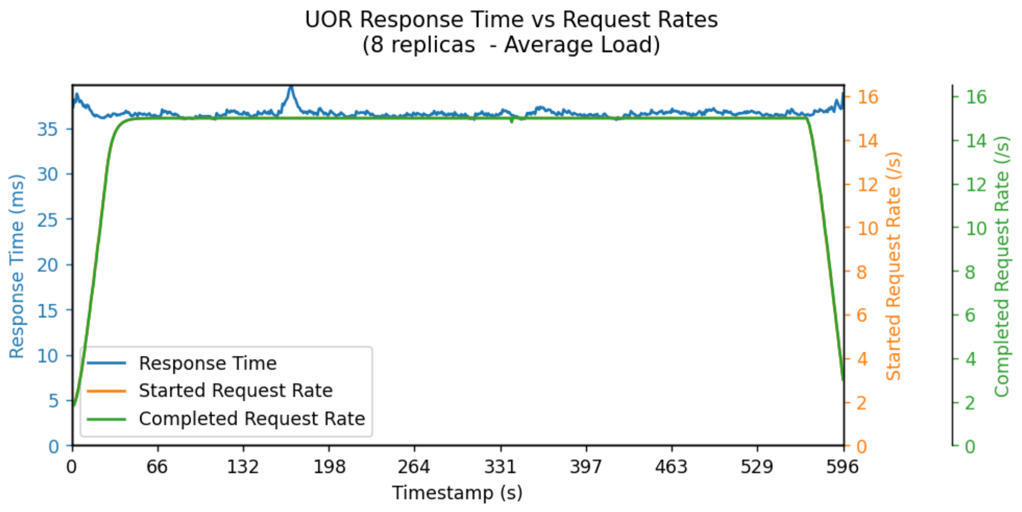
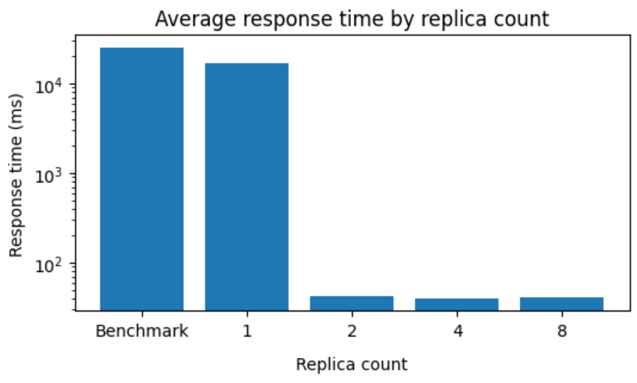
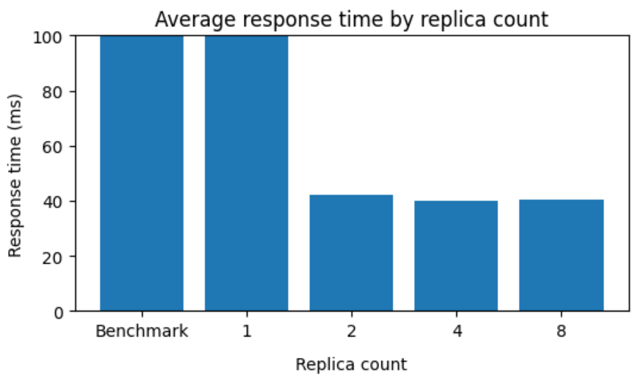


Fig. 5.7: UOR response time vs. request rate graph - breakpoint load with eight replicas

Regardless, any tested number of Django replicas running on the cluster is able to process incoming requests as fast as they arrive (evidenced in Fig 5.5 and 5.7), so there are no significant request rate differences between the benchmarks and the replica count experiments. However, where the benchmarked local deployment shows evidence of performing worse than the clustered deployment is when testing spike and breakpoint loads with more than one replica. As shown in Fig. 5.8, a replica count of one performs similar to the benchmark, where the median response time reaches almost 17,000 milliseconds. The median response times for the remaining replica counts (2, 4 and 8) are magnitudes lower, ranging from 39.89 to 42.14 milliseconds.



