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

Modern chemical process simulation software systems have performance limitations caused by primarily operating on standalone machines. Using containerisation techniques, an existing process simulation platform is transitioned from a locally operating system to a distributed, Kubernetes-based system. A load-testing procedure is outlined for comparing the performance and system behaviours of a distributed deployment of the platform. Analysis is conducted on the system test results, comparing how various software scaling configurations react to different load profiles. The analysis reports that significant performance improvements have been obtained, but bottlenecks still exist that need addressing to further improve maximum system throughput.

Acknowledgements

I would like to give thanks to my supervisors, Mark Apperley and Tim Walmsley, for their guidance and support over the course of the capstone project, which has been invaluable in the completion of the many unfamiliar tasks that made up this project.

Thanks to my fellow software engineers at Ahuora: Shean Danes Aton, Bert Downs, Ethan MacLeod, Mahaki Leech and Alma Walmsley. Working with you over the course of this year has been an exciting, opportunity-filled, stressful, and pleasurable experience. Whether it was through moments of laughter or through frustration, I am glad to have had you all as co-workers.

My deep appreciation goes to Stephen Burroughs, who gave up much of his time (including thesis-writing time) providing much-needed help in moments of sheer panic and confusion, and continually over the course of this year.

To the master of taking breaks, thanks to Lana Kong for her consistent effort in getting us overworked software engineers to the tea room, as well as her general support and encouragement.

My wider thanks is given to the rest of the Ahuora team that I have spent time with this year: Elsa Klinac, Ahuroa Leech, Ben Lincoln, Keegan Hall, Mostafa Pasandideh, Matthew Taylor, Safiya Al Habsi, Parami Dilruk, Isaac Severinsen, Daniel Chong, Jack O'Leary, Catherine Muldoon, and Aleeza Adeel.

Finally, I'd like to thank my parents. Although they mostly have no idea what I have been doing all year or what this project is about, they have provided critical support for me throughout my university education, and my broader life, and they have helped in getting me to where I am now.

Declaration of Authorship

I, Caleb Archer, declare that the content of this work is of my own personal authorship, unless otherwise stated.

Table of Contents

List of Figures

List of Tables

Chapter 1. Introduction

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.

1.1 Project aim

A process simulation platform, called the Ahuora Digital Twin Platform, or the Ahuora platform, is being actively developed to resolve the aforementioned challenges. As a platform that makes use of a mathematical optimisation engine to conduct chemical process modelling, simulation and optimisation, there is a need for access to computational resources that exceed the limits of standalone devices. With a goal of being able to perform process analysis in parallel, and provide this functionality to multiple users, the ability to dynamically respond to demand to handle variable system load is critical.

This project sets out to construct and configure such a distributed environment using Kubernetes, a containerisation platform, and do so on top of a physical Raspberry Pi computing cluster. To assess how this transition allows the platform to conduct more analysis, a load testing tool is used to simulate varying request load levels. The results of these tests are analysed, providing insight into the behaviour of a distributed system within this specific platform context.

Ultimately, this project aims to answer two core questions:

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

1.2 Summary of literature review

Early software architectures made use of modular designs in order to improve the ability of engineering teams to collectively contribute to the development of software systems [1]. This view of modularity shifted from single-unit software systems to the distributed computational world [2].

Parallel computing eventually became available on retail computing systems with the arrival of the multi-core processor [3]. In general, the parallel computing space has broadened to include several styles of parallelism [4].

Distributed computing makes heavy use of virtualised environments [5], including virtual machines, which were created to tackle the limitations of creating software for specific architectures and operating systems [6]. Their use as part of software service networks developed due to their ability to simplify the deployment process. Containers enable this deployment ease without the overhead of virtualised operating systems [7].

Process simulation tools are available in two primary forms: interactive user interfaces [8], and mathematical process modelling [9]. User-interface based simulation tools lack the ability to solve some large-scale modelling problems [10], while mathematical process modelling techniques require expertise that is not held by industrial process engineers [11]

Self-adaptive systems are systems capable of dynamically adjusting themselves based on changes in the external environment [12]. Self-adaptation has been applied to microservice architectures [13]. Kubernetes is a modern container orchestration system that is used to deploy microservices [14].

In testing these systems, empirical performance testing techniques are used [15], which attempt to refer to KPIs (Key Performance Indicators) when assessing whether the target system is meeting performance requirements. Load testing is one such empirical testing technique [16], which simulates user activity at various scales to examine system performance.

Chapter 2. Background

2.1 The Ahuora Digital Twin Platform

Note: This section was written collaboratively by Caleb Archer, Shean Danes Aton, Bert Downs, and Ethan MacLeod.

The work presented in this report is part of a larger, multi-disciplinary project. This section provides an overview of the Ahuora Digital Twin Platform and related work, to provide context for the rest of the report.

2.1.1 Background

'Project Ahuora' is a Ministry of Business, Innovation and Employment (MBIE) funded project that aims to decarbonise the process heat sector. By decarbonising, New Zealands' greenhouse gas emissions will be reduced. Cost savings from reduced energy consumption are anticipated, along with increased energy independence. This is a multi-disciplinary project that involves researchers from the University of Waikato, University of Auckland, Massey University, and other global universities. Chemical Engineers bring understanding of the chemical processes that are used in industry. Electrical Engineers bring understanding of the grid system and how to integrate renewable energy sources. Mechanical engineers bring understanding of how to design and build more efficient systems. Software Engineers bring understanding of how to model, simulate, and monitor complex systems.

A key objective is to develop a digital twin platform for the chemical processing industry. This platform will allow New Zealand factory operators to model their processes, simulate different scenarios, and monitor process state in real-time. This will enable factory operators to make data-driven and scientifically backed decisions on how to improve their processes. A digital twin can recognise where the factory is underperforming, suggest real-time improvements, and help plain future investments.

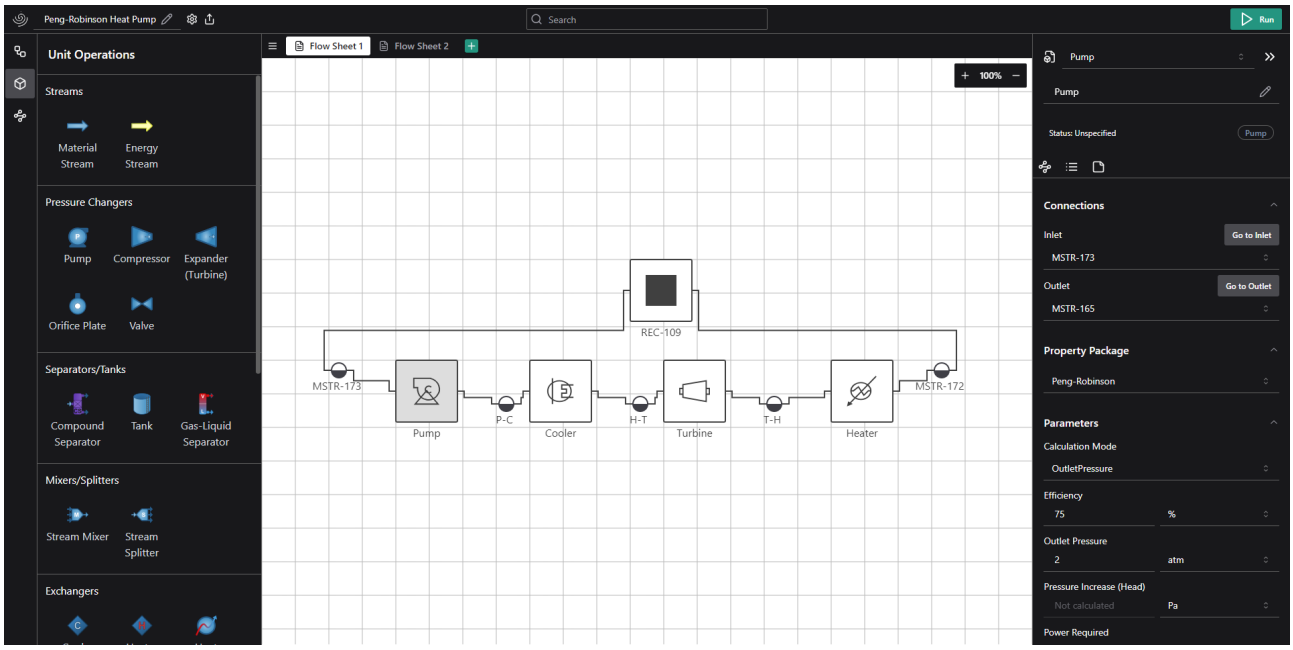


Fig. 2.1: Example Flowsheet in the Ahuora Simulation Platform

2.1.2 The Ahuora Simulation Platform

A key deliverable of the Ahuora project to date is the Ahuora Simulation Platform. This is a web-based platform that allows users to model a factory or other energy system, and simulate its performance. Much of the analysis functionality is achieved by leveragin the IDAES Process Systems Engineering framework within the backend. IDAES is a Python library that provides tools for modelling and simulating chemical processes.

Currently, the platform can model a factory at a single point in time. The user specifies the properties of the factory, such as the flow rates of different materials, the temperature and pressure of different streams, and the efficiency of different unit operations. The platform then simulates the factory and provides the user with a report on the factory's performance.

2.1.2.1 Flowsheet Interface

Fig. 2.1 shows a screenshot of the Ahuora Simulation Platform, as at August 2024. The user interface is divided into three main sections. The left-hand panel shows a list of unit operations from a factory, including pumps, heaters, heat exchangers, reactors, and material streams. The user can drag and drop these unit operations onto the canvas in the centre of the screen. The user can then connect the unit

operations together to create a process flow diagram. The right-hand panel shows the properties of the selected unit operation, such as the flow rate of the material stream, the temperature and pressure of the stream, and the efficiency of the unit operation. The user can edit these properties to simulate different scenarios.

The displayed flowsheet shows a simple heat pump cycle. The block on the top is a “recycle", specifying that the output of the cooler is fed back into the pump. A more complex flowsheet would replace the cooler and heater with heat exchangers, which exchange heat with their environment, but this provides a simple example.

2.1.2.2 Online Integration

The Ahuora Simulation Platform is designed as a web-based multi-user platform. This offloads processing and data storage to the server, allowing users to access the platform from any device with a web browser. Simulation can be very computationally expensive, particuarly in advanced models, so this is a key feature. It enables simulation to be run in parallel on powerful servers, as the simulation is not done within the local web browser, but rather via distributed server infrastructure. This allows the platform to be used in industry without requiring significant upfront investment in hardware.

In future, this will also enable the platform to be used for real-time collaboration between multiple users. Its API allows it to be integrated with other software, enabling enhanced functionality, real-time updates, and broader interactions.

2.1.3 Platform Architecture

The platform is hosted in a Kubernetes cluster, located on-site. The Kubernetes cluster handles all web traffic, deployments from the private Github Repository, and scaling of services.

Within the platform there are various containers running through the Docker platform, that can be replicated as required to scale based on service demand. The Database stores all flowsheets and model data. The solving interface uses the IDAES Process Simulation software [17], built on the Pyomo equation-oriented modelling language [18], to simulate a factory in real time. The User Interface is written in React Typescript, and uses the ShadCN UI Library and Redux Toolkit for state management and communication with the Django Backend. The Django API links all the services together, and

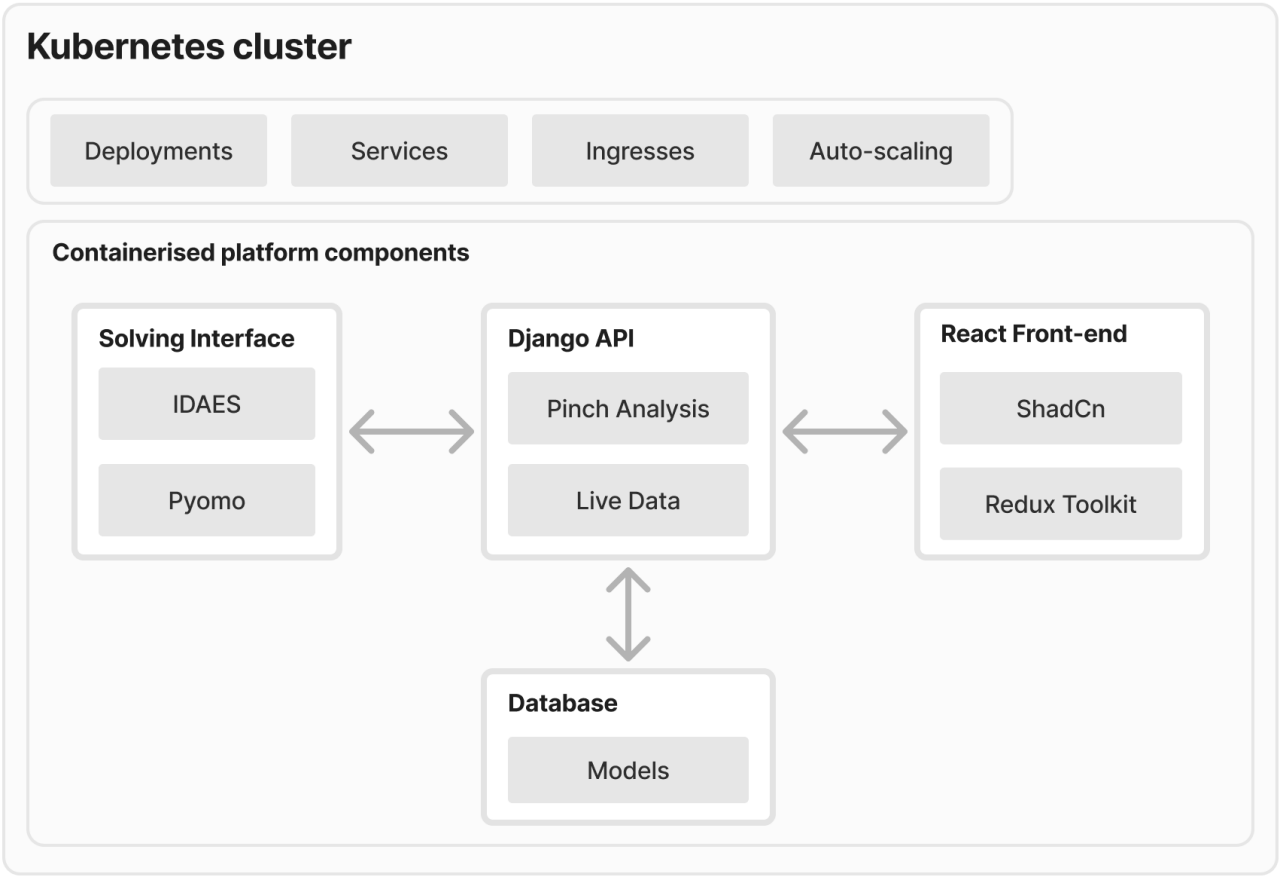


Fig. 2.2: Architecture of the Ahuora Digital Twin Platform

handles the business logic for functionality such as Pinch Analysis and Live Data processing.

2.1.4 Ongoing Projects

Multiple other projects are being completed in parallel by other team members. Each project has a distinct topic, and is largely independent, but collaboration is required to ensure that each other's work is compatible.The projects cover a variety of topics, from analyis and data processing, to usability and deployment.

2.1.4.1 Live Data Processing

Team Member Responsible: Bert Downs

The Ahuora Platform currently supports building and simulating a factory, but it has no functionality to link live data from a factory into the Ahuora Platform. By integrating real-time sensor data into

the simulation, the platform can monitor the factories' performance, and suggest tunings that will optimise resource efficiency. The data can also be used to predict and avoid failures and downtime, a key problem where many resources are wasted. Additionally, models created in the Ahuora Simulation Platform during the design phase could also be used during operation, minimising overhead costs.

2.1.4.2 Pinch Analysis

Team member responsible: Ethan MacLeod

One of the key optimisation tools the Ahuora Platform provides is the Pinch Analysis module. Pinch analysis is the process of identifying heat recovery pockets in a given system, and indicating where heat can be exchanged. The end goal of pinch analysis is reducing the overall heat consumption of a process, which in turn results in lower operational costs for a plant, and less greenhouse gas emissions produced. From the processes that are modelled on the Ahuora Digital Twin Platform, there is a distinct need for integration with process optimisation tools like Pinch Analysis to inform the decision-making process for operators and engineers.

2.1.4.3 User Interface Usability

Team member responsible: Shean Danes Aton

This project aims to deliver an intuitive user interface and quality user experience in the Ahuora Digital Twin Platform through improving its UI through employing user centred methodologies including A/B testing, interviews, card sorting, the thinkAloud method, and surveys. Additionally, modern diagramming tools were reviewed and end users' knowledge were leveraged to the development of the ADTP UI. This project utilises software technologies and existing design components,ShadCn, for design implementations. Enhancing the ADTP's UI increases user satisfaction, increases productivity, and minimises operational error.

2.1.4.4 Distributed Platform Deployment

Team member responsible: Caleb Archer

Each of the software components making up the platform need to be deployed together within a distributed environment that allows for efficient allocation and use of computational resources. This

is achieved within the context of a Kubernetes cluster, on which several replicas of each platform component may run at any given time based on system load. The ability to scale workloads in response to demand increases the capacity of the platform to perform process modelling and simulation, and handle a greater number of users.

2.1.5 Other Long-term Platform Objectives

The projects listed pave the way for future work on the Ahuora Digital Twin Platform. This includes expanding the number and type of unit operations supported, and supporting a wider range of chemical processes. It is intended that the Ahuora Platform also support Hybrid modelling, where Machine Learning Models are used to represent complex unit operations that do not have an exact mathematical representation. Analysis functionality to be added includes Dynamic Simulation, to model the factory over time and predict the effect of changes in the system, Process Variable Optimisation, to calculate optimal operating conditions, common diagramming and reporting functionality, and scheduling tools.

Because of the long-term context of the Ahuora project, work in this dissertation is constrained by the future requirements of the system. Care has been taken to ensure that the methods outlined can be extended to support the future objectives of the Ahuora Platform.

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 Concepts

2.2.1.1 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.

2.2.1.2 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.

2.2.1.3 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.

2.2.1.4 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.

2.2.1.5 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 system 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 Performance test experiment set up

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 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
```

3.2.2 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, 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.

3.2.3 Empirical comparison

3.2.3.1 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 as making a request to a website at some URL, and waiting for the response. During a test, k6 displays

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

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

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

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

3.2.3.2 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.

3.3 Load test experiments

3.3.1 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.

3.3.2 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.

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

3.3.3 Test scenarios

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

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

Table 3.1: API endpoints used for system testing

The unit operation retrieval endpoint interacts with the Django API server and the PostgreSQL database. The flowsheet solving endpoint interacts with both of these, and 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.

3.3.3.1 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 (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.

3.3.3.3 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 3.2: Base options set for k6 load tests

3.3.3.4 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

3.3.3.5 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.

3.3.3.6 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 3.3). The goal is to determine what

performance degradation (if any) happens when the system experiences load that is still within an expected range.

3.3.3.7 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.

3.3.3.8 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 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.

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.

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.

3.3.4.1 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 3.4: 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

3.3.4.2 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

This chapter explores the steps involved in setting up the distributed Kubernetes environment that the Ahuora platform is intended to run on, from the hardware configuration, to the establishment of pipelines that automatically deploy both software and cluster changes within a repository to the cluster.

4.1 Raspberry Pi hardware and network provisioning

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

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

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

managed university 4G network, and thereby obtain internet access. With only eight ethernet ports on the switch, cluster nodes (Pi devices) requiring internet access were connected to the router over Wi-Fi. Two of the eight nodes have been configured to access the internet this way.

After this, the nodes still needed IP addresses assigned for the network switch interface. A DHCP (Dynamic Host Configuration Protocol) server called DNSMasq was set up on one of the Wi-Fi connected nodes (called the *ingress* node). DHCP allows for devices on a network to be automatically assigned IP addresses from an available address pool, avoiding the tedious process of manually configuring network interfaces on each individual device. In this context, it was necessary to assign static (unchanging) IP addresses to each device, so DNSMasq was configured to assign specific addresses based on the MAC address of each device. The ingress node was assigned 192.168.100.101, with the rest numbered from 192.168.100.121 to 192.168.100.127. DNSMasq was also used as a DNS (Domain Name Server) server to enable name-based communication over the network, instead of being forced to recall specific IP addresses.

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

4.2 Ansible playbook automation

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

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

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

[1] K3s Ansible GitHub repository: https://github.com/k3s-io/k3s-ansible

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.

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

[1] ZeroTier VPN: https://www.zerotier.com/

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.

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.

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

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[1] 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 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.

[1] Cloudflare Tunnel: https://www.cloudflare.com/products/tunnel/

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 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-built 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 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 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.

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

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 to be 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.

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

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 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 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. 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.

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

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 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[2] 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

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

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

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 final step in enabling full pipeline 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.

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.

With the transition of the Ahuora platform to the Kubernetes complete, the platform can now be accessed and used over the internet for chemical process modelling, analysis and simulation. The following chapter will explore the insights that can be obtained from analysis of the testing of this platform implementation.

Chapter 5. Results and Discussion

5.1 Performance analysis

To determine the effects of different cluster configurations on the achievable system performance, analysis needs to be conducted

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 [12], 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.

5.1.2 Secondary metrics

5.1.2.1 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 prevent invalid observations being made about test results. 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.

5.1.2.2 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.

5.1.2.3 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 and request rates will be assessed using an exponentially-weighted moving average (EWMA), with a window of ten seconds. 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

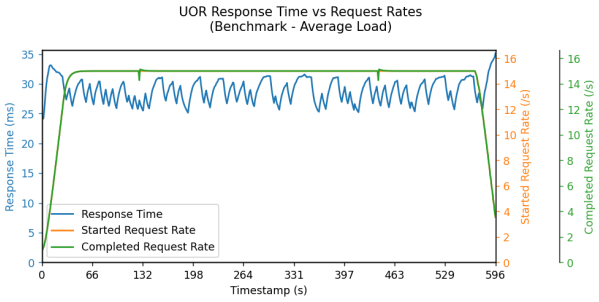


Fig. 5.1: UOR response time vs. request rate graph - average load benchmark

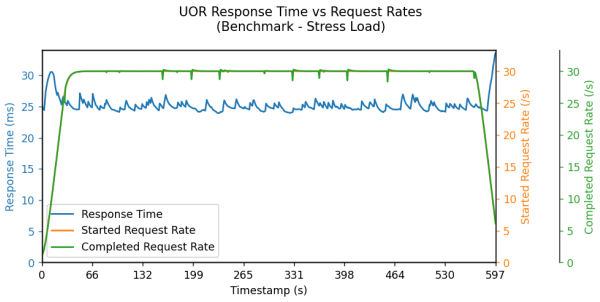


Fig. 5.2: UOR response time vs. request rate graph - stress load benchmark

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 nor stress load tests had any failed requests.

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

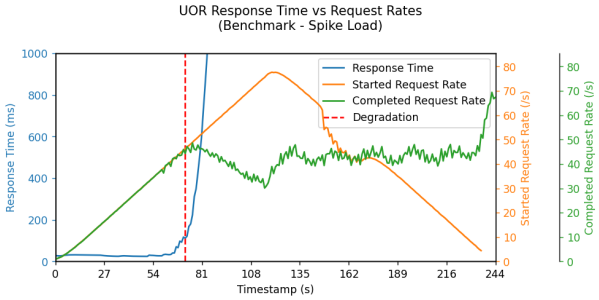


Fig. 5.3: UOR response time vs. request rate graph - spike load benchmark

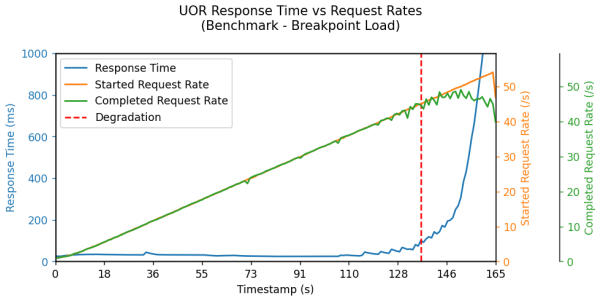


Fig. 5.4: UOR response time vs. request rate graph - breakpoint load benchmark

requests failed in the breakpoint tests, but this in part due to the early termination threshold used by breakpoint tests.

5.2.2 Replica count

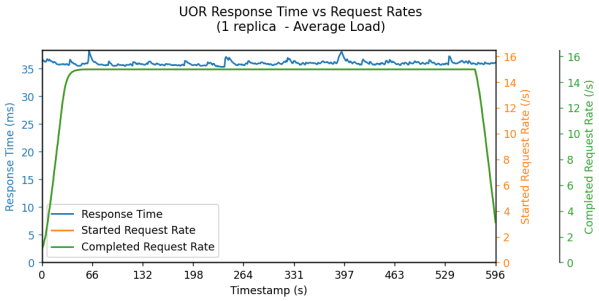


Fig. 5.5: UOR response time vs. request rate graph - average load with one replica

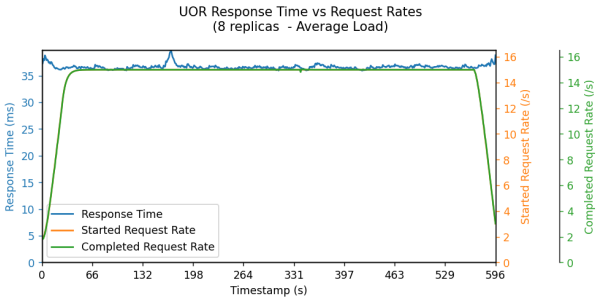


Fig. 5.6: UOR response time vs. request rate graph - average load with eight replicas

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. 5.7) when average load is applied. The simple explanation for this difference is the use of TLS in the HTTPS connections used between the test client and the cluster, which may take between 5 and 10

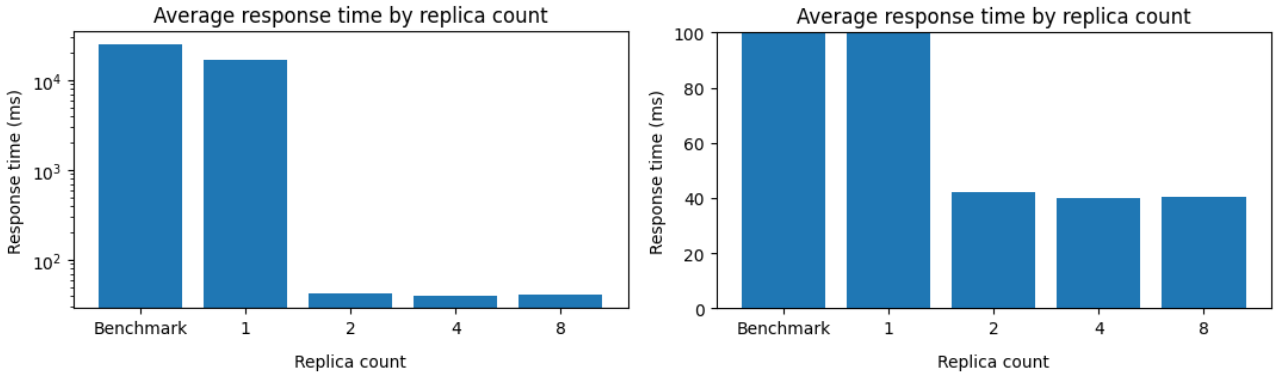
Table 1: Table of UOR median response times by replica count, average load

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

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.)

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 5.6), 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.

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

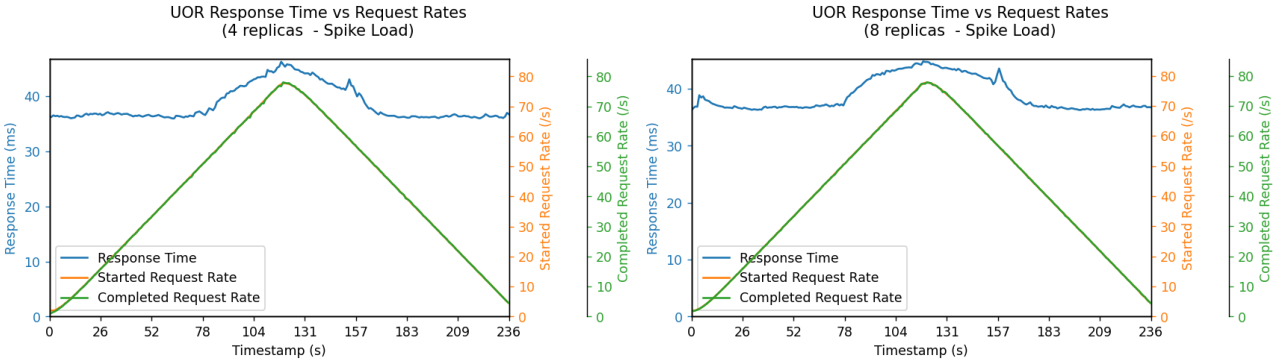


(a) Logarithmic scale

(b) Linear scale

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

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.

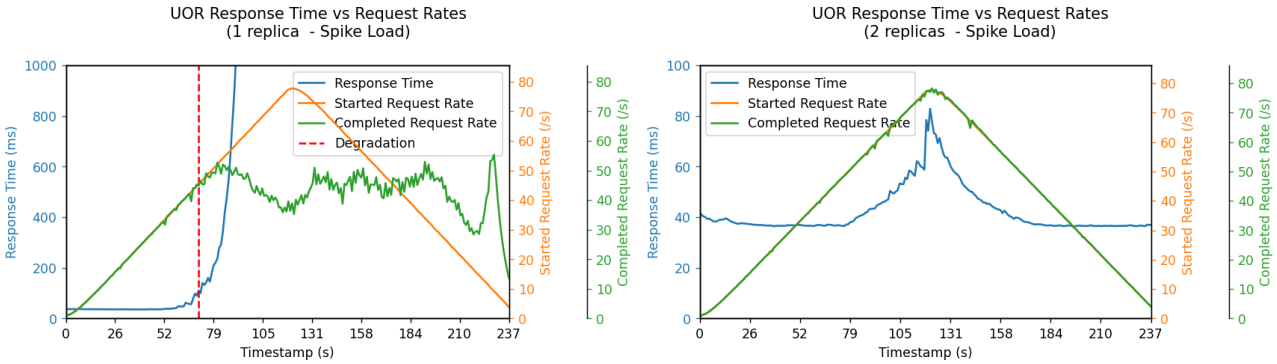


(a) 4 replicas

(b) 8 replicas

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

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

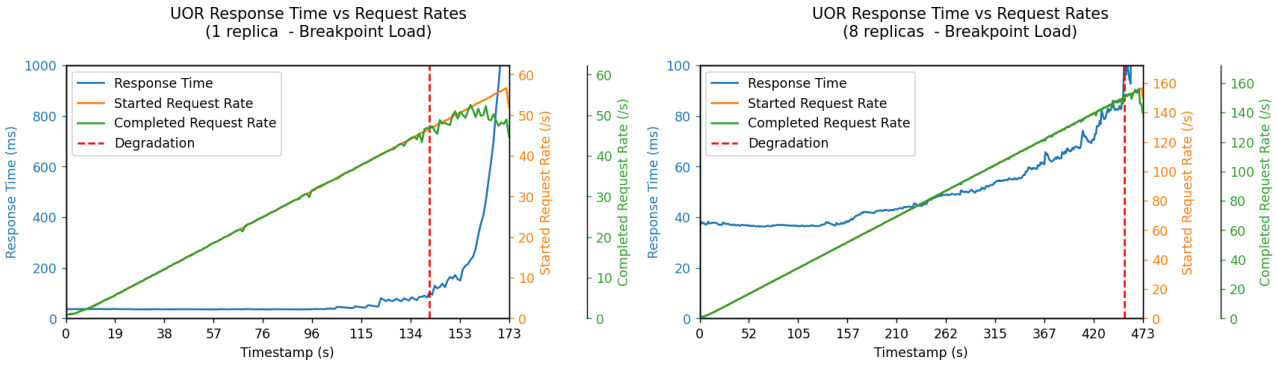


(a) 1 replica

(b) 2 replicas

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

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)

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.

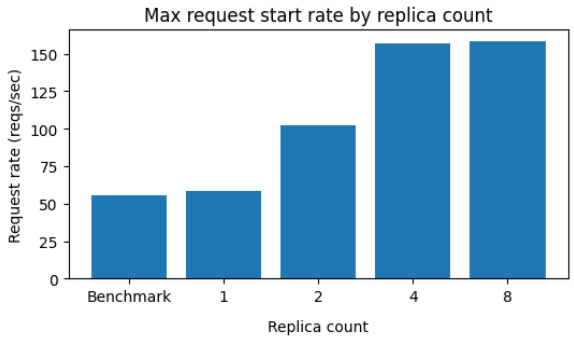


Fig. 5.12: Chart of maximum request start rates by replica count

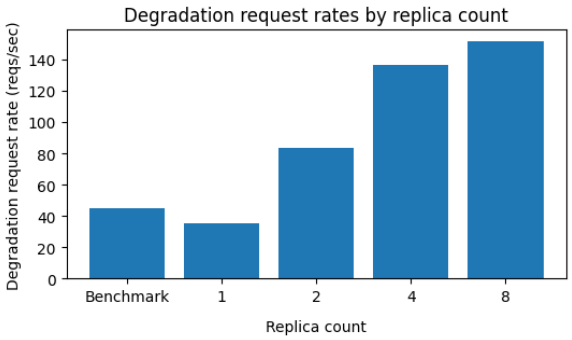
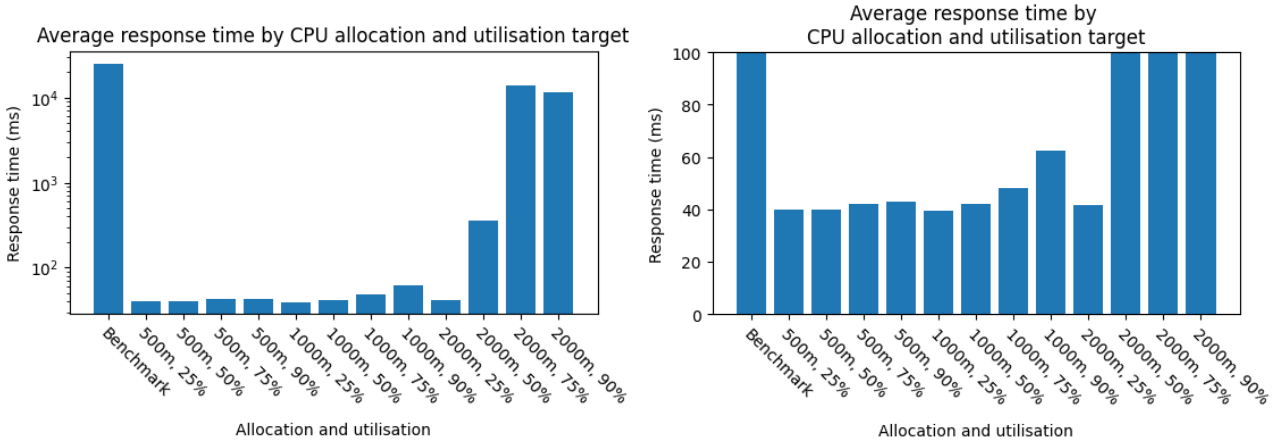


Fig. 5.13: Chart of degradation request rates by replica count

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

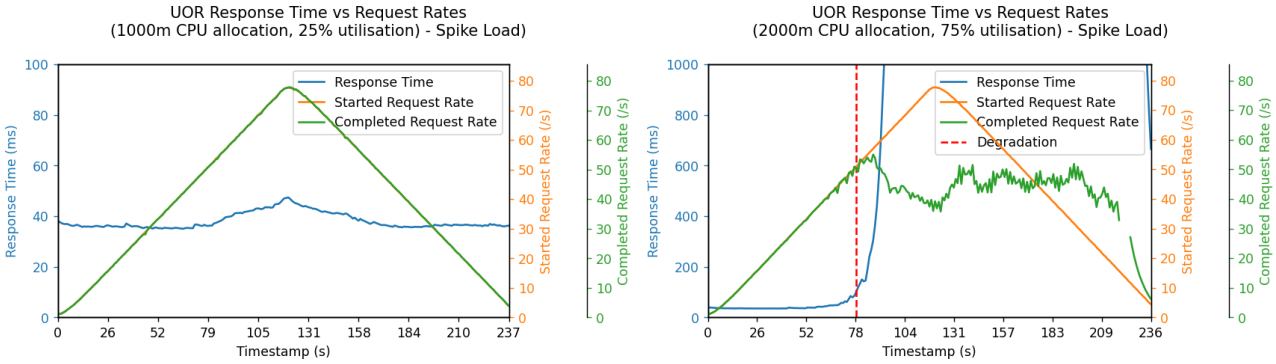
Fig. 5.14: UOR median response times by CPU allocation and utilisation target (spike load)

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 exacerbated when the CPU allocation is simultaneously 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.

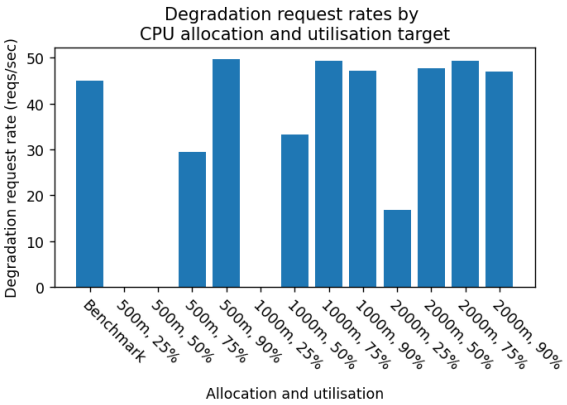


Fig. 5.16: Degradation request rates by CPU allocation and utilisation target (spike load)

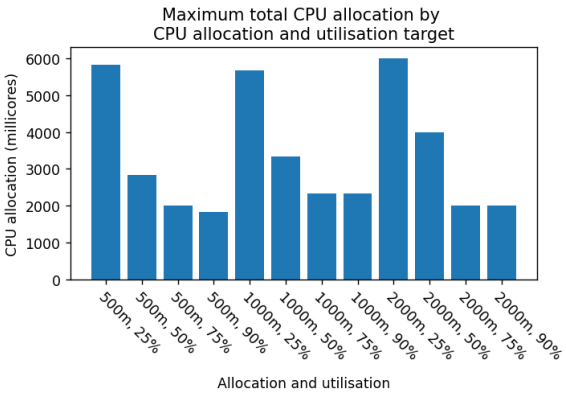
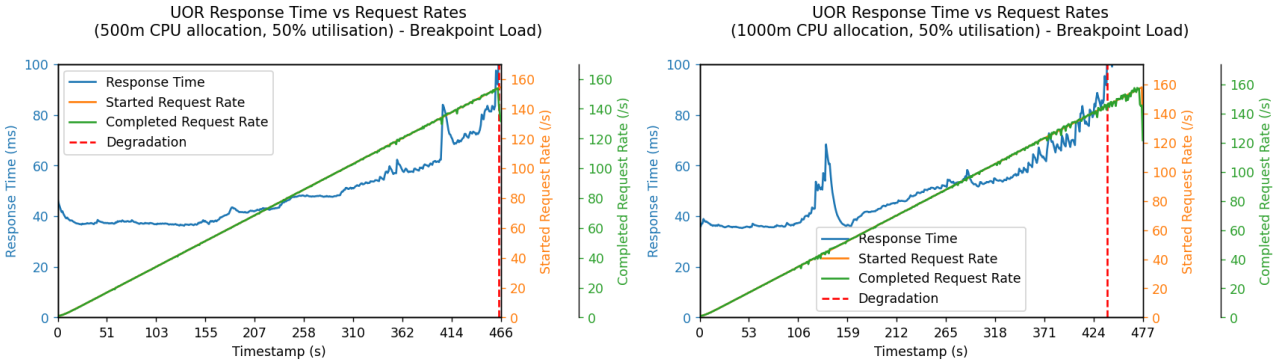


Fig. 5.17: CPU allocation by CPU allocation and utilisation target (spike load)

Within the breakpoint tests, the best performing configuration in terms of the degradation request rate is 500 millicores with a 50% target (Fig. 5.19). The response time curve, as shown in Fig. 5.18 (a), shows distinct spikes where the effect of scaling actions can be observed. At approximately 60

started RPS, there is a momentary rise in the response time, which then quickly decreases again. The response time generally increases with the request rate, as seen previously in the replica count tests (Fig. 5.11).



(a) 500m CPU allocation, 50% target utilisation

(b) 1000m CPU allocation, 50% target utilisation

Fig. 5.18: UOR response time vs. request rate graphs (breakpoint load)

For 1000 millicores and a 50% utilisation target, a more pronounced scaling action effect is observed in 5.19 (b). At 45 started RPS, the response time increases by ~75%, and sharply drops. This configuration show signs of greater volatility than the 500 millicores and 50% target experiment, especially in the latter half of the test. However, another response time spike is seen in 5.18 (a), at ~140 started RPS. Despite not crossing the degradation threshold as early as the 1000 millicores experiment, its response time curve does not showcase a perfect load reaction ability of the system. In fact, the 1000 millicores experiment had the highest maximum request start rate (162.33) of all resource allocation experiments, giving evidence that there are various trade offs involved across the configurations.

As before, the 2000 millicore experiments largely performed poorly (Fig. 5.19). However, in the breakpoint tests, it is observed that the 1000 millicore experiments with 75% and 90% targets exceed the degradation threshold early in their respective tests, but do not reach their breakpoints until much later (Fig. 5.20). The reason for this pattern is explained by Fig. 5.21, describing the behaviour of the 1000 millicores, 75% target experiment. Before reaching the breakpoint, the response time increases by more than 100% (at ~50 started RPS) and drops by the same amount, repeating a similar pattern at ~90 started RPS, before rising once more, never returning below 100 milliseconds after this.

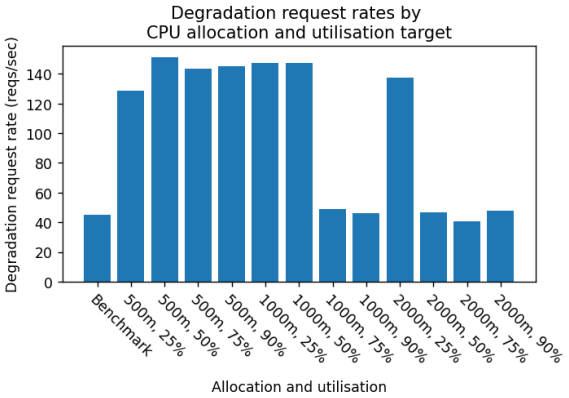


Fig. 5.19: UOR degradation request rates by CPU allocation and utilisation target (breakpoint load)

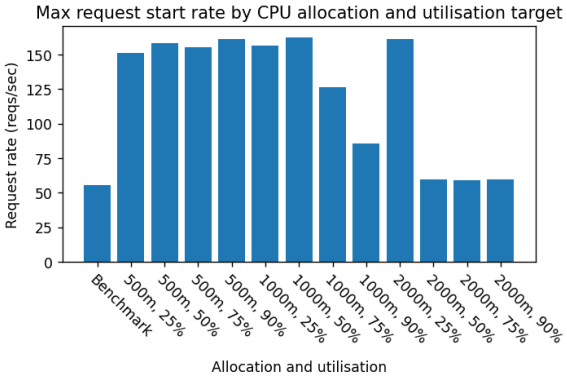


Fig. 5.20: UOR max request start rates by CPU allocation and utilisation target (breakpoint load)

These repeated spikes show how an inappropriate scaling policy can cause system performance to be very volatile. In this case, the autoscaler takes scaling action too late, allowing for load to increase excessively before scheduling more pods.

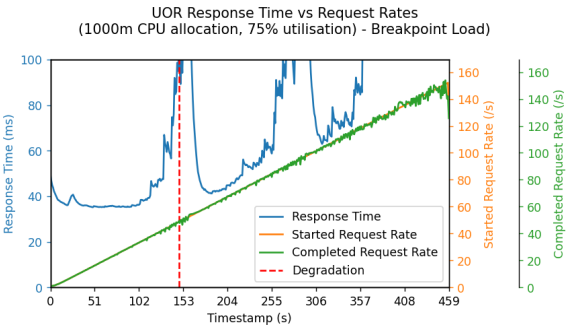


Fig. 5.21: UOR response time vs. request rate graphs (breakpoint load)

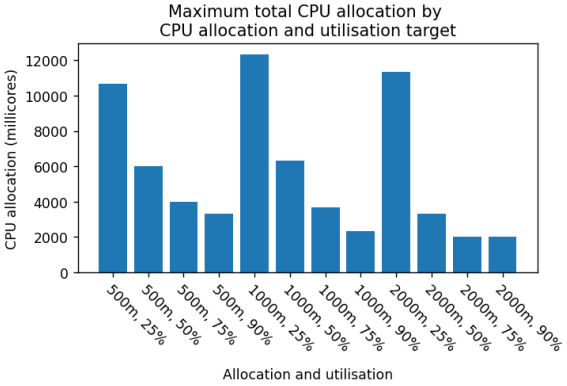


Fig. 5.22: UOR CPU allocation by CPU allocation and utilisation target (spike load)

The breakpoint tests also saw the maximum CPU allocations increase compared with the spike tests (Fig. 5.22), again showing the low target experiments receiving large allocations. The highest allocation comes from the 1000 millicores, 25% target experiment, at an average maximum of 12,333 millicores. The Kubernetes cluster tested on has 28,000 millicores available, so this allocation is ~44% of the total CPU available.

5.3 Flowsheet solving (FS) experiment results

The following results showcase the performance behaviour of a local deployment of the Ahuora platform versus various scaling configurations for a distributed, Kubernetes-based deployment. In this case, the results are now impacted by two primary software components: the Django API, and the IDAES service.

5.3.1 Benchmarks

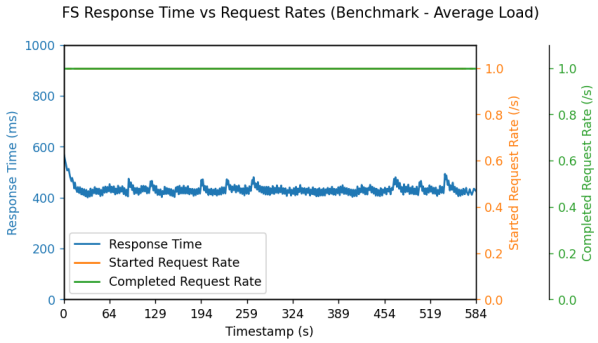


Fig. 5.23: FS response time vs. request rate graph - average load benchmark

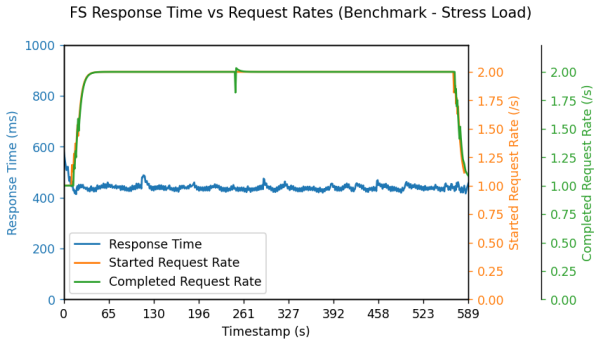


Fig. 5.24: FS response time vs. request rate graph - stress load benchmark

Similarly to the UOR benchmarks, the local deployment has few issues sustaining both the average and stress load profiles (Fig. 5.23 and 5.24). However, it can be noted that the median response time of ~376 milliseconds under average load is significantly higher than the UOR average load benchmark (~30 milliseconds). The FS API endpoint used does more work than the UOR endpoint (retrieving all data for a flowsheet and performing process simulation), and involves communication between the Django API and the IDAES service. All of this results in the FS endpoint taking longer to fulfil a request on average.

As with the UOR spike test benchmark, the local deployment struggles to complete as many requests as received (Fig. 5.25), exceeding the 1000 millisecond threshold after a start request rate of 4.74. The median response time in the spike tests is 4718 milliseconds, more than ten times the average load median response time. Approximately 24% of spike test requests failed. The breakpoint tests (Fig. 5.26) see the degradation request rate being ~4.63 RPS, and the start rate reaching no more

than 8 RPS.

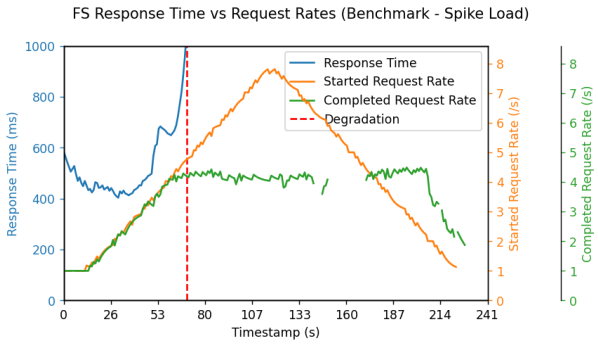


Fig. 5.25: FS response time vs. request rate graph - spike load benchmark

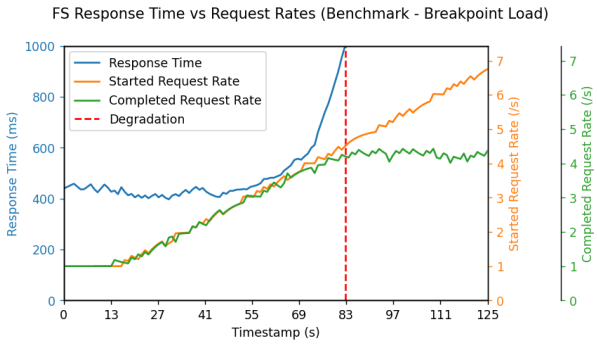


Fig. 5.26: FS response time vs. request rate graph - breakpoint load benchmark

5.3.2 Replica count

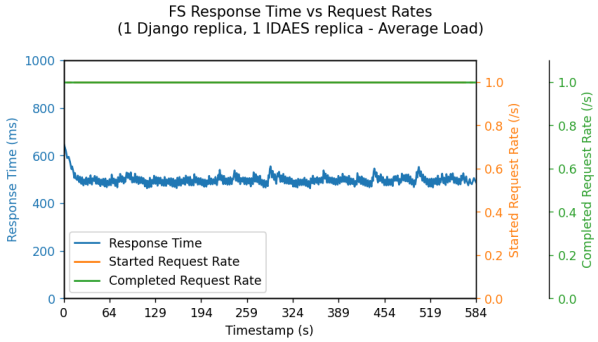


Fig. 5.27: FS response time vs. request rate graph - average load (1 Django replica, 1 IDAES replica)

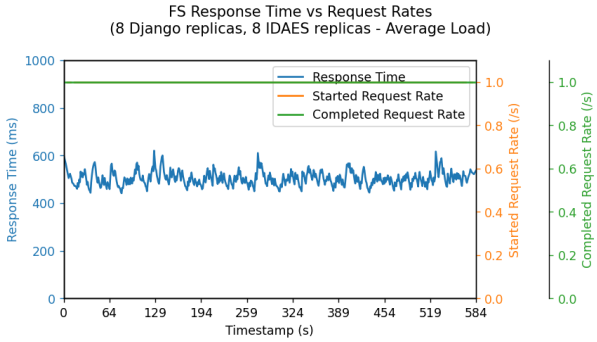


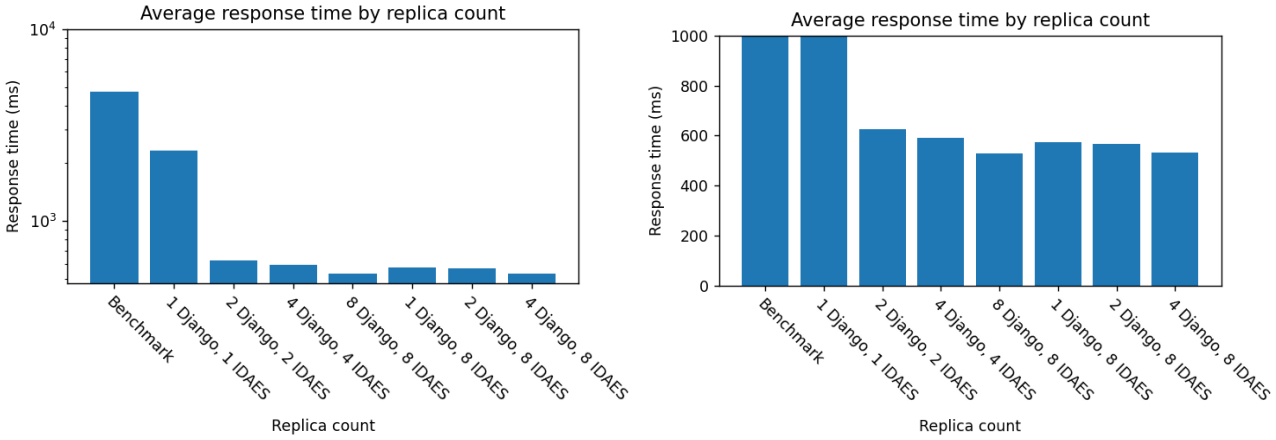
Fig. 5.28: FS response time vs. request rate graph - average load (8 Django replicas, 8 IDAES replicas)