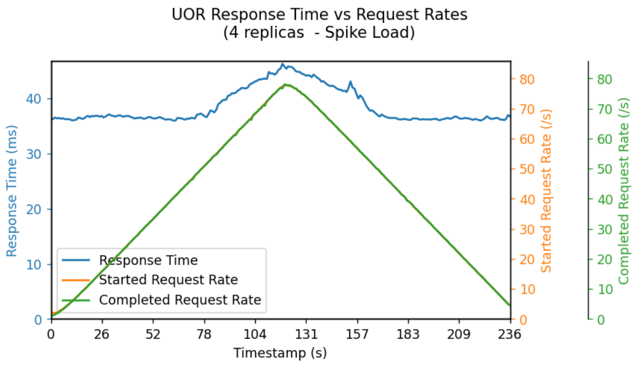
(a) Logarithmic scale



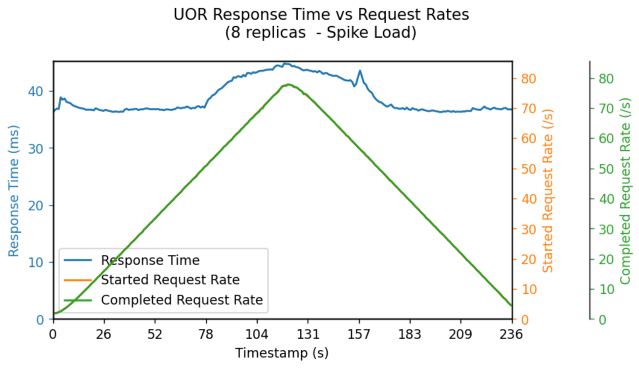
(b) Linear scale

Fig. 5.8: Median response times by replica count (spike load)

With either four or eight Django replicas, the average response time increases by almost 30% at the peak request arrival rate (Fig. 5.9), but the system is otherwise able to process these requests at the arrival rate. Early signs of degradation are seen when using two replicas, with the response time sharply doubling at the peak request rate, though both request rate curves are mostly consistent 5.10. Interestingly, the response time curve dampening effect with four replicas is highly similar to that with eight replicas, despite doubling the number of available workers. With one replica, this is not the case, having a similar request completion curve (Fig. 5.10) to the benchmark (Fig. 5.3), along with an 8.79% request fail rate, which is 246.35 times worse than the benchmark. The usage of one replica results in the same degradation request rate as the benchmark. None of the other replica count variants have a degradation request rate. When it comes to the started and completed request ratio, a single replica sees a maximum ratio of 1.34 started requests to completed requests, whereas the other replica counts all reach no more than 1.02.

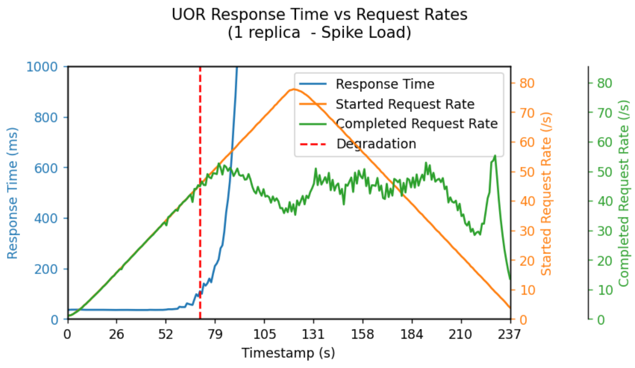


(a) 4 replicas

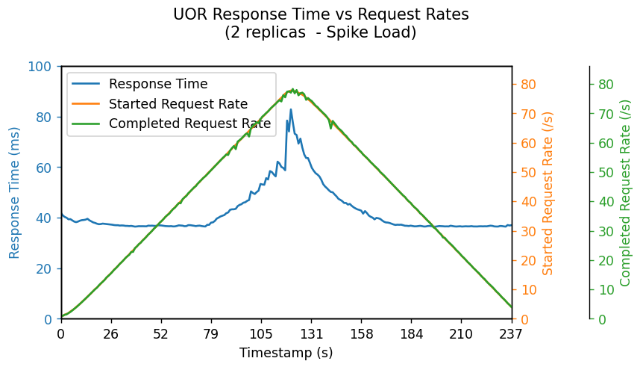


(b) 8 replicas

Fig. 5.9: Response time vs. request rate graph (spike load, 4 and 8 replicas)



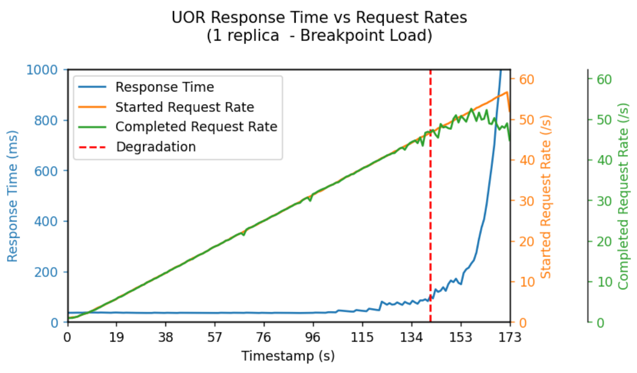
(a) 1 replica



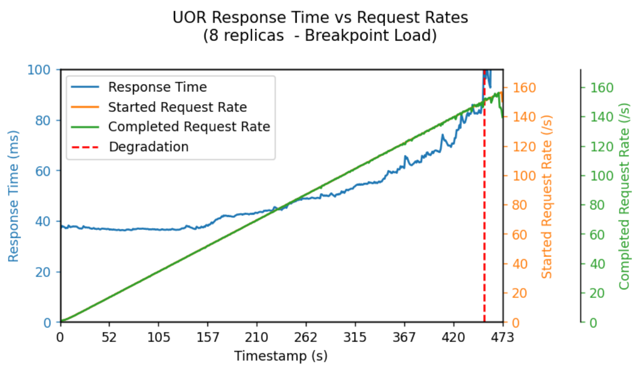
(b) 2 replicas

Fig. 5.10: Response time vs. request rate graph (spike load, 1 and 2 replicas)

Within the breakpoint tests, there is a significant distinction between the maximum request start rate reached across the lowest and highest number of replicas. As visible in Fig. 5.11, an eight-replica Django deployment reaches ~150 started RPS (3.36 times the benchmark) before the response time exceeds 1000 milliseconds. On the other hand, the single replica deployment passes the degradation threshold at ~35.3 started RPS. Over the course of the eight replica breakpoint test, the response time gradually increases with the request rate, becoming more unstable over time. Fig. 5.12 shows that a start RPS of 158 (2.84 times the benchmark) is the maximum reached by the eight-replica deployment. The four-replica deployment reaches a slightly smaller limit of 157 RPS, though it has a lower degradation request rate of ~136.4 RPS (Fig. 5.13).



(a) 1 replica



(b) 8 replicas

Fig. 5.11: Response time vs. request rate graph (breakpoint load, 1 and 8 replicas)