An average load applied to the smallest and largest FS replica count configuration behaves similarly to the benchmark (Fig. 5.27 and 5.28), with no difference in the ability of the cluster deployment to serve requests compared to the benchmark. As seen in the UOR replica count tests previously, the median response time is between 17% and 24% higher (Fig. 5.29) with the cluster deployment compared with the benchmark. The potential reasons for this are outlined in 5.2.2.

Table 2: Table of FS median response times by replica count, average load

Django replica count	IDAES replica count	Median response time (ms)	Benchmark multiplier
1	1	441.67	1.17
2	2	456.85	1.21
4	4	466.84	1.24
8	8	452.06	1.2
1	8	461.4	1.23
2	8	463.51	1.23
4	8	452.28	1.2

A similar response time pattern to the UOR replica count spike tests is seen in Fig. 5.30. A higher number of replicas decreases the median response time, with both the benchmark having an order of magnitude higher response time, and the single replica configuration (both Django and IDAES) between three to four times higher (with 25.44% of requests having failed). With eight Django and IDAES replicas, the response time is lowest, though the difference is not significant for the spike load profile.

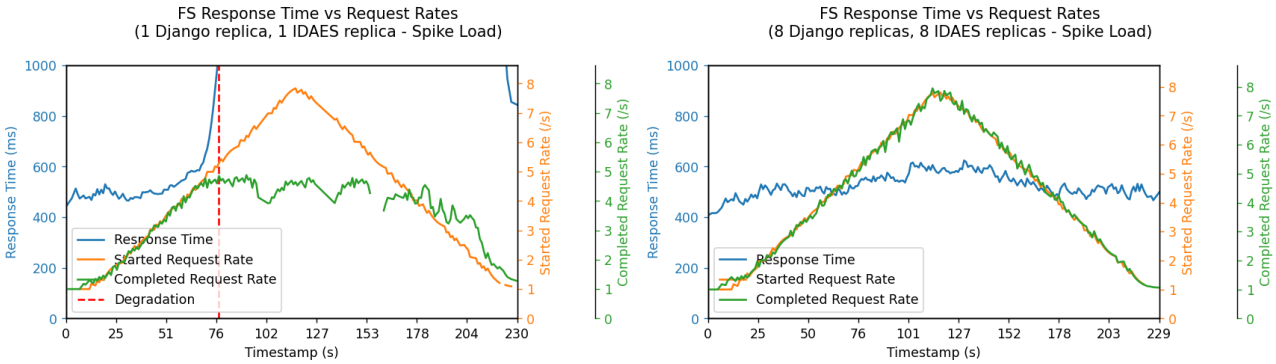


(a) Logarithmic scale

(b) Linear scale

Fig. 5.30: FS median response times by replica count (spike load)

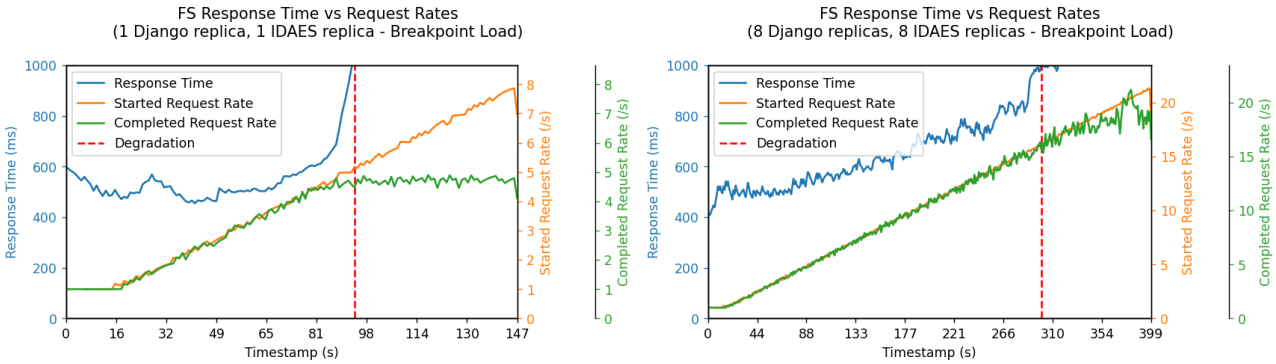
For the single replica configuration, the degradation request rate is 5.17 RPS, which is 9% higher than that encountered by the benchmark. With the eight replica configuration (Fig. 5.31 (b)), the system is able to keep the request completion rate consistent with the started request rate, albeit with



(a) 1 Django and IDAES replica

(b) 8 Django and IDAES replicas

Fig. 5.31: FS response time vs. request rate graph (spike load)



(a) 1 Django and IDAES replica

(b) 8 Django and IDAES replicas

Fig. 5.32: FS response time vs. request rate graph (breakpoint load)

some volatility. Likewise, a similar degradation request rate of 5.19 RPS is seen in the breakpoint test (Fig. 5.32) for the single replica configuration. On the other hand, the degradation request rate with eight replicas is 16.75 RPS. As with the UOR tests, simply increasing the replica count linearly does not likewise linearly increase the degradation request rate or the maximum request start rate reached. Fig. 5.33 and 5.34 both show that the difference between having four or eight Django replicas is not substantial. There is a larger gap between the four and eight IDAES replica scenarios, but even this difference is not more than 37%.

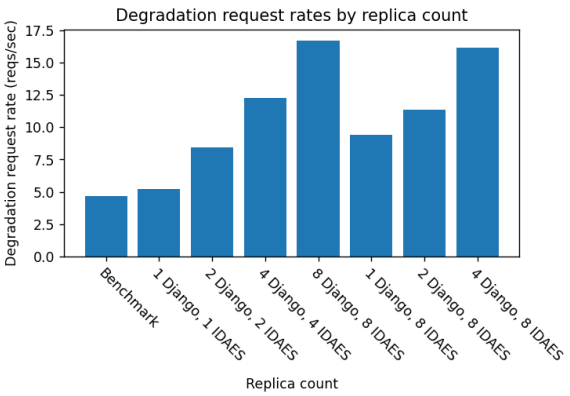


Fig. 5.33: FS degradation request rates by replica count (breakpoint load)

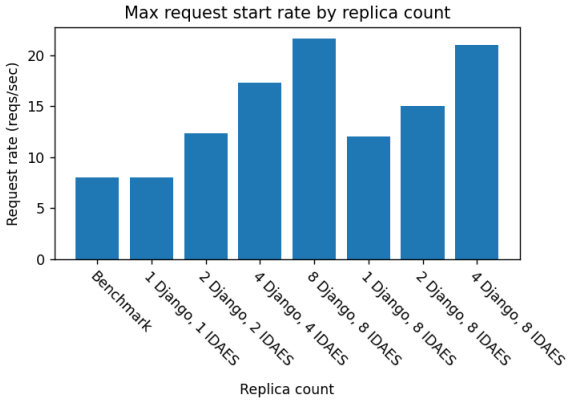
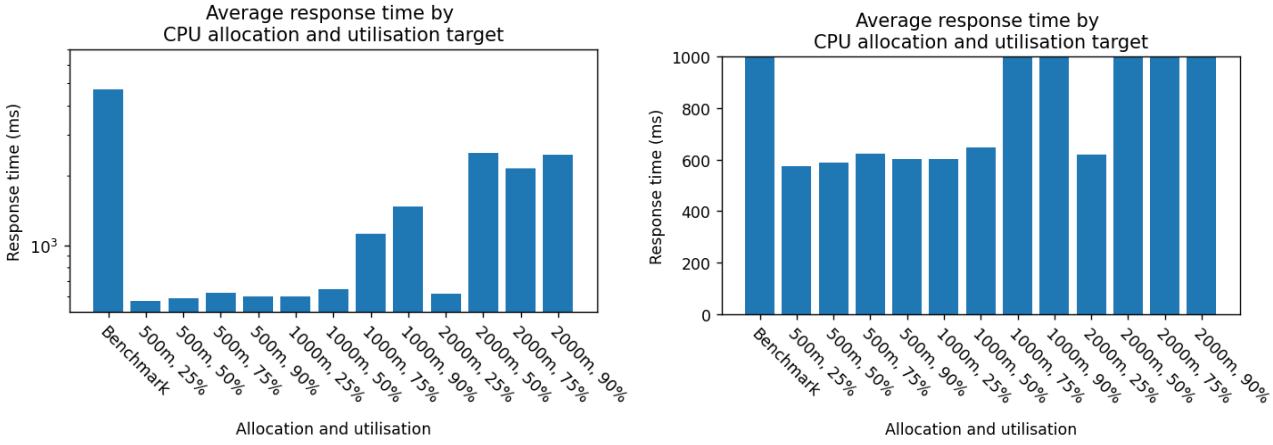


Fig. 5.34: FS max request start rates by replica count (breakpoint load)

5.3.3 Resource allocation and utilisation

These resource allocation and utilisation tests show the effect of autoscaling on two logically linked deployments, though the focus is on the scaling of the IDAES service, which has its autoscaling configuration varied between experiments.



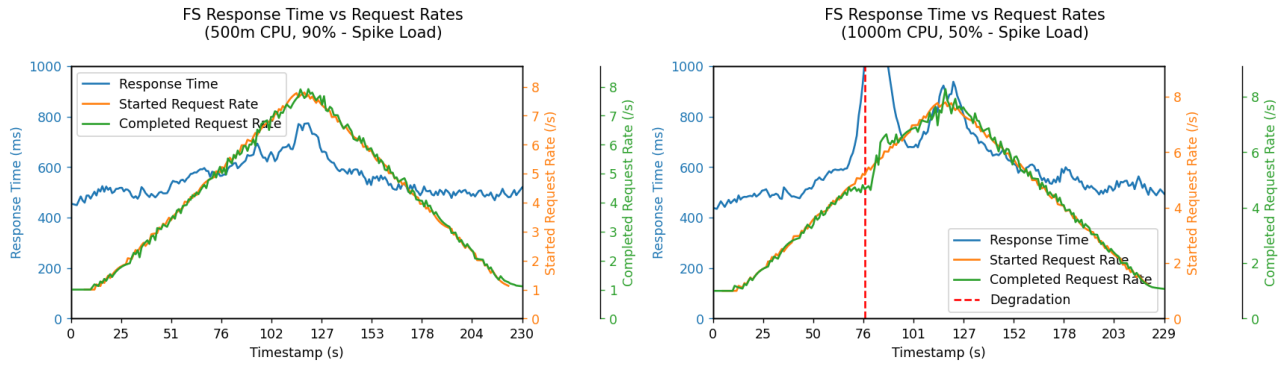
(a) Logarithmic scale

(b) Linear scale

Fig. 5.35: FS median response times by CPU allocation and utilisation target (spike load)

As Fig. 5.35 shows, all of the 500 millicores allocation configurations had relatively consistent median response times under the spike load tests, while the utilisation target seems to play a much more important role for the 1000 millicores allocations. The 75% and 90% targets experienced

significantly higher median response times when 1000 millicores was allocated. As has been seen numerous times previously, all but the 25% target of the 2000 millicores set have extremely poor response time performance, though not as high as the benchmark. The lowest response time is achieved by the 500 millicore, 25% target configuration. However, the only configuration that does not suffer from breaches of the degradation threshold is a 500 millicores and 90% target (Fig. 5.37), with the rest ranging from 4 to 8 degradation RPS levels. Fig. 5.36 (a)presents the performance of this configuration over the course of a test: the response time (somewhat erratically) increases with the request rate, but remains below 1000 milliseconds, before stabilising downwards as the request rate decreases. Fig. 5.36 (b) displays instead the slow system reactivity with a 1000 millicores allocation and 50% target. The response time spikes, and before the peak request rate, it returns to a new baseline, before increasing again along with the request rate.



(a) 500m CPU allocation, 90% target utilisation

(b) 1000m CPU allocation, 50% target utilisation

Fig. 5.36: FS response time vs. request rate graphs (spike load)

Fig. 5.38 shows a pattern that has presented itself in the UOR resource allocation and utilisation tests: low utilisation targets result in high total CPU allocation. Whether the subject workload is the Django API or the IDAES service, this pattern seems to remain true.

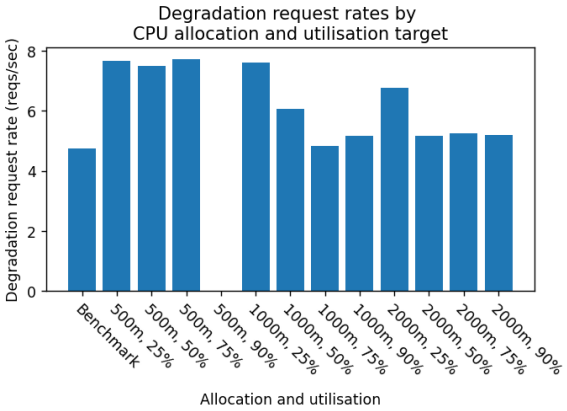


Fig. 5.37: FS degradation request rates by CPU allocation and utilisation target (spike load)

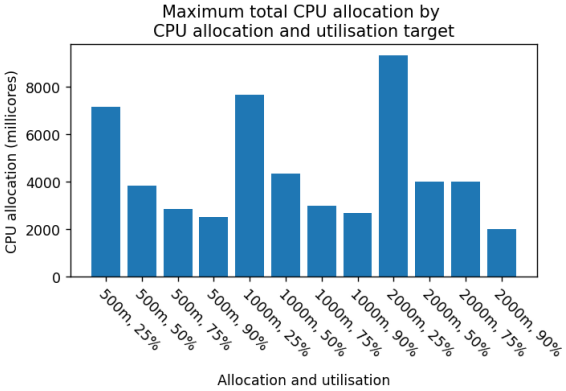
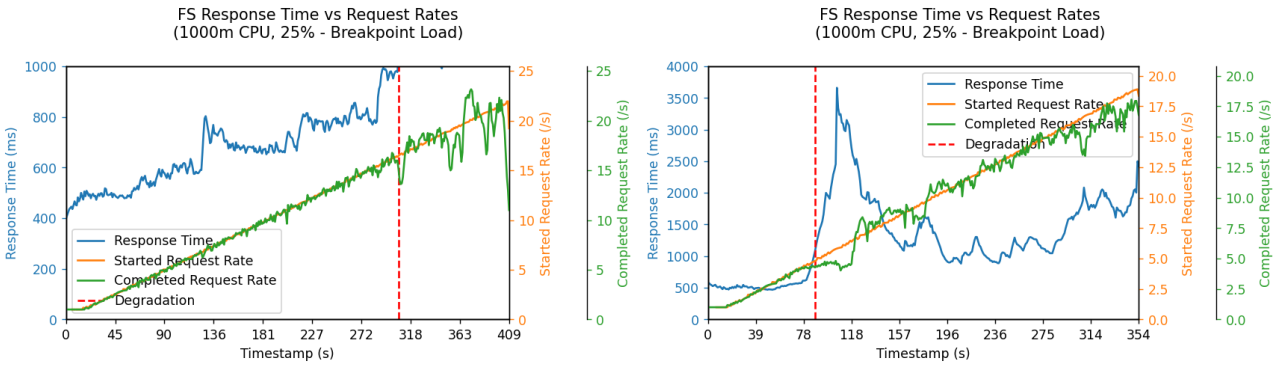


Fig. 5.38: FS CPU allocation by CPU allocation and utilisation target (spike load)



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

(b) 1000m CPU allocation, 75% target utilisation

Fig. 5.39: FS response time vs. request rate graphs (breakpoint load)

In the breakpoint tests, the best configuration with respect to the degradation request rate is that with 1000 millicores allocated and a 25% target, reaching 16.43 RPS (~3.5 times the benchmark) before exceeding the 1000 millisecond threshold (Fig. 5.40). It can be noted that, despite this, the response time is still quite volatile, and there is evidence of slow scaling actions at several points in the test (Fig. 5.39). Along with this, the total CPU allocation across the cluster by this configuration was 22,000 millicores (Fig. 5.42), or 22 cores, the second-highest allocation of all configurations. This value constitutes ~78.6% of all available CPU on the cluster, which is likely to cause significant scheduling issues for other deployments.

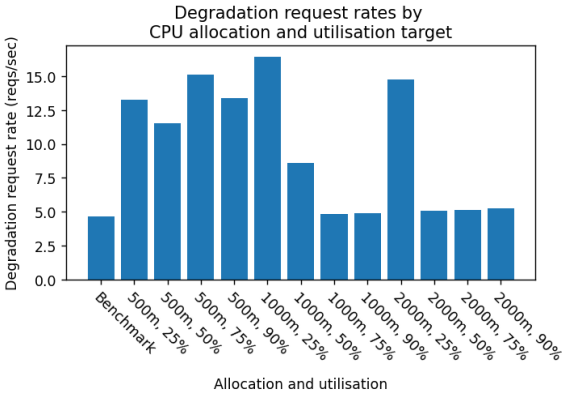


Fig. 5.40: FS degradation request rates by CPU allocation and utilisation target (breakpoint load)

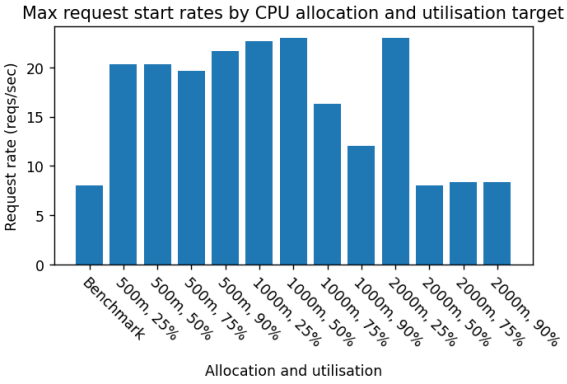


Fig. 5.41: FS max request start rates by CPU allocation and utilisation target (breakpoint load)

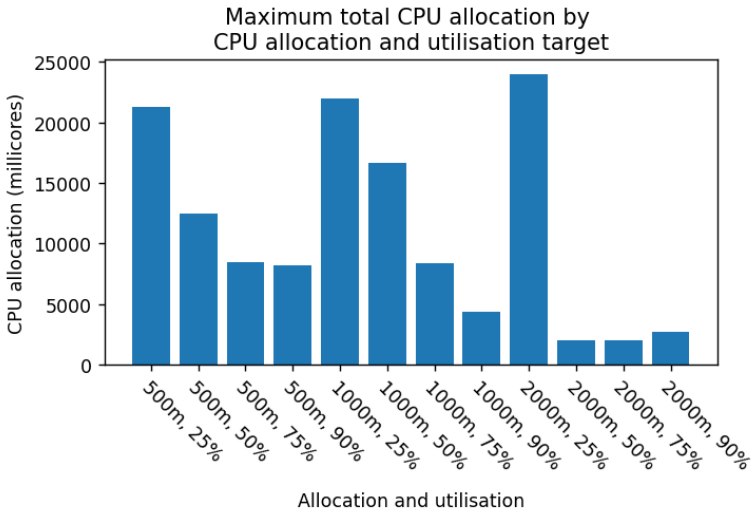


Fig. 5.42: FS CPU allocation by CPU allocation and utilisation target (breakpoint load)

Across both the UOR (unit operation retrieval) and FS (flowsheet solving) experiments, a selection of scaling configurations allow the number of requests processed in a stable fashion to increase by three to four times, allowing more analysis to be conducted, and more users to access the platform. However, as indicated by the diminishing returns when increasing the number of replicas (for either the Django API or the IDAES service), there is evidence that bottlenecks exist in the system (a component is slowing the system down), which will need further investigation.

5.4 Impacts

With the successful migration of the Ahuora Digital Twin Platform to a distributed Kubernetes cluster, the platform is now able to take advantage of additional processing power, with no need to invest in individually powerful machines to perform efficient process modelling, optimisation and analysis at scale. Additionally, by making use of a cloud-native environment, expanding the platform's capabilities becomes more straightforward. New internal software services can be easily integrated into the existing system architecture, and the expansive world of community cloud-native software can likewise serve to extend platform functionality.

Having performed empirical performance testing and analysis, the produced data can be used to inform further optimisation of the deployed software, in particular the Django API and IDAES service, which are core pieces of the Ahuora platform. By appropriately setting resource allocation requests and utilisation targets, the platform will be able to handle variable system load while avoiding unnecessary overprovisioning of cluster resources, which can be allocated to other deployed software.

5.5 Threats to validity

There are some elements of this work that could impact the legitimacy of the obtained results, or at the very least, their usefulness in practice. First, the metrics collected at one-minute intervals from the cluster during performance testing make it difficult to perform fine-grained analysis on how all parts of the internal system are behaving. Client-side performance data, from the perspective of the k6 test client, is plentiful, but this is not the case with system metrics. Such a long interval means that very short-term behaviour (like pod scaling actions) may be missing, or otherwise inaccurately captured.

Another potential cause for concern is the limited scaling configuration parameter space explored. Because of the limited time available to conduct extensive testing, some parameter combinations could not be explored in full. For example, with the flowsheet solving resource allocation and utilisation tests, the scaling settings for the IDAES service were varied, but the Django API settings were fixed, despite the API being a core part of the solving pipeline. It is highly likely that a restructuring of the solving pipeline will prevent this from being an issue in future.

Chapter 6. Conclusion and Future Work

A process simulation platform, called the Ahuora Digital Twin Platform, has been migrated to a Kubernetes-based distributed computing system. Individual software components, including the Django API, the IDAES service, the front-end and a database have been configured to run on andn communicate within the cluster. A load-testing procedure was created, using a series of load profiles to assess system performance. Performance analysis was conducted, identifying the impact of different software scaling configurations on system performance.

Future work

There are several areas of the work showcased in this document that can be improved upon. First, performance improvements may be obtained by decoupling the Django API from the IDAES service, integrating a message queue to asynchronously send solve requests and receive the results. From here, customised metrics can be used to allow event-based scaling, and thereby the IDAES service will be scaled not based on CPU usage, but instead the number of tasks waiting in a message queue. Work is needed to identify what is creating the bottleneck that prevents system throughput from increasing more than it currently is. Finally, the security of the cluster could be substantially improved: strict inter-pod and namespace policies should be put in place to minimise the risk of cross-software exploitation.

Bibliography

[1] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *Communications of the ACM*, vol. 15, no. 12, pp. 1053–1058, Dec. 1, 1972.

[2] J. Sifakis, "A framework for component-based construction," in *Third IEEE International Conference on Software Engineering and Formal Methods (SEFM'05)*, Sep. 2005, pp. 293–299.

[3] M. Bridges, N. Vachharajani, Y. Zhang, T. Jablin, and D. August, "Revisiting the sequential programming model for multi-core," in *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*, Dec. 2007, pp. 69–84.

[4] Z. Xu, Y. He, W. Lin, and L. Zha, "Four styles of parallel and net programming," *Frontiers of Computer Science in China*, vol. 3, no. 3, pp. 290–301, Sep. 2009.

[5] R. P. Goldberg, "Survey of virtual machine research," *Computer*, vol. 7, no. 6, pp. 34–45, Jun. 1974.

[6] A. Whitaker, M. Shaw, and S. D. Gribble, "Denali: Lightweight virtual machines for distributed and networked applications," 2001.

[7] I. Pietri and R. Sakellariou, "Mapping virtual machines onto physical machines in cloud computing: A survey," *ACM Computing Surveys*, vol. 49, no. 3, pp. 1–30, Sep. 30, 2017.

[8] V. A. Merchan, E. Esche, S. Fillinger, G. Tolksdorf, and G. Wozny, "Computer-aided process and plant development. a review of common software tools and methods and comparison against an integrated collaborative approach," *Chemie Ingenieur Technik*, vol. 88, no. 1, pp. 50–69, 2016.

[9] E. N. Pistikopoulos *et al.*, "Process systems engineering – *The generation next?*" *Computers & Chemical Engineering*, vol. 147, p. 107 252, Apr. 1, 2021.

[10] L. T. Biegler, "Recent advances in chemical process optimization," *Chemie Ingenieur Technik*, vol. 86, no. 7, pp. 943–952, 2014.

[11] M. Martin, R. Gani, and I. M. Mujtaba, "Sustainable process synthesis, design, and analysis: Challenges and opportunities," *Sustainable Production and Consumption*, vol. 30, pp. 686–705, Mar. 1, 2022.

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

[13] M. Filho, E. Pimentel, W. Pereira, P. H. M. Maia, and M. I. Cortés, "Self-adaptive microservice-based systems - landscape and research opportunities," in *2021 International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, May 2021, pp. 167–178.

[14] J. Figueira and C. Coutinho, "Developing self-adaptive microservices," *Procedia Computer Science*, 5th International Conference on Industry 4.0 and Smart Manufacturing (ISM 2023), vol. 232, pp. 264–273, Jan. 1, 2024.

[15] X. Cheng, "Performance, benchmarking and sizing in developing highly scalable enterprise software," in *Performance Evaluation: Metrics, Models and Benchmarks*, S. Kounev, I. Gorton, and K. Sachs, Eds., Berlin, Heidelberg: Springer, 2008, pp. 174–190.

[16] F. Aydemir and F. Başçiftçi, "Building a performance efficient core banking system based on the microservices architecture," *Journal of Grid Computing*, vol. 20, no. 4, p. 37, Nov. 2, 2022.

[17] A. Lee *et al.*, "The idaes process modeling framework and model library—flexibility for process simulation and optimization," *Journal of advanced manufacturing and processing*, vol. 3, no. 3, e10095, 2021.

[18] M. L. Bynum *et al.*, *Pyomo - Optimization Modeling in Python, 3rd Edition*. Springer, 2021.

Appendices

Chapter A. Literature review

The literature review is attached and begins on the next page.

ENGEN582-24X
Honours Research and Development Project

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

Caleb Archer

Supervised by Tim Walmsley, Mark Apperley

Contents

1. Introduction

The world of process systems engineering has seen substantial growth since its burgeoning in the 1950s [1], creating and serving critical tools for chemical engineers working in industry and academia alike for improving the performance of resource-intensive processes. Process modelling, simulation and optimisation software solutions have played a substantial part in this journey. Over the same period, the nature of software engineering has evolved rapidly, with new paradigms swiftly overtaking their forebears, and pushing forward our understanding of what software can accomplish.

One such paradigm, cloud computing, has positioned itself as critical to the efficient and effective operation of businesses around the world. It is the goal of this literature review to explore how cloud computing oriented software architectures could be better aligned with the process engineering software space to improve upon the performance, capability, and accessibility of such software solutions. Specifically, the research questions to be answered by this review are:

- What architectures are available for deploying and running systems on cloud environments?
- What challenges and limitations are being faced by process simulation software?
- What tools or techniques have been used for assessing the performance of different software architectures?

Firstly, this review explores the space of software architecture, delving into the ways in which software systems are designed and arranged, how they have evolved, and what purpose each of these serves. Next, the state of process engineering software will be assessed, including interactive (user-interface based) and programmatic (computer-based mathematical modelling) approaches to performing simulation, modelling, and optimisation of chemical processes. The purported benefits and drawbacks of each approach will be discussed, as well as any wider issues with the area overall. Following this, we will determine how self-adaptive computing has influenced modern distributed computing, and what systems are actively being used in this realm. Finally, the field of software performance modelling and testing is be evaluated, focusing on how the associated techniques can be used to assess the performance of implemented software architectures.

2. Software deployment architectures

For various reasons, new software architectures evolve out of others, often substantially changing the nature and behaviour of the underlying systems, as well as the problems faced by engineers working with said architectures. The appropriateness of a given architecture can be considered dependent on the needs of the business or entity deciding on it.

2.1 Evolution of modular software systems

Parnas, in his seminal 1972 paper, described modularity as a means of improving the coherence and future changeability of a system [2]. Systems should be able to be decomposed into modules such that teams can independently contribute to the system without running into conflicts. These modules should have little knowledge of how other modules have been designed, and usage of other modules should be easily interchangeable. Applying these ideas, as described by Parnas, should lead to less engineering effort over time, ease of refactoring the target system and ease of understanding the system. The examples of applying modular design to a system focused on standalone programs, where one module is one isolated section of code. Despite this contextual view, these principles can and have been applied to systems since the publishing of this paper.

Parnas *et al.* clarified these principles of modularity in a later 1985 paper [3], after identifying a gap in the industrial application of software engineering practices versus those discussed in academic contexts. They took a non-trivial software system that already existed and redesigned it using modularity principles in order to compare the result to the original, rather than justifying the principles via small-scale, toy examples. Through their exploration, they emphasised the specific idea of information hiding, as well as the use of specific module creation guides for projects to help engineers to use information hiding within the context of a specific project (or sub-module). Though this paper also focused on standalone (but internally decomposed) software systems (as in [2]), it showcased the practical importance of constructing a system in a layered fashion, with thin connections between those layers. McGregor evaluated the modularity that can be achieved with the use of the model-view-controller design pattern, which separates program logic into modules that handle system interaction, data interaction, and the space in between the two. They emphasised the use of diagramming and related techniques to determine the breakdown of a system prior to beginning development [4].

The previous papers, however, spent little time exploring how a system decomposed into modules could contribute to reusability within the system, which was addressed by Almeida *et al.* [5]. To promote repeated usage of system components and modules, they constructed a guiding framework for reusability called RISE (Reuse in Software Engineering). This framework sought to define the different aspects of reuse, not only from a technical standpoint, but also an organisational one, including the reuse of knowledge and processes. Application of these ideas was purported to speed up the overarching engineering process, as described by Parnas [2] and further evidenced by Parnas *et al.* in 1985 [3].

Sifakis explored in more detail the interaction of software components across potentially distributed systems in [6], defining a formal method for verifying the properties and behaviours of such interactions. Though it approached system composability from a formal, mathematical view, it contributed to expanding the domain of defining modular systems from a standalone software system to a distributed one.

As an example of a more concrete step towards distributed modularisation of systems, Goncalves *et al.* depicted the process, and related impacts, of refactoring a monolithic software system into a more modular one with clear domain separation, which were all originally addressed by the monolith [7]. However, the study completed only represented the intermediary step of standalone modularisation,

without expanding towards the ultimate stated goal of moving towards microservices.

2.2 Evolution of parallel computation

Parallel computing involves the simultaneous processing of a defined algorithm, such as simultaneously calculating the gradients for a set of differentiable equations, as opposed to calculating them one-by-one, which would be defined as sequential computing.

A 1982 survey by Haynes *et al.* was conducted regarding the state of parallel computing at the time [8]. It showed that parallel systems were, for the most part, designed with esoteric chip architectures to maximise the amount of parallelism that could be achieved. Along with this, to handle large-class problems that a single parallel machine could not solve on its own, a large number of these machines were networked together. Such systems were considered to have grown beyond the traditional single-CPU system capable only of attending to tasks in a serialised fashion. In progressing to 2006, this view of parallel computing became multi-faceted, with Asanovic *et al.* addressing the arrival of multi-core computation in retail hardware [9]. Their view was that multi-core systems would face similar challenges to traditional parallel computing systems, and that instead small-scale "manycore" systems would be a better target, though this has not eventuated in the context of modern CPU architectures.

In response to the aforementioned growing presence of multi-core retail computer systems, a study towards converting existing sequential programs to partially parallelised versions [10] was performed in 2007. Bridges *et al.* argued that the arrival of affordable multi-core systems did not come with a similarly increasing utilisation of these additional cores, attributed to the increased complexity involved in manually defining parallel processes. A framework was used in demonstration of thread extraction of a benchmarking suite, increasing the performance of this benchmarking software by several times. Asanovic *et al.*, in a later 2009 paper [11], continued to push for adoption of "manycore"-centric programming models, stating that a large bottleneck in reaching full exploitation of additional cores on microprocessors is the programmer, who is not typically acquainted with writing programs in a parallel fashion. Their proposal, similar to Bridges *et al.*, was to construct an autotuner for introducing parallelism to existing sequential software. However, as shown in [12], some programs are easier to parallelise than others, and may not see the expected gains as shown in examples of this particular paper.

Chen *et al.* provided a survey on parallel computing [13], its architectures, methods and algorithms, performance, and its applications. At a high level, there are two types of parallel computing architectures: the Common Parallel Computer (CPC), and the Supercomputer. CPCs are relatively cheap, commonly available computers that can be found in the home, at workstations, and in pockets, with multi-core CPUs. Supercomputers, on the other hand, are far less accessible, but still demonstrate far greater computing power capability for massively parallel tasks. This may be a simplistic model, especially with the entrance of the GPU (Graphics Processing Unit), as discussed by Xu *et al.*, where they focused not on the architecture of parallel computing, but general styles (or scales) in which many different architectures can fit [14]. These styles include internet, datacentre, process and accelerator scales of parallel computing. The GPU may be considered an alternative actualisation of general "manycore" computing as introduced previously by Asanovic *et al.*

The refactoring of an electric-market simulation software towards multi-core parallelisation [15], and subsequent results, was shown by Seveso *et al.* The application was originally written and run in a sequential fashion, but faced performance limitations as its usage was expanded for simulation scenarios of greater complexity, where the time taken to undertake a simulation exceeded acceptable runtime boundaries. Substantial improvements to some aspects of the simulation software were seen, but the authors neglected to provide a full analysis of simulation time between the sequential and parallel ar-

chitectures.

Vivas *et al.* provided a better characterisation [16] of performance differences between different architectures. Both HPC (High-Performance Computing) and non-HPC styles of parallel computing are available as choices for researchers conducting scientific computing. Non-HPC systems can be made up of existing computing hardware available within a research institution, serving as an alternative to purchasing dedicated supercomputing hardware. Though it was found that dedicated HPC hardware performed well beyond each of the networked clusters, it would have been beneficial to provide a quantitative assessment of reliability across these different architectures.

With parallel computing now more accessible by default via multi-core retail computing systems, as well as relatively inexpensive dedicated parallel processors (such as the GPU), it has become substantially more viable for software systems built for general hardware to take advantage of parallelism.

2.3 Evolution of distributed systems

Distributed systems are software systems made up of a network of potentially heterogeneous computing hardware (nodes), where the tasks running on each node may not all be the same as every other node. The rise of networked devices has enabled software systems to move beyond the format of a standalone application running locally on one machine. Such distributed systems may even be geographically heterogeneous, as is seen with cloud computing. This paradigm can be viewed as an evolution of both parallel computing and modular software design, attempting to take advantage of additional computing hardware in order to construct more powerful, sophisticated systems. Initially, these systems would have entered the computing domain with each software service requiring an entire machine with one operating system to run securely, but new technologies arose to handle computing resources efficiently.

2.3.1 Virtual machines

Virtual machines are entirely virtualised computers with virtual computing resources, including CPU, memory and storage, assigned to them [17]. A single physical computer can often run several virtual machines, where software running inside the virtual machine has no knowledge of software running in others.

As indicated by Goldberg in their 1974 survey of virtual machine research [17], virtual machines were originally developed to handle the limitations of a rigid kernel interface, where it was difficult to test the behaviour of kernel-level software, or entire operating systems, which require direct access to all functions of the underlying system. Virtual machines enabled this direct access without having to repeatedly move between compatible systems; an entire operating system could be simulated via another base system, and with full resource isolation (CPU, memory, etc.). An early issue identified with virtual machines is the performance overhead, a natural consequence of having two or more kernel layers involved in running virtualised systems.

As time progressed, virtual machines saw their use expanded beyond mere simulation, compatibility and testing purposes. Whitaker *et al.* detailed a relatively new use-case: distributed networked applications [18]. They argued for the usage of lightweight virtual machines to make a service available to a network, which they considered to be more viable than allocating dedicated physical machines to each service (which may require resource isolation for security and performance purposes). With individual computers being capable of running multiple virtual machines, they saw this novel multi-tenancy service model as being newly reasonable. However, performance and storage overheads from virtual machines may be a largely unavoidable issue. Li *et al.* also identified performance overheads

as a prominent issue with virtual machines that has seen a great deal of attention within research [19]. As well as this, there has been great emphasis on migration of virtual machine images and resource sharing.

With the advent of cloud computing, virtual machines have played a significant role in providing infrastructure as a service [20], where individuals and organisations can rent virtual machines with defined CPU, memory and storage resources from cloud providers. Hardware may have previously been accessible in dedicated form, typically on-premise, but cloud computing has seen this shift to a distributed form of hardware access. As described by Sharma *et al.*, virtual machines enabled the construction of virtual computing clusters, where seemingly independent machines can be virtually networked together to serve some collective goal. The hard resource limits on virtual machines prove to be problematic when resource efficiency is a desired system attribute, which was previously discussed in [19].

Pietri *et al.* addressed this resource allocation issue directly, discussing research focused on mapping virtual machines to physical hardware [21], especially how virtual machines can be dynamically resized to serve variable application resource requirements. It was pointed out that part of the difficulty of this problem is the computation required to perform any reallocation, requiring carefully designed algorithms for deciding when reallocation should occur. Xiao *et al.* presented such a dynamic allocation strategy in [22], using predictions of future resource requirements to decide when additional virtual machine instances are needed. Conversely, a study of virtualisation history by Randal found that even with these hard resource limits said to provide system isolation, virtual machines cannot be considered equally as secure as systems deployed on isolated hardware, given their proven susceptibility to recent memory-based weaknesses [23]. Despite this, virtual machines continue to serve as foundational tools for cloud computing services.

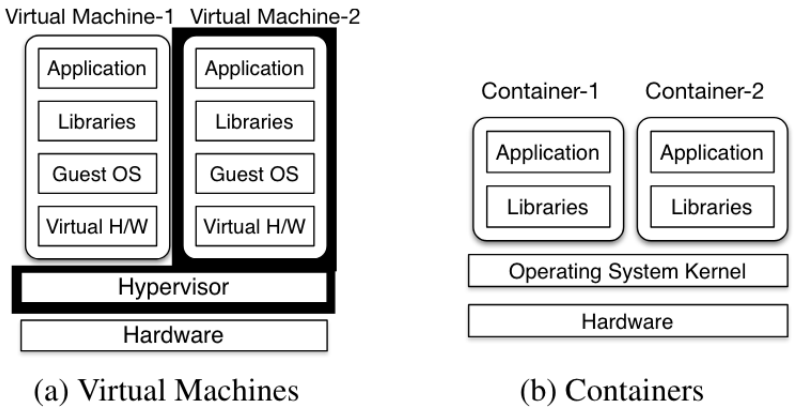
Virtual machines improved the scene for software deployment substantially, allowing for services to be run on operating systems that would otherwise be incompatible, and paved the way towards cloud computing via secure multi-tenancy. Though they serve as a critical stepping stone towards efficient resource utilisation, their typically large nature prevents systems from taking full advantage of the underlying hardware.

2.3.2 Containers

Containers have emerged as popular alternatives to full virtualisation as seen with virtual machines, and instead rely on strong filesystem access control and process separation to provide a facade of full environment isolation [23]. Their more lightweight nature has seen a presence takeover of containers for direct software deployment purposes [20], where previously an entire virtual machine may have been dedicated to a single software service [18]. Instead of each service being bundled with an entire operating system, as with virtual machines, containers allow for a shared operating system to be used under-the-hood, while separating the required user-level libraries and dependencies into isolated filesystems (see Figure 1).

The utility of distributed cloud computing for bioinformatics was addressed in [24]. The need to share information between biomedical institutes and centres has seen the resource sharing and centralisation benefits of distributed cloud computing receive attention, where the traditional approach has been isolated self-management of required computing infrastructure. Provisioning of computing resources, whether via dedicated hardware, virtual machines or containerisation, is now a trivial task. At the same time, some risks of a move to cloud computing were acknowledged, including data security and dependence on third-parties to deal with threats to infrastructure.

Pahl *et al.* assessed the state of containerisation research and current focuses and prominent use-cases



(a) Virtual Machines (b) Containers

Figure 1: Virtual machine and container stacked architectures [20]