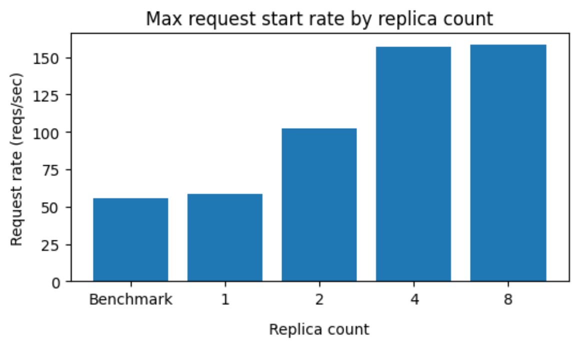


Fig. 5.12: Chart of maximum request start rates by replica count

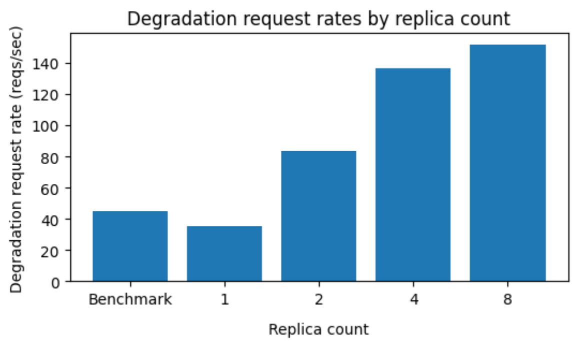
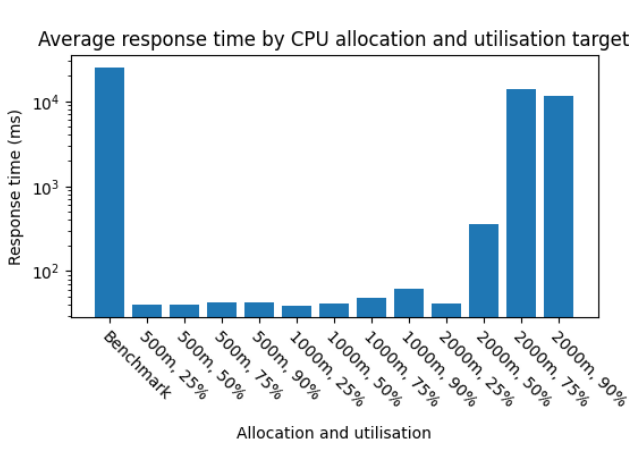


Fig. 5.13: Chart of degradation request rates by replica count

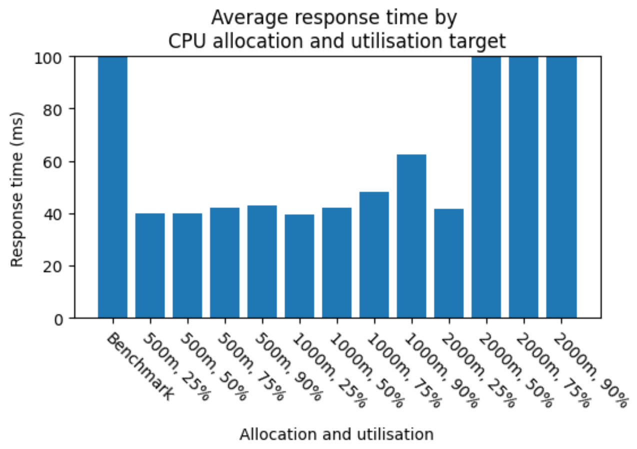
In general, a higher number of replicas allows for a higher degradation threshold, though when it comes to the maximum start RPS, the relationship does not appear to be as straightforward. The maximum start RPS only slightly increases from four to eight replicas, which indicates there may be a bottleneck elsewhere in the system preventing a higher maximum.

5.2.3 Resource allocation and utilisation

Results based on a static number of replicas have been showcased, but the effect of dynamic replica auto-scaling on key system metrics is also assessed.



(a) Logarithmic scale



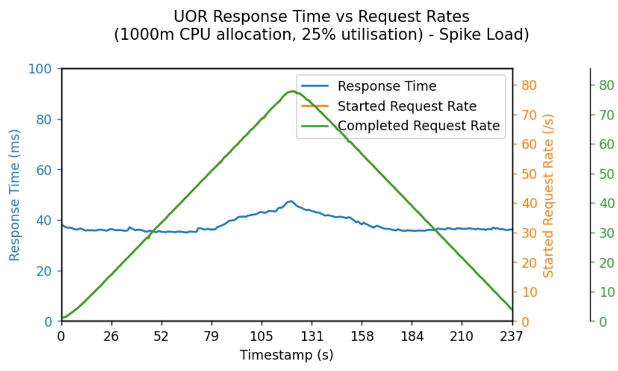
(b) Linear scale

Fig. 5.14: Median response times by CPU allocation and utilisation target (spike load)