[25]. It was determined that containerisation is heavily present in the software deployment space, being used to package applications to serve to end-users, and being done so across different technical fields of interest. Along with this, they also found a general immaturity of automation of containerised system management, but container orchestration is increasing in usage in order to lower resource usage costs and improve system performance. There seems to be more focus on the flexibility of containerised systems rather than performance, though these two elements can be interlinked. As steps towards improving container management automation, several heuristic techniques for container scheduling and allocation are described in [26], though by nature, these approaches may not be globally optimal, and require consideration of context to drive a system towards superior performance.

The resource utilisation and management benefits of containers were surveyed and explored in [27], within the context of edge and fog computing, rather than just traditional cloud environments. They stated that virtual machines have recently dominated the computing resource allocation domain, but containerisation continues to expand its presence, especially with the substantially lower overhead required to provision a set of containers compared to one of virtual machines. However, a known issue is that many containerised systems are themselves deployed within virtual machines, creating potentially significant overheads for container to host communication, though this is complemented by taking advantage of resource isolation provided by the host virtual machine [20]. As well as this, containers typically do not adhere strictly to resource usage limits defined by runtime managers, in opposition to standard virtual machine behaviour. Vaño *et al.* elaborated on the use of containerisation technology [28], outlining some elements of public cloud computing that have been adopted in edge computing. While several container management technologies at the edge look promising, such as K3s or MicroK8s, they point out that problematic attributes of containers, including image size storage requirements and minimal application isolation, serve as challenges to adoption at the edge.

Containers are powerful enablers of efficient resource utilisation without any redundant performance and storage overheads. Subsequently, this nature has allowed for distributed software systems to consist of machines running many more services than before, while retaining clean environment separation.

3. Process systems engineering software

At a high-level, there are two general approaches to performing process modelling, simulation and optimisation. The most common approach used in industry is interactive process modelling, involving locally available GUI (Graphical User Interface) software. An approach with more direct utilisation in academia is mathematical modelling, where a model is carefully defined in code.

3.1 Interactive process modelling

Process simulation and optimisation software is heavily used on an interactive basis to enable easy access to modelling of processing plants [29], with tools such as Aspen Plus and Aspen HYSYS serving as common examples of these interactive environments. However, these process modelling tools often lack the models and solvers required to handle some simulation, optimisation or synthesis problems. Such problems need to be solved using less intuitive tools such as Python or MATLAB, or via modelling languages like Modelica and GAMS. Most commercial process modelling tools, including Aspen HYSYS, Aspen Plus, and ProSIM, make extensive use of sequential-modular flowsheet solving [30]. These tools are typically locally run on machines of end-users.

The use of such process modelling software has been validated in both research and industrial contexts [31], where experimental data is often compared with model outputs to determine the level of agreeance. Models found to be accurate can subsequently be used to experiment with different process parameters and configurations and observe downstream and high-level impacts [32]. While not perfect representations of the underlying process, analysis via model usage can be performed quicker than that of experimental analysis, which would also involve risky live changes to plant operations. As depicted in [33], interactive process modelling tools can ultimately be used in an economic optimisation sense, such as identifying points in a process where resource usage can be increased or reduced to improve system efficiency and reduce operating costs.

Klatt *et al.* indicated the importance of process modelling, simulation and optimisation tools and frameworks being easily accessible to non-experts (working in the process industry) within interactive packages [1]. The process engineering industry is heavily dependent on these tools, in interactive form, to apply modelling and optimisation techniques to their specific contexts. The needs of these industrial practitioners, however, differ from those of researchers. The time required to perform computational optimisation [34] of a process, for example, is a deciding factor for whether certain models or tools are utilised in practice.

3.1.1 Sequential modular simulation

Sequential modular simulation is the flowsheet solving and optimisation method most commonly used by commercial process simulation software [35]. It solves different parts of a process flowsheet in a serial fashion, processing each "module" without consideration for the entire system.

A 1993 study by Harrison [12] of the parallelisation of a sequential-modular simulation algorithm on a supercomputer found that, although some reduction of computation time can be achieved, these improvements are "modest" at best. More complex models were better improved by parallelisation techniques, but the differences were not magnitiudinal. The relatively early nature of this study means modern improvements to sequential modular algorithms and increased uptake of multi-core workstation computers could not be reflected in the improvements shown, but the findings still have implications for sequential-modular simulation today. A similar conclusion is provided by Ralphs *et al.* in their overview of the parallel solving space for MILP problems (mixed integer linear programming) [35]. They state that further magnitudinal improvements to sequential MILP solving algorithms are

highly unlikely, making room for further emphasis on parallelised approaches, which have otherwise been stunted by the emphasis on said sequential solving improvements. Exploration of parallel MILP solving methods is challenging, as existing sequential solvers behave in ways that are fundamentally incompatible or problematic for effectively carrying out parallelisation.

Further exploration of sequential-modular simulation is shown by Kulikov *et al.* [36], where it is applied to the solving of particulate process flowsheets. Standardised variants of this algorithm have traditionally been applied to fluid operations. An process simulation tool called CHEOPS is applied to make use of solvers from various existing toolkits. Their results demonstrated sequential-modular simulation could be applied via existing products and libraries in achieving a fine-grained model result for non-standard process problems. However, this approach, depending on the problem, may require access to several commercial tools with prohibitive licensing costs. The authors also identified that the required computation time, despite efforts to reduce this, is still a problem, due to the underlying iterative nature of sequential-modular simulation. They posit that algorithmic improvements can improve simulation efficiency, though, as identified previously by Harrison [12], a high level of improvement may be doubtful.

A performance assessment of a parallel-modular optimisation algorithm was conducted in [37], an adaptation of existing sequential solving algorithms. Message-passing was utilised across a distributed set of computation nodes, which received instructions via MIMD (multiple instruction, multiple data). A significant performance improvement proportional to the number of nodes used was observed, though little improvement was seen beyond 14 nodes. It may be difficult to generalise this solution, with customised usage of parallel programming libraries required, and the iterative nature of the underlying sequential optimisation still serves as a limiting factor.

While serving as the core driver of most interactive flowsheet optimisation, sequential modular simulation is seen as having little opportunity for performance improvements, and cannot be used for solving certain optimisation problems.

3.2 Mathematical process modelling

With improvements to sequential-modular solving and optimisation being few and far between [35], other mathematical modelling approaches are starting to present themselves as more viable tools for industrial use [38]. An increase in general available computing power has made previously intractable (or otherwise prohibitively expensive) problem formulations possible to solve, especially with respect to Mixed Integer Non-linear Programming problem (MINLP) formulations. However, global optimisation of processes still presents a challenge for researchers, and so the field has continued to shape leading up to wider industrial application. These direct mathematical formulations require deep understanding of each underlying problem and the techniques needed to correctly formulate said problem [39], making generally accessible mathematical modelling approaches generally inaccessible to non-expert users. De Tommaso *et al.* affirmed the "solid theoretical background" required to perform process simulation, but ignored the significant industrial audience these tools have [40]. Even with a general modelling approach defined, the requirement to have good understanding of computer programming and mathematical modelling proves problematic to wider usage of these techniques. Martín *et al.* additionally identified the issue of open access to models [39]. Advances in process modelling research have not been evenly accompanied by availability of the newly-created or modified models, particularly with respect to sharing code.

Increases in computer processing power have triggered further development of mathematical modelling techniques such as superstructure optimisation [41]. As compared to sequential solving methods (such as sequential-modular simulation), superstructure optimisation attends to the entire flowsheet simulta-

neously. The approach struggles to reach industrial usage due to prior knowledge needed to initialise the model, as well as computational difficulty. Such a method for performing process systems synthesis was described by Duran *et al.*, involving a Mixed-Integer Nonlinear Programming (MINLP) formulation [42]. A superstructure master problem is defined, where subsequently derived sub-problems are solved recursively. This algorithm, while somewhat generalisable, only consistently applies to convex functions (such as a function of a quadratic shape).

According to Klatt *et al.*, early efforts towards mathematical modelling within process engineering were conducted in an exploration of how the computer could help solve substantially sized flowsheet models [1]. This research was performed as a response to otherwise ad-hoc design thinking that was often applied to engineering process systems at the time. Despite a long history of research, these high-performance models are sparsely applied in industry. Such models could be utilised for real-time optimisation, but once again, find themselves underrepresented in practical usage. For greater prominence of these numerical methods to arise, they would need to be made available to non-expert users via existing or new interactive process modelling software.

Nayak *et al.* described the creation and design of an open-source, equation-oriented process modelling framework [43] called OpenModelica, which uses a process modelling language named Modelica. It consists of a pre-defined set of model components and functions which can be re-combined together to construct context-specific models. Though it may be very useful for educational and research purposes, the framework does not address the issue of the skill required to utilise modelling frameworks and libraries in general, and is unlikely to see industrial applications.

3.2.1 Equation oriented simulation

Equation oriented simulation is commonly placed in opposition to sequential modular simulation, as it is encompasses simultaneous solving and optimisation approaches, though it is less available in interactive form. Within a broad overview of Process Systems Engineering history, [44] indicates that equation-oriented simulation has widened the scope for optimisation of process engineering activities, and beyond, as compared to its sequential-modular counterpart.

Biegler continued this thought [30], saying that despite the overwhelming popularity of sequential-modular flowsheet solvers, they are overly sensitive to convergence failure due to dependence on gradient approximations, and therefore limit the subset of optimisation problems that can be solved. Equation-oriented solvers, on the other hand, create such a speed-up that they can be used for real-time optimisation (RTO). Biegler stated that equation-oriented simulation and optimisation was not common in commercial flowsheeting software, in part because of an unfilled need for accurate first and second-order gradients.

In [45], in an attempt to improve the usage of equation-oriented optimisation for solving large, complex flowsheets, made use of automatic differentiation to obtain precise first and second derivatives that are needed to ensure model convergence. The experiment was performed on an Intel-based computer, on which they reported an average of 15 CPU-minutes of computation time. While they achieved results consistent with an existing Aspen Plus optimiser, no comparison of the time complexity involved in the calculations was provided. Generalisable GPU-based (Graphics Processing Unit) equation oriented optimisation was tested in [46]. They found that GPU usage could offer performance speed-ups when compute utilisation is high, but smaller flowsheet models performed worse than their traditionally CPU-calculated counterparts due to lower compute utilisation and model complexity. Their experiment involved the testing of one model at a time, but did not consider the possibility of testing individual models simultaneously.

DAE Tools, an equation oriented process modelling framework for Python was proposed in [47]. It provides general capabilities for solving model-based mathematical simulation and optimisation problems, which can then be made embedded as part of offline and online user applications. Although it supports parallel computation, this is only in the form of generated code to be separately run on tightly-coupled parallel systems involving several nodes.

With the goal of attaining performant global optimisation, [48] described a hybrid approach to solving a flowsheets for phase equilibrium models, involving a combination of sequential-modular solving and equation-oriented solving methods. The experiment made use of in-house optimisation tools. The approach depicted was successful in meeting requirements, but requires good problem-specific model formulation and niche knowledge requirements in order for the approach to be usable in other contexts.

The work in [49] applied pseudo-transient (PT) equation-oriented optimisation to flowsheets involving dividing-wall distillation columns. Prior attempts at using equation-oriented optimisation suffered from the need to initialise models with good initial starting points to allow for convergence, considered to be a difficult task. Use of pseudo-transient modelling allows for the range of viable initial starting points to be expanded. The study found that this overall method enables consistent, rapid model convergence of the entire flowsheet for a model, and better enable cost reductions for a given process design problem. A review of PT modelling advances in equation-oriented simulation [50] showcased at a broader level the impact that PT approaches have had on viable and efficient optimisation. PT models can better provide exact solutions for a given optimisation problem, and reduce the work required to initialise a model. Despite this progress, challenges remain for making PT modelling widely feasible. The review outlined that PT modelling has not yet made its way into commercial simulation software due to the difficulty of implementation, and work is ongoing to make PT approaches more computationally feasible. It can be said that, while there has been a great deal of progress made in making equation-oriented simulation a true competitor to the status quo, the lag between research and implementation makes accessibility to this progress challenging.