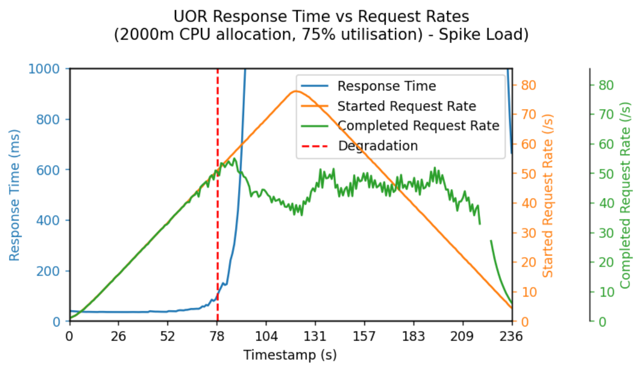
There are a number of response time patterns that can be observed in Fig. 5.14. The experiments configured to use 500 millicores of CPU per Django pod have relatively consistent response times, with the lowest median response time being 40.03 milliseconds when the target utilisation is set to 25%. The lowest response time is, however, claimed by the 1000 millicores and 25% utilisation combination. Within the 1000 millicore experiments, there is a clear increase in the median response time as the utilisation target is increased, rising to 62.62 milliseconds when set to 90%.

The 2000 millicore experiments perform poorly on average, save for when the target utilisation is at 25%. Their median response times range from 355.09 to 11,585 milliseconds. The response time versus request rate graph for the worst performing configuration (2000 millicores with 90% utilisation target) is seen in Fig. 5.15 (b). It hits the degradation threshold at 46.96 started RPS, and then fails to keep up with further incoming requests. Towards the end of this test, there are missing points for the completed request rate. Since the completed request rate is based on successful requests only, this gap is explained by a period in which all requests failed. In fact, 15.16% of requests failed with this configuration. When looking at the experiment with 1000 millicores and a 25% utilisation target, there is a noticeable increase in the response time at the request rate peak (as previously seen in the replica count experiments), but otherwise, no signs of performance degradation are present.

There are three experiments that did not encounter the 100 millisecond degradation threshold: 500 millicores with the 25% and 50% targets, and 1000 millicores at 25%. All other spike load experiments experienced degradation within a range of 16.87 to 49.73 milliseconds.



(a) 1000m CPU allocation, 25% target utilisation



(b) 2000m CPU allocation, 75% target utilisation

Fig. 5.15: UOR response time vs. request rate graphs (spike load)

An observation that can be made (Fig. 5.14) is that high utilisation targets tend to suffer from poorer response times than low targets, with this being exarcerbated when the CPU allocation is simulateneously high. There are several explanations for this. The first is that the Django API has not been configured as a multi-processing application; it does not make use of multiple cores. From this, it is plain to see why a CPU allocation of 2000 millicores results in the worst response time averages, as a utilisation target exceeding 1000 millicores (50% of 2000 millicores) is unattainable. Secondly, the horizontal pod autoscaler has a reactive rather than predictive nature. As the load on the system increases, previously set utilisation targets are eventually exceeded, and the autoscaler responds by scheduling more replicas in an attempt to meet the target. With higher target utilisation values, individual pods receive higher loads before additional pods spread the overall load further. A higher utilisation target may eventually lead to increased response times, as each API instance attempts to serve more requests at a time, potentially more than can be sustainably handled.

Another clear trend is the negative correlation between the CPU utilisation target and the maximum total requested CPU across all Django API pods. With smaller utilisation targets, the autoscaler schedules pods more frequently in response to increased load, and schedules more pods overall. This can be problematic, as the Kubernetes scheduler uses resource requests to determine whether a node has sufficient capacity to run additional pods, and with provisioning of heavily underutilised resources, nodes may be treated as at capacity, despite low actual resource usage. In this context, as depicted in Fig. 5.17, a 25% utilisation target with 500 millicores of CPU sees up to 6000 millicores of CPU requested across all Django pods, though only 1500 millicores (1.5 cores) is used in practice based on the target.

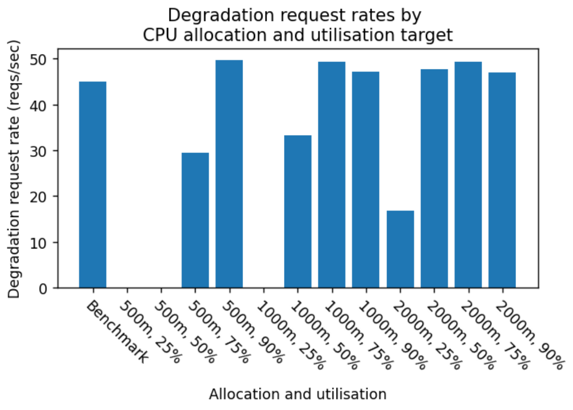


Fig. 5.16: Degradation request rates by CPU allocation and utilisation target (spike load)

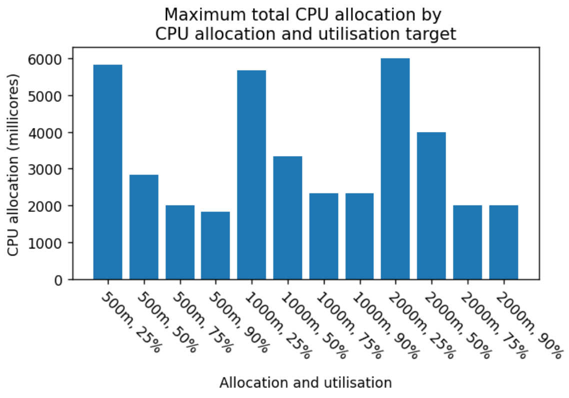


Fig. 5.17: CPU allocation by CPU allocation and utilisation target (spike load)

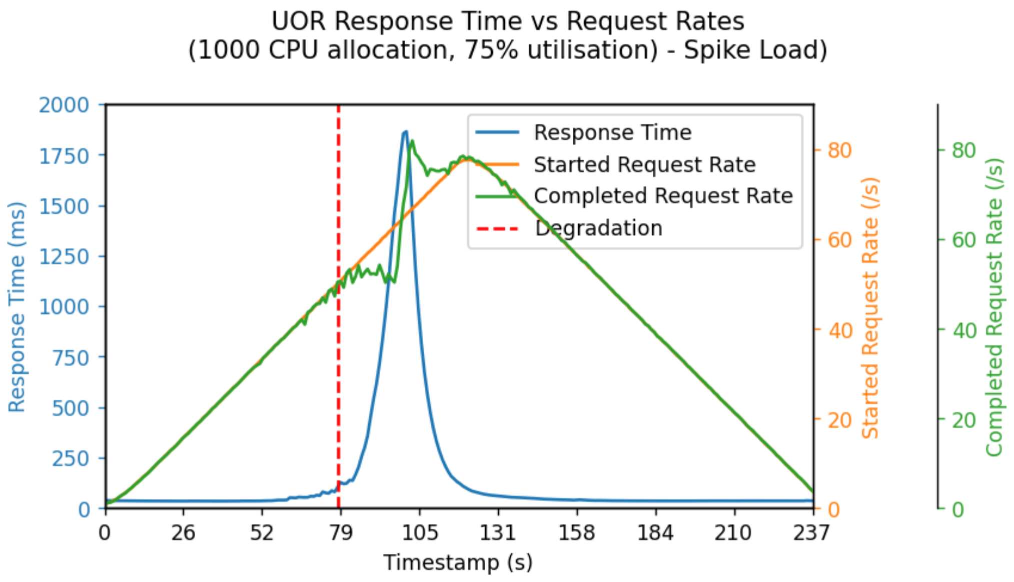


Fig. 5.18: UOR response time vs. request rate graph (1000m CPU allocation, 75% target utilisation - spike load)

5.3 Flowsheet solving (FS) experiment results

5.3.1 Benchmarks

5.3.2 Replica count

5.3.3 Resource allocation and utilisation

5.4 Impacts

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

5.5 Threats to validity

Chapter 6. Conclusion and Future Work

Bibliography

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