What can be observed for interactive and mathematical modelling approaches is that there is a performance and accessibility trade-off. Equation-oriented simulation, and general mathematical modelling approaches, are much more performant and reliable modelling and optimisation tools compared to those commonly used in interactive process engineering software. However, they are generally far less accessible to industrial users because of their required expertise and deep foreknowledge of how the underlying models work. They are sparsely available in interactive form, which is, in part, due to ongoing research to make them more generalisable. The interactive approaches to process modelling, which are used far more extensively in industry, suffer from over-reliance on inherently performance-limited solvers, preventing the industry from taking advantage of accurate optimisation at speed. A wider problem with process simulation software is that, even if more performant solving methods are available, these are limited by the localised nature of the desktop software operating environments.

4. Self-adaptive approaches to distributed computing systems

Self-adaptive computing, which addresses self-regulating systems, is a field that has received implicit attention from distributed computing technologies. The advent of microservices and their common modes of implementation have caused this merging of paradigms.

4.1 Foundations of self-adaptive systems

Inspired by the many subconscious and self-regulating functions of the human body, Kephart *et al.* described autonomic computing, an aspirational paradigm with the goals of having computer systems perform self-management, self-configuration, self-optimisation, self-healing, and self-protection [51]. This was in response to the increasing complexity of software systems and the effort involved in maintaining them. The authors acknowledged that this high-level paradigm depiction demands a great deal from engineers and researchers, and also exacerbates the consequence of human-caused error, as errors would instead be made at the goal specification level, having effects on the entire self-managing system.

Weyns provided a more concrete realisation and evolution of the paradigm described by Kephart *et al.*, with a presentation of the history and current state of self-adaptive systems [52]. Weyns affirmed that the complexity involved in managing modern software systems has increased substantially, and that there is a need for such systems to be able to manage and configure themselves autonomously, thus the entrance of the term, *self-adaptive systems*. Over time, self-adaptive system models have evolved from task automation to a much more involved management system responsible for a lower-level sub-system. The conceptual model (as seen in Figure 2) described consists of four components: the environment, the managed system, the defined adaptation goals and the managing system. The most recent iteration of self-adaptive systems has been the implementation of control theory inspired systems, where a feedback-loop drives a managed and monitored system towards its specified goals. Weyns notes that a challenge for this field of research is the empirical validation of the proposed models and strategies, as well as whether the latest wave of self-adaptive systems (rooted in control theory) can be demonstrated as a good foundation for self-adaptive systems in practice.

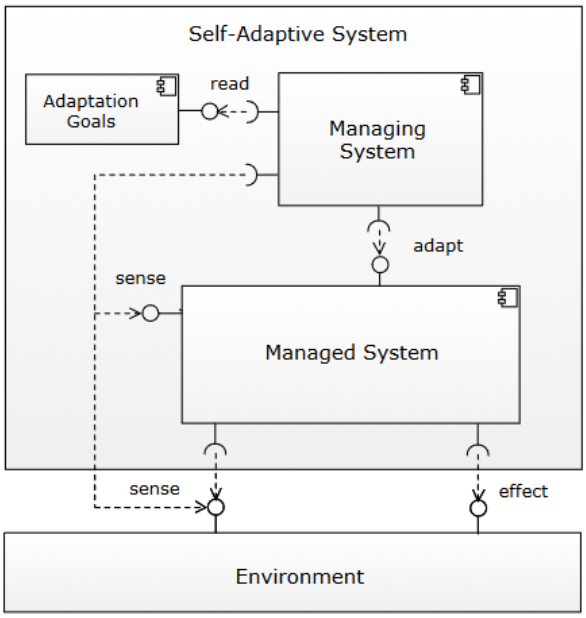


Figure 2: Conceptual model of a self-adaptive system by Weyns [52]

These ideas, while not firm in how they are to be implemented or the scope of implementation, provide a basis for systems to be designed from the very beginning with consideration for reliable performance, with the expectation that they can handle interruptions provided by a volatile external environment.

4.2 Microservice architectures

A microservices architecture was defined by Fowler *et al.* as an approach to building modular software systems with well-defined domain boundaries [53], which is more suited to enabling evolution of the underlying design. Microservices draw from principles of modular software design [2], and expand this idea from focusing on the modularity of functions, classes, and other code-level components, to a higher-level system component level. They are a response to bulky monolithic systems, where, for example, all back-end logic of an application is centralised in one place, and as such, any change to that back-end requires a complete redeployment of all instances of the monolith [54]. The decoupled nature of microservices encourages a distributed deployment of these systems, serving one of the motivations of the architecture: resilience to failure [53].

These motivations were confirmed by a survey by Viggiato *et al.* [55], where benefits such as independent deployment, ease of scalability, maintainability and technology flexibility were identified by respondents as having some or great importance. It may have been better, however, to ask more open-ended questions around the benefits of microservices instead of the very guided questions reported. Taibi *et al.* also added a motivation of having no other choice with regards to handling system complexity [56], though their survey sample size was small.

Gravanis *et al.* presented a counterargument for the adoption of microservices [57], stating that a rapid adoption of this newfound architecture showcases its own problems (including the complexity of distributed communication, as mentioned by Viggiato *et al.*), and that the monolith can still stand as a valid choice of architecture for businesses. While Fowler *et al.* see decentralised data management as advantageous, Gravanis *et al.* identify this as a non-trivial flaw of the architecture, including issues such as inconsistency of data across a system. However, the primary point made is that choosing a particular software architecture should be informed by the alignment of the intended nature of a system and the benefits provided by said architecture, rather than a dogmatic selection based on current trends. The need for a performant system and the scalability provided by microservices is one such alignment. Scalability was re-oriented as a core feature of microservices by Dragoni *et al.*, with other desired system attributes such as performance, availability and resilience made possible through scalability [58].

An example of premature architecture selection is seen in [59], where a third-party Kubernetes control plane component was initially designed as a set of microservices, but was eventually converted into a traditional monolith after failing to reap the often promoted benefits of microservices. This case for moving to a monolith, however, seems to be rooted in an initial misalignment of the concerns of the developers of the system, versus those who would actually install and deploy said system, which was briefly addressed by Mendonça *et al.* Jamshidi *et al.* instead see the distributed nature of microservices as being a point of caution, requiring attention paid to how well a system under this architecture performs self-healing, which is a non-trivial endeavour, as failure within a distributed system itself becomes potentially distributed [54].

Self-adaption has, over time, been applied in increasing amounts to microservice architectures, as shown by Filho *et al.* [60], where various papers applying different aspects of self-adaptation were analysed. Although this transition has been quite plain, Filho *et al.* seem to ignore the obvious mainstream application of self-adaptation to microservices, Kubernetes, which Jamshidi *et al.* identified as being part of the fourth wave of microservices usage [54].

In an exploration of how microservices perform compared to monoliths, Al-Debagy *et al.* collected experimental results for a small example application with two variants for the two architectures [61]. They did not find any substantial performance differences between the two architectures, though little description of the test scenario was provided, and the entire experiment was run on a single system. These results are contradicted by Aydemir *et al.*, where in [62], a sophisticated banking application was re-designed as a microservices-based system, with the central goal of attaining a more performant system in mind. A monolithic application which originally handled bank accounts, transfers, loans and customer management was restructured to have a microservice for each of those four banking elements. The final system was found to handle substantially higher throughput with less resource utilisation.

When applying principles of modularity to the design of a reliable distributed system, with a need for various independent components to be well-integrated, microservices can be seen as a natural starting approach. However, on their own, reliability cannot be entirely attained without a well-designed approach for handling failure.

4.3 Frameworks for developing self-adaptive distributed systems

While there is a large range of esoteric frameworks and systems available for deploying microservices architectures and ensuring the various self-oriented properties of self-adaptive systems, the two most prominent of these in the literature are Docker and Kubernetes.

4.3.1 Docker

Docker is a container runtime and management engine, making easy the process of creating and running custom containers, as well as using container images published by other individuals or organisations to an open repository of community-made images [63]. It is available for use on all major operating systems for software development, including Linux, macOS and Windows.

Docker was used in a performance comparison between a monolithic architecture and microservices architecture enacted by Al-Debagy *et al.* [61]. Its utilisation in this small-scale experiment can be explained by its ease of use and general pervasive availability on different systems. Saquicela *et al.* use Docker in a more sophisticated fashion, using it to build and run various containerised (third-party and custom-made) microservices, such as a load balancer, service discovery provider, and a Redis database.

With containers being one of the primary means of deploying individual microservices [65], Docker is naturally used by default, as it played a major part in the evolution of container runtimes, building on the foundational container technology of LXC (Linux Containers), and contributing to standardisation of the container image format, OCI (Open Container Initiative) [23].

4.3.2 Kubernetes

Kubernetes is a container orchestration and management system [66] designed to coordinate container-based workloads across multiple machines (a cluster) with ease. A control plane oversees the rest of the cluster, monitoring for failure or poor resource utilisation, restarting failed pods (each pod is a group of tightly-coupled containers) or instantiating more or less of a given pod based on system load. Kubernetes is one of the more concrete mainstream implementations of a self-adaptive system, as outlined by Weyns [52].

As part of their testing of a microservices architecture for a banking system, Aydemir *et al.* deployed their system using Kubernetes, allowing for the system to adapt to variable levels of throughput generated by their load testing tool. [62]. Figueira *et al.* specifically discuss developing self-adaptive

microservices, and describe a system architecture built on top of Kubernetes for its scalability and availability attributes. Randal further presented the picture of Kubernetes as being the mainstream choice for container orchestration, in part for its ability to reduce the complexity involved in managing a distributed software system [23].

While it has not been defined explicitly as the culmination of self-adaptive computing and distributed computing combining, Kubernetes is targeted towards providing engineers with the ability to construct systems capable of autonomously maintaining attributes such as performance and reliability, especially in the context of microservices.

5. Software system performance analysis

When assessing software systems and their respective configurations, there is a range of approaches that can be used. The literature reviewed can be categorised into empirical performance testing and performance modelling. Empirical performance testing involves the experimental collection of performance-related data using active and passive system probing techniques. Performance modelling addresses theoretical results obtained via representations of the underlying system.

5.1 Empirical performance testing

Cheng emphasised the use of appropriate KPIs (Key Performance Indicators) [68] for thoroughly and properly assessing the performance of enterprise software systems. These KPIs can be assessed with variations of end-to-end testing, simulating the behaviour of both a single user (performing some sequence of actions) and multiple users for wider load-testing experiments. With these KPIs, the impact of different optimisations (such as configuration tuning, adding additional hardware, architectural changes, etc.) can be measured by comparing KPI changes before and after experiments. However, Cheng suggested using performance tests for determining explicit hardware allocation guidelines, which may no longer be appropriate when striving for autonomous system management. These guidelines would also fail to consider dynamic system load as a result of variable user activity.

Chuang *et al.* use the Selenium test automation framework to perform active testing and monitoring of a data analysis system built on Kubernetes [69]. They compared the response times of different user requests, and the CPU throttling effects on different system components, before and after configuration tuning. After tuning, they were able to observe a decrease in CPU throttling. It could be considered problematic for accurate performance measurement when the testing tool (Selenium) is run from inside the clustered system being tested, rather than from an external host with separate resources.

The usage of load-testing was expanded upon in [62], where a legacy monolithic banking application was compared against a new microservices architecture deployed on Kubernetes. Apache JMeter was used to create scenarios of between 500 and 5000 virtual users interacting with the banking system. Each load-test was conducted over the course of 5 minutes, with an initial ramp-up of 1 minute before reaching a maximum request-rate, and all were run from an external system. Two system-level metrics were collected during tests: throughput and response time. The impact of different Kubernetes configurations, however, was not considered.
This active testing approach is included in an expanded performance monitoring description given by Nambiar *et al.* [70]. They, however, preferred the usage of passive monitoring to observe performance impacts, as active monitoring or testing introduces additional load to the system subject to analysis. Their passive monitoring recorded properties of packets sent as part of client requests and server responses. It may be difficult to confirm the purported utility of the specific tool used by Nambiar *et al.*, as it is licensed software.

5.2 Performance modelling

An alternative to live testing, monitoring and analysis of a software system is to model or simulate it. This was shown in [71], where a Poisson modelling approach is used to analyze the performance of a serverless AWS (Amazon Web Services) Lambda application. Such an approach can avoid the complication of preparing a suitable testing environment that replicates real-world conditions. To validate the predictions made by the model, additional empirical analysis was conducted. They found that the model was well-aligned with actual data. From this, the utility of modelling a system interaction scenario prior to actual testing is supported.

Simpler pre-experiment modelling was performed in [72] (based on queueing theory), followed by similar subsequent experimental validation. A simple application was designed with both monolithic and microservices architecture variants. The JMeter load-testing tool was used to observe the impact of an increasing number of users on the average response time for each architecture configuration. While the data obtained helped to prove how testing can help make system structure decisions, the application was designed specifically for the study, and as such was trivially implemented, and may lack the complexity to realistically assess the validity of the data.

Woodside presented a case [73] for the integration and dual-use of both system performance models and empirically obtained data to inform decisions. While models will not capture actual data perfectly, they can be fit to previously obtained performance data and utilised for predictions of untested scenarios. However, this does require an established understanding of the underlying model used, as stated by Trivedi *et al.* [74]. Modern software systems are of greater complexity (and increasingly distributed), and thus require more sophisticated models to accurately represent system behaviour. This, in turn, makes the modelling process harder to understand, and as such, it is underutilised within industry. Trivedi *et al.* put that a deep understanding of modelling paradigms, such as queueing theory, Markov chains, and Petri nets, should not be required in order to make productive use of them.

Following review, it can be gathered that the utilisation of both empirical performance testing and performance modelling may be important. Empirical testing serves well to collect actual representations of how a system may perform under various conditions, while modelling techniques can provide predictive power and initial expectations of behaviour, which may be especially useful with complex systems.

6. Summary

A completed review of literature has demonstrated the development and current state for both the engineering of software system architectures, and process simulation software. What has been found is that there is a strong lack of crossover between the modern iterations of both fields. Restated are the original research questions, corresponding answers, and subsequent decisions made with respect to the project at hand:

What architectures are available for deploying and running systems on cloud environments?

A distributed, self-adaptive software system built on Kubernetes, running modular, containerised software components is most suited to deploying systems on cloud environments. The benefits of such an architecture were found to be performance, efficient resource utilisation, and system resilience. The flexible and lightweight nature of containers can serve the project's needs in terms of performance very well, and serve as the basis for all software components involved when deployed. Such flexibility can better enable the analysis of complex systems, especially when the analysis is multi-faceted. In a distributed fashion, the usage of containers will allow a deployed analysis system to efficiently take advantage of computing power provided by an array of heterogeneous hardware.

As such, the project will move towards the described architecture.

What challenges and limitations are being faced by process simulation software?

Interactive process simulation software primarily use solving methods that are challenging to parallelise, and most software is deployed in a standalone, local fashion that cannot take advantage of heterogeneous hardware resources. More performant and reliable mathematical modelling techniques are largely unavailable via such interactive software, preventing their extensive use in industry.

As such, the project will assess the specific performance improvements obtained by deploying an integrated process engineering software platform to the aforementioned architecture.

What tools or techniques have been used for assessing the performance of different software architectures?

Both empirical performance testing and performance modelling methods have been used to assess software systems, including active load-testing and passive monitoring as empirical methods, and Markov chains and Petri nets as modelling methods.

As such, the project will include the usage of well-defined performance modelling methods in addition to empirical analysis techniques to assess the performance of the developed software architecture.

References

[1] K.-U. Klatt and W. Marquardt, "Perspectives for process systems engineering—personal views from academia and industry," *Computers & Chemical Engineering*, Selected Papers from the 17th European Symposium on Computer Aided Process Engineering held in Bucharest, Romania, May 2007, vol. 33, no. 3, pp. 536–550, Mar. 20, 2009.

[2] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *Communications of the ACM*, vol. 15, no. 12, pp. 1053–1058, Dec. 1, 1972.

[3] D. Parnas, P. Clements, and D. Weiss, "The modular structure of complex systems," *IEEE Transactions on Software Engineering*, vol. SE-11, no. 3, pp. 259–266, Mar. 1985.

[4] J. D. McGregor, "Software architecture.," *The Journal of Object Technology*, vol. 3, no. 5, p. 65, 2004.

[5] E. de Almeida, A. Alvaro, D. Lucredio, V. Garcia, and S. de Lemos Meira, "RiSE project: Towards a robust framework for software reuse," in *Proceedings of the 2004 IEEE International Conference on Information Reuse and Integration, 2004. IRI 2004.*, Nov. 2004, pp. 48–53.

[6] J. Sifakis, "A framework for component-based construction," in *Third IEEE International Conference on Software Engineering and Formal Methods (SEFM'05)*, Sep. 2005, pp. 293–299.

[7] N. Goncalves, D. Faustino, A. R. Silva, and M. Portela, "Monolith modularization towards microservices: Refactoring and performance trade-offs," in *2021 IEEE 18th International Conference on Software Architecture Companion (ICSA-C)*, Stuttgart, Germany: IEEE, Mar. 2021, pp. 1–8.

[8] Haynes, Lau, Siewiorek, and Mizell, "A survey of highly parallel computing," *Computer*, vol. 15, no. 1, pp. 9–24, Jan. 1982.

[9] K. Asanovic, R. Bodik, B. C. Catanzaro, *et al.*, "The landscape of parallel computing research: A view from berkeley," Dec. 18, 2006.

[10] M. Bridges, N. Vachharajani, Y. Zhang, T. Jablin, and D. August, "Revisiting the sequential programming model for multi-core," in *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*, Dec. 2007, pp. 69–84.

[11] K. Asanovic, R. Bodik, J. Demmel, *et al.*, "A view of the parallel computing landscape," *Communications of the ACM*, vol. 52, no. 10, pp. 56–67, Oct. 1, 2009.

[12] B. K. Harrison, "Computational inefficiencies in sequential modular flowsheeting," *Computers & Chemical Engineering*, An International Journal of Computer Applications in Chemical Engineering, vol. 16, no. 7, pp. 637–639, Jul. 1, 1992.

[13] G.-L. Chen, G.-Z. Sun, Y.-Q. Zhang, and Z.-Y. Mo, "Study on parallel computing," *Journal of Computer Science and Technology*, vol. 21, no. 5, pp. 665–673, 2006.

[14] Z. Xu, Y. He, W. Lin, and L. Zha, "Four styles of parallel and net programming," *Frontiers of Computer Science in China*, vol. 3, no. 3, pp. 290–301, Sep. 2009.

[15] F. Seveso, R. Marichal, E. Dufrechou, and P. Ezzatti, "Refactoring an electric-market simulation software for massively parallel computations," in *High Performance Computing*, Springer, Cham, 2022, pp. 190–204.

[16] A. Vivas and H. Castro, "Quantitative characterization of scientific computing clusters," in *High Performance Computing*, Springer, Cham, 2022, pp. 47–62.

[17] R. P. Goldberg, "Survey of virtual machine research," *Computer*, vol. 7, no. 6, pp. 34–45, Jun. 1974.

[18] A. Whitaker, M. Shaw, and S. D. Gribble, "Denali: Lightweight virtual machines for distributed and networked applications," 2001.

[19] Y. Li, W. Li, and C. Jiang, "A survey of virtual machine system: Current technology and future trends," in *2010 Third International Symposium on Electronic Commerce and Security*, Jul. 2010, pp. 332–336.

[20] P. Sharma, L. Chaufournier, P. Shenoy, and Y. C. Tay, "Containers and virtual machines at scale: A comparative study," in *Proceedings of the 17th International Middleware Conference*, ser. Middleware '16, New York, NY, USA: Association for Computing Machinery, Nov. 28, 2016, pp. 1–13.

[21] I. Pietri and R. Sakellariou, "Mapping virtual machines onto physical machines in cloud computing: A survey," *ACM Computing Surveys*, vol. 49, no. 3, pp. 1–30, Sep. 30, 2017.

[22] Z. Xiao, W. Song, and Q. Chen, "Dynamic resource allocation using virtual machines for cloud computing environment," *IEEE Transactions on Parallel and Distributed Systems*, vol. 24, no. 6, pp. 1107–1117, Jun. 2013.

[23] A. Randal, "The ideal versus the real: Revisiting the history of virtual machines and containers," *ACM Computing Surveys*, vol. 53, no. 1, 5:1–5:31, Feb. 6, 2020.

[24] A. Rosenthal, P. Mork, M. H. Li, J. Stanford, D. Koester, and P. Reynolds, "Cloud computing: A new business paradigm for biomedical information sharing," *Journal of Biomedical Informatics*, vol. 43, no. 2, pp. 342–353, Apr. 1, 2010.

[25] C. Pahl, A. Brogi, J. Soldani, and P. Jamshidi, "Cloud container technologies: A state-of-the-art review," *IEEE Transactions on Cloud Computing*, vol. 7, no. 3, pp. 677–692, Jul. 2019.

[26] I. Ahmad, M. G. AlFailakawi, A. AlMutawa, and L. Alsalman, "Container scheduling techniques: A survey and assessment," *Journal of King Saud University - Computer and Information Sciences*, vol. 34, no. 7, pp. 3934–3947, Jul. 1, 2022.

[27] P.-J. Maenhaut, B. Volckaert, V. Ongenae, and F. De Turck, "Resource management in a containerized cloud: Status and challenges," *Journal of Network and Systems Management*, vol. 28, no. 2, pp. 197–246, Apr. 1, 2020.

[28] R. Vaño, I. Lacalle, P. Sowiński, R. S-Julián, and C. E. Palau, "Cloud-native workload orchestration at the edge: A deployment review and future directions," *Sensors*, vol. 23, no. 4, p. 2215, 2023.

[29] V. A. Merchan, E. Esche, S. Fillinger, G. Tolksdorf, and G. Wozny, "Computer-aided process and plant development. a review of common software tools and methods and comparison against an integrated collaborative approach," *Chemie Ingenieur Technik*, vol. 88, no. 1, pp. 50–69, 2016.

[30] L. T. Biegler, "Recent advances in chemical process optimization," *Chemie Ingenieur Technik*, vol. 86, no. 7, pp. 943–952, 2014.

[31] S. Begum, M. G. Rasul, D. Akbar, and N. Ramzan, "Performance analysis of an integrated fixed bed gasifier model for different biomass feedstocks," *Energies*, vol. 6, no. 12, pp. 6508–6524, Dec. 2013.

[32] Q. Smejkal and M. Šoóš, "Comparison of computer simulation of reactive distillation using aspen plus and hysys software," *Chemical Engineering and Processing: Process Intensification*, vol. 41, no. 5, pp. 413–418, May 1, 2002.

[33] N. M. A. Al-Lagtah, S. Al-Habsi, and S. A. Onaizi, "Optimization and performance improvement of lekhwair natural gas sweetening plant using aspen HYSYS," *Journal of Natural Gas Science and Engineering*, vol. 26, pp. 367–381, Sep. 1, 2015.

[34] R. Gani, "Chemical product design: Challenges and opportunities," *Computers & Chemical Engineering*, vol. 28, no. 12, pp. 2441–2457, Nov. 15, 2004.

[35] T. Ralphs, Y. Shinano, T. Berthold, and T. Koch, "Parallel solvers for mixed integer linear optimization," in *Handbook of Parallel Constraint Reasoning*, 1st ed., Springer Cham, Apr. 6, 2018, pp. 283–336.

[36] V. Kulikov, H. Briesen, R. Grosch, A. Yang, L. von Wedel, and W. Marquardt, "Modular dynamic simulation for integrated particulate processes by means of tool integration," *Chemical Engineering Science*, vol. 60, no. 7, pp. 2069–2083, Apr. 1, 2005.

[37] N. Abdel-Jabbar, B. Carnahan, and C. Kravaris, "A multirate parallel-modular algorithm for dynamic process simulation using distributed memory multicomputers," *Computers & Chemical Engineering*, vol. 23, no. 6, pp. 733–761, Jun. 1, 1999.

[38] E. N. Pistikopoulos, A. Barbosa-Povoa, J. H. Lee, *et al.*, "Process systems engineering – *The generation next?*" *Computers & Chemical Engineering*, vol. 147, p. 107 252, Apr. 1, 2021.

[39] M. Martín and T. A. Adams II, "Challenges and future directions for process and product synthesis and design," *Computers & Chemical Engineering*, vol. 128, pp. 421–436, Sep. 2, 2019.

[40] J. De Tommaso, F. Rossi, N. Moradi, C. Pirola, G. S. Patience, and F. Galli, "Experimental methods in chemical engineering: Process simulation," *The Canadian Journal of Chemical Engineering*, vol. 98, no. 11, pp. 2301–2320, 2020.

[41] S. Cremaschi, "A perspective on process synthesis: Challenges and prospects," *Computers & Chemical Engineering*, Special Issue: Selected papers from the 8th International Symposium on the Foundations of Computer-Aided Process Design (FOCAPD 2014), July 13-17, 2014, Cle Elum, Washington, USA, vol. 81, pp. 130–137, Oct. 4, 2015.

[42] M. A. Duran and I. E. Grossmann, "A mixed-integer nonlinear programming algorithm for process systems synthesis," *AIChE Journal*, vol. 32, no. 4, pp. 592–606, Apr. 1986.

[43] P. Nayak, P. Dalve, R. A. Sai, *et al.*, "Chemical process simulation using OpenModelica," *Industrial & Engineering Chemistry Research*, vol. 58, no. 26, pp. 11 164–11 174, Jul. 3, 2019.

[44] G. Stephanopoulos and G. V. Reklaitis, "Process systems engineering: From solvay to modern bio- and nanotechnology.: A history of development, successes and prospects for the future," *Chemical Engineering Science*, Multiscale Simulation, vol. 66, no. 19, pp. 4272–4306, Oct. 1, 2011.

[45] A. W. Dowling and L. T. Biegler, "A framework for efficient large scale equation-oriented flowsheet optimization," *Computers & Chemical Engineering*, A Tribute to Ignacio E. Grossmann, vol. 72, pp. 3–20, Jan. 2, 2015.

[46] Y. Ma, Z. Shao, X. Chen, and L. T. Biegler, "A parallel function evaluation approach for solution to large-scale equation-oriented models," *Computers & Chemical Engineering*, vol. 93, pp. 309–322, Oct. 4, 2016.

[47] D. D. Nikolic, "DAE tools: Equation-based object-oriented modelling, simulation and optimisation software," *PeerJ Computer Science*, vol. 2, e54–e54, Apr. 6, 2016.

[48] D. Bongartz and A. Mitsos, "Deterministic global flowsheet optimization: Between equation-oriented and sequential-modular methods," *AIChE Journal*, vol. 65, no. 3, pp. 1022–1034, 2019.

[49] R. C. Pattison, A. M. Gupta, and M. Baldea, "Equation-oriented optimization of process flowsheets with dividing-wall columns," *AIChE Journal*, vol. 62, no. 3, pp. 704–716, 2016.

[50] Y. Kang, Y. Luo, and X. Yuan, "Recent progress on equation-oriented optimization of complex chemical processes," *Chinese Journal of Chemical Engineering*, vol. 41, pp. 162–169, Jan. 1, 2022.

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

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

[53] M. Fowler and J. Lewis. "Microservices - a definition of this new architectural term," martinfowler.com. (Mar. 25, 2014), [Online]. Available: `https://martinfowler.com/articles/microservices.html` (visited on 04/09/2024).

[54] P. Jamshidi, C. Pahl, N. C. Mendonça, J. Lewis, and S. Tilkov, "Microservices: The journey so far and challenges ahead," *IEEE Software*, vol. 35, no. 3, pp. 24–35, May 2018.

[55] M. Viggiato, R. Terra, H. Rocha, M. T. Valente, and E. Figueiredo, *Microservices in practice: A survey study*, Aug. 14, 2018.

[56] D. Taibi, V. Lenarduzzi, and C. Pahl, "Processes, motivations, and issues for migrating to microservices architectures: An empirical investigation," *IEEE Cloud Computing*, vol. 4, no. 5, pp. 22–32, Sep. 2017.

[57] D. Gravanis, G. Kakarontzas, and V. Gerogiannis, "You don't need a microservices architecture (yet): Monoliths may do the trick," in *Proceedings of the 2021 European Symposium on Software Engineering*, ser. ESSE '21, New York, NY, USA: Association for Computing Machinery, Mar. 26, 2022, pp. 39–44.

[58] N. Dragoni, I. Lanese, S. T. Larsen, M. Mazzara, R. Mustafin, and L. Safina, "Microservices: How to make your application scale," in *Perspectives of System Informatics*, A. K. Petrenko and A. Voronkov, Eds., vol. 10742, Cham: Springer International Publishing, 2018, pp. 95–104.

[59] N. C. Mendonça, C. Box, C. Manolache, and L. Ryan, "The monolith strikes back: Why istio migrated from microservices to a monolithic architecture," *IEEE Software*, vol. 38, no. 5, pp. 17–22, Sep. 2021.

[60] M. Filho, E. Pimentel, W. Pereira, P. H. M. Maia, and M. I. Cortés, "Self-adaptive microservice-based systems - landscape and research opportunities," in *2021 International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, May 2021, pp. 167–178.

[61] O. Al-Debagy and P. Martinek, "A comparative review of microservices and monolithic architectures," in *2018 IEEE 18th International Symposium on Computational Intelligence and Informatics (CINTI)*, Nov. 2018, pp. 000 149–000 154.

[62] F. Aydemir and F. Başçiftçi, "Building a performance efficient core banking system based on the microservices architecture," *Journal of Grid Computing*, vol. 20, no. 4, p. 37, Nov. 2, 2022.

[63] "Why docker — docker." (Nov. 11, 2021), [Online]. Available: `https://www.docker.com/why-docker/` (visited on 04/29/2024).

[64] V. Saquicela, G. Campoverde, J. Avila, and M. E. Fajardo, "Building microservices for scalability and availability: Step by step, from beginning to end," in *New Perspectives in Software Engineering*, J. Mejia, M. Muñoz, Á. Rocha, and Y. Quiñonez, Eds., vol. 1297, Cham: Springer International Publishing, 2021, pp. 169–184.

[65] M. Waseem, P. Liang, M. Shahin, A. Di Salle, and G. Márquez, "Design, monitoring, and testing of microservices systems: The practitioners' perspective," *Journal of Systems and Software*, vol. 182, p. 111 061, Dec. 1, 2021.

[66] "Kubernetes." (), [Online]. Available: `https://kubernetes.io/` (visited on 04/29/2024).

[67] J. Figueira and C. Coutinho, "Developing self-adaptive microservices," *Procedia Computer Science*, 5th International Conference on Industry 4.0 and Smart Manufacturing (ISM 2023), vol. 232, pp. 264–273, Jan. 1, 2024.

[68] X. Cheng, "Performance, benchmarking and sizing in developing highly scalable enterprise software," in *Performance Evaluation: Metrics, Models and Benchmarks*, S. Kounev, I. Gorton, and K. Sachs, Eds., Berlin, Heidelberg: Springer, 2008, pp. 174–190.

[69] C.-C. Chuang and Y.-C. Tsai, "Performance evaluation and improvement of a cloud-native data analysis system application," in *2021 International Conference on Electronic Communications, Internet of Things and Big Data (ICEIB)*, Dec. 2021, pp. 60–62.

[70] M. Nambiar and H. K. Kalita, "Performance monitoring and analysis of a large online transaction processing system," in *Performance Evaluation: Metrics, Models and Benchmarks*, S. Kounev, I. Gorton, and K. Sachs, Eds., Berlin, Heidelberg: Springer, 2008, pp. 303–313.

[71] N. Mahmoudi and H. Khazaei, "Performance modeling of serverless computing platforms," *IEEE Transactions on Cloud Computing*, vol. 10, no. 4, pp. 2834–2847, Oct. 2022.

[72] M. Jagiełło, M. Rusek, and W. Karwowski, "Performance and resilience to failures of an cloud-based application: Monolithic and microservices-based architectures compared," in *Computer Information Systems and Industrial Management*, K. Saeed, R. Chaki, and V. Janev, Eds., Cham: Springer International Publishing, 2019, pp. 445–456.

[73] M. Woodside, "The relationship of performance models to data," in *Performance Evaluation: Metrics, Models and Benchmarks*, S. Kounev, I. Gorton, and K. Sachs, Eds., Berlin, Heidelberg: Springer, 2008, pp. 9–28.

[74] K. S. Trivedi, B. R. Haverkort, A. Rindos, and V. Mainkar, "Techniques and tools for reliability and performance evaluation: Problems and perspectives," in *Computer Performance Evaluation Modelling Techniques and Tools*, G. Haring and G. Kotsis, Eds., Berlin, Heidelberg: Springer, 1994, pp. 1–24.

Appendices

The project proposal can be found on the next page.

ENGEN582-24X
Honours Research and Development Project

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

Caleb Archer

Supervised by Tim Walmsley, Mark Apperley

Proposal Summary

With new modes of software system organisation and management available to organisations, which help to deliver efficient and scalable outcomes, the process engineering computing space is seeing itself left behind, suffering from both a stunted speed of analysis and a lack of centralised data integration. This project will move the Ahuora Digital Platform onto a distributed Kubernetes cluster hosted on Raspberry Pi nodes, assessing the performance consequences of this move, as well as determine the ideal set of policies with respect to performance outcomes, building on the foundations of existing research. Given the existing access to the necessary compute power, networking components, and free software, the progress of cluster research and development is not dependent on acquisition of further resources. Completion of this project will provide a basis for the execution of more efficient, sustainable, and speedy process simulation and analysis, while taking another step towards industry decarbonisation.

1. Background

Computing and access to computing has changed substantially in the past two decades. Cloud computing is one of the main drivers of this change, and continues to grow rapidly [1]. There has been a shift from software that is run entirely on local devices, paid for via costly perpetual licenses, to software-as-a-service styles of delivery, often made available at a lower, ongoing cost. The complexity involved in managing this new model of software systems, dependent on multiple components (distributed software) to control and deliver the product, has seen tools developed in order to manage this complexity [2]. One such tool is Kubernetes, a container orchestration and management platform built to improve system efficiency and better enable complex software systems to scale according to fluctuating needs. These two properties, efficiency and scalability, provide the core motivation for the Ahuora Digital Platform to deploy on top of Kubernetes.

A 2019 CACHE (Computer Aids for Chemical Engineering) Corporation survey [3] indicated that the proportion of the respondents from the chemical and process engineering industry using numerical analysis software had risen from 26% in 2003 to 38%. Likewise, the proportion of respondents reporting usage of process simulation software had risen from approximately 34% to almost 50%. The two most common process simulation tools were, by a large margin, Aspen Plus and Aspen Hysys.

With the Aspen process simulation tools representing the majority of process simulation software usage amongst chemical and process engineers, it is clear that the industry has a dependency on standalone, closed-source, monolithic software, and it lags in the adoption of modern cloud computing paradigms, tools, and techniques. F. Tapia et al determined that a microservices architecture provides benefits with respect to hardware resource efficiency, costs and productivity over a traditional monolithic software architecture [4]. The chemical and process engineering industry is missing out on these benefits, especially given the compute and time-intensive nature of process simulation and analysis tasks. Monolithic simulation and analysis systems are more sensitive to the fidelity-speed tradeoff, where simulations of higher (desirable) fidelity take longer to complete compared to a result of lower fidelity [5].

The software usage habits of the chemical and process engineering industry also face centralisation issues. By the nature of monolithic, locally deployed software, it is difficult to maintain a centralised and shared view of an entire process plant or site, as information may be available only on one licensed machine, or spread amongst many. With the usage of a distributed, cloud-native architecture, it would be easier to centralise data access and improve the efficiency of information-sharing amongst a process-centred organisation.

The end vision of the Ahuora Digital Platform sees a wide range of process tools being integrated in a centralised fashion, including hybrid pinch, P-graph, deterministic and stochastic optimisation, physics and machine learning-based simulation, data streaming, MPC control and scheduling, and more. This desired integration will be easier to realise by employing Kubernetes as a system management foundation.

2. Overall Aim of the Project

The goal of the project is to construct and test, on top of a physical cluster of Raspberry Pi nodes, a performant, self-adaptive and self-healing distributed system architecture for the Ahuora Digital Platform. This shall involve utilisation of the Kubernetes container orchestration and cluster management platform to deploy, manage, and scale the various components of the platform. To achieve this, load-balancing and scaling policies need to be controlled in such a way that the system meets such performance requirements. Research involving comparative performance analysis of such policies will need to be carried out to inform policy choices.

The finished system will need to efficiently enable process engineering simulations to be carried out, while appropriately managing total consumption of available compute power and memory. It should be possible to automatically deploy and configure the entire Kubernetes cluster. Along with this, the integrity of data stored across a distributed PostgreSQL database must be preserved. This project will ultimately culminate in access to flexible, performant and fault-tolerant analysis of complex processing systems over multiple operating contexts or system parameter configurations.

3. Research and Development Plan

Progress made towards achieving project outcomes will be broken into an interconnected series of stages, some more focused on research, and others development.

The first major research component will involve conducting a literature and technology review in order to assess the previous work and findings of other researchers in the fields of software system performance analysis, distributed system orchestration and configuration, and the cross-section of these.

The existing system components of the Ahuora Digital Platform be migrated towards fully containerised implementations, making them ready for initial deployment to the cluster. These containerised processes, along with cluster deployments, will have baseline policies put in place, which will then be refined further in the research and development process.

Throughout the development process, automated infrastructure management tools, such as Ansible, will be employed to quasi-deterministically create and configure the systems on which the cluster will run, as well as the cluster configuration itself. Such tools are required to reduce the impact of error-prone, manual configuration management, which may also be hard to replicate. This will simplify the process of duplicating the ideal state of the cluster across different physical cluster environments.

Experiment-based comparative analysis of possible load-balancing, scaling, resource allocation and topological arrangement policies shall be conducted to determine the optimal configuration of the Kubernetes cluster. This will be in the form of performance testing, where traffic and request load simulations will be executed, and key metrics shall be measured using a range of monitoring tools, and compared to a baseline measurement across different configuration strategies and policy choices. Experiments will be constructed with well-defined test-cases, which will identify the variable to assess

and modify, as well as variables to keep constant. Similarly structured experiments will also be used to infer any system bottlenecks, allowing decisions to be made on the ideal granularity of containerised processes, especially with respect to compute-heavy analysis components. Any analysis will take into consideration previous work identified in the aforementioned literature review.

Research into the advantages and disadvantages provided by centralised versus distributed software systems shall be conducted. This shall involve direct comparisons of performance achieved between distributed and local versions of the same software system.

There are elements of the research process that create a level of uncertainty, which may delay or otherwise hinder project progress. For example, the system bottleneck analysis assessing containerisation granularity could result in slow progress, depending on what stage of development each process analysis component is in. If it is determined that two components could be containerised together without any performance impacts, there may be a large delay in being able to action this decision, since this would depend on other engineers within Ahuora to make the necessary changes, which may or may not be substantial. Another source of uncertainty is the accuracy of the performance measurement tools used, and the impact they may have on results. Along with this, even with confidence in measurement accuracy, results may be skewed regardless due to the performance overheads from conducting such measurements.

4. Resources

As a software-focused project, there is no substantial dependency on many different physical resources in order to conduct the research and development process of the project.

There is already access to eight Raspberry Pi 5 devices, on which the Kubernetes cluster will be deployed and configured. Along with this, there is a 4G router for providing internet access to the cluster (independent of the primary university network), an eight-port network switch, ethernet cables, and a rack for storing the cluster nodes. Necessary IDEs (Integrated Development Environments) and code editors such as IntelliJ PyCharm and Visual Studio Code are already installed on a dedicated office workstation. Any unforeseen need for particular software is likely to be met by additional open-source software.

One of the more probable risks that could impact project progress is failure of one or more of the Raspberry Pi nodes that shall make up the cluster. Although there could be performance impacts due to a lower node count, cluster operation and development could easily continue, as one of the primary features provided by Kubernetes is resilience to node failure.

A greater problem that could present itself would be failure of a control node (a node partially or fully responsible for managing the rest of the cluster), especially if there is only one control node. Failure of a lone control node could paralyze the entire cluster. Mitigations include: usage of the configuration automation tool Ansible (whereby the cluster set-up process is defined in code), designating at least three control nodes to the cluster for high availability, and regular back-ups of critical cluster data.

In general, replacing a failed Raspberry Pi would not be at great cost, as a single Raspberry Pi 5 costs approximately $160 NZD, including a new active cooler.

5. Sustainability and Vision Mātauranga

Through the usage of a performance and resource-optimised Kubernetes cluster designed for enabling efficient and effective process simulation and analysis within process engineering contexts, sustainable

computational resource usage is greatly enhanced. Energy already provided to computers on which process software is run can be driven to high efficiency and utilisation, and consequently, process engineers can gain access to the results they need sooner, and therefore make decisions to optimise a particular process faster.

Though this project does not directly invoke integration of Vision Mātauranga principles, it does contribute to the wider Ahuora Digital Platform aspiration of promoting kaitiatikanga (guardianship) and responsible stewardship of the environment, through providing the process engineering industry the digital tools they need to decarbonise and make more efficient their operations.

References

[1] Mordor Intelligence. Cloud computing market - size, growth, report & analysis, n.d. https://www.mordorintelligence.com/industry-reports/cloud-computing-market.

[2] Kubernetes. Overview, n.d. https://kubernetes.io/docs/concepts/overview/.

[3] Robert P. Hesketh, Martha Grover, and David L. Silverstein P.E. Cache/asee survey on computing in chemical engineering. In *2020 ASEE Virtual Annual Conference Content Access*, number 10.18260/1-2–34249, Virtual On line, June 2020. ASEE Conferences. https://peer.asee.org/34249.

[4] Freddy Tapia, Miguel Ángel Mora, Walter Fuertes, Hernán Aules, Edwin Flores, and Theofilos Toulkeridis. From monolithic systems to microservices: A comparative study of performance. *Applied Sciences*, 10(17), 2020. https://www.mdpi.com/2076-3417/10/17/5797.

[5] sJie Xu, sSi Zhang, sEdward Huang, sChun-Hung Chen, sLoo Hay Lee, and sNurcin Celik. Efficient multi-fidelity simulation optimization. In *Proceedings of the Winter Simulation Conference 2014*, pages 3940–3951, 2014.

Appendices

A. Gantt Chart