UNIVERSITY OF TARTU
Faculty of Science and Technology
Institute of Computer Science
Software Engineering Curriculum

Ihar Suvorau

Scaling Out the Discovery of Business Process Simulation Models from Event Logs

Master's Thesis (30 ECTS)

Supervisors: Marlon Dumas, Prof.
David Chapela de la Campa, PhD

Tartu 2023

Scaling Out the Discovery of Business Process Simulation Models from Event Logs

Abstract:
Background. The automated discovery of business process simulation (BPS) models has received considerable attention in the process mining community in the past decade. The main open question in this field is how to make such discovery accurate, fast and efficient to provide more value for the end-users.
Aim. This thesis aims at re-architecting an existing tool for automated BPS model discovery, namely Simod, to manage varying workloads in a scalable and robust manner.
Methods. Scalability and robustness are achieved through building a distributed event-based system using the integration with the Kubernetes API. An efficiency metric has been used to evaluate the scalability of the final solution. A robustness-under-load experiment shows that the re-architected system remains available under high demand.
Results. The results of the validation experiments showed the system is scalable for small-sized event logs and robust under high load. A limitation of the study is that the testing environment, based on *kind*-clusters of 1, 2, 3, and 4 worker nodes, is not suitable for large-scale load testing experiments.
Conclusion. This thesis provides a framework for implementing scalable, robust, and resilient workflows on Kubernetes for BPS model discovery that can benefit the process mining community. Further work is needed to improve the Simod architecture by splitting it into smaller independent components to achieve higher scalability and resource utilisation.

Keywords:
Process mining, process discovery, process simulation, horizontal scaling, Kubernetes, cloud architecture

CERCS: P170—Computer Science, Numerical Analysis, Systems, Control

Äriprotsesside simulatsioonimudelite avastamine sündmuste logidest

Lühikokkuvõte:
Taust. Automatiseeritud äriprotsesside simulatsiooni (BPS) mudeli avastamine on protsesside kaevandamise kogukonnas oluline saavutus. Peamised küsimused on nüüd seotud selle avastamise täpseks, kiireks ja tõhusaks muutmisega, et lõppkasutajatele rohkem väärtust pakkuda.
Eesmärk. See lõputöö lihtsustab juurdepääsu Simodile, pakkudes HTTP API-d ja võimaldab skaleeritavat BPS-i mudelite avastamist, mis haldavad tippnõudlust tõhusalt.
Meetodid. Skaleeritavus ja töökindlus on saavutatud hajutatud sündmustepõhise süsteemi loomisega, kasutades integratsiooni Kubernetes API-ga. Tõhususe mõõdikut on kasutatud lõpplahenduse skaleeritavuse kinnitamiseks; ja koormuse all töökindluse katse on näidanud, et süsteem suudab suure nõudluse korral kättesaadavaks jääda.
Tulemused. Valideerimiskatsete tulemused näitasid, et süsteem on skaleeritav väikese suurusega sündmuste logide jaoks ja vastupidav suure koormuse korral. Kuid 1, 2, 3 ja 4 töötaja sõlmedest koosnevaid *kind*-klastreid kasutanud testimiskeskkonna eripärad ei sobi suuremahuliste koormustestimise katsete jaoks.
Järeldused. See lõputöö pakub raamistikku skaleeritavate, töökindlate ja vastupidavate töövoogude käitamiseks Kubernetes BPS-i avastamiseks, mis võib olla kasulik protsesside kaevandamise kogukonnale. Täiendavat tööd on vaja Simodi arhitektuuri ülevaatamiseks ja väiksemateks sõltumatuteks komponentideks jagamiseks, et saavutada suurem mastaapsus ja ressursside kasutamine.

Võtmesõnad:
Protsesside kaevandamine, protsesside avastamine, protsesside simuleerimine, horisontaalne skaleerimine, Kubernetes, pilvearhitektuur

CERCS: P170—Arvutiteadus, numbriline analüüs, süsteemid, juhtimine

Contents

List of Figures

List of Tables

Introduction

Business process simulation (BPS) is a technique for analysing the efficiency of business processes. A business process is a series of events in a particular order executed by resources to achieve a business goal. Simulating such sequences of events under various configurations allows us to answer *what-if* questions about the process: "What if an order of activities is changed?", "What happens if more resources are assigned to the front-end activity?", "Would the new adjustment create a bottleneck downstream of the process?". Besides, recording various process performance measures during this simulation (e.g., processing and waiting times, resource utilisation, case frequency) is used for quantitative analysis of the process.

A Business Process Simulation model (henceforth, a *BPS model*) captures the behaviour of a business process and acts as a starting point in the simulation. A BPS model is constructed from a process model enriched with additional simulation parameters, such as resource working hours, case arrival rates, branching probabilities, and resource costs. Business analysts use BPS models for process improvement, performance analysis, and analysis of compliance with industry standards.

To a great extent, the simulation's quality depends on the input to a simulator, i.e., the process model and simulation parameters. Traditionally, BPS models are composed by domain experts using manual data collection techniques, such as employee interviews and on-site observations [CDG20]. This approach is time-consuming and error-prone. Automated BPS model discovery from event logs improves the discovery by removing the manual component [CDG20]. It also enables organisations to utilise their data collected in organisation management software[1] for business process improvement.

BPS model discovery is more complex than traditional business process model discovery, which focuses on the control-flow perspective. The discovery of simulation parameters can be complicated and tedious. Additionally, large enterprises would generate gigabytes of log data that BPS model discovery algorithms should handle, while the number of users in such enterprises can be in the order of dozens or hundreds of concurrent users. Thus, scalable discovery of BPS models is essential if it is to be adopted by the industry. Besides large event logs, the discovery solution facing the internet can meet unpredictable demand, leading to peak load hours and long hours of inactivity.

Simod is a tool for automated discovery of accurate BPS models from event logs developed by Camargo et al. [CDG20]. It was originally developed as a research prototype with a command-line interface (CLI) and dependencies in two

[1]Such management software as supply chain management systems, enterprise resource planning, or customer relationship management.

programming languages. The barrier for installing and using Simod v2.3.1[2] was quite high. Also, the tool was not designed to handle large workloads or handle simultaneous discovery requests, thus, limiting the application of Simod.

This thesis aims at simplifying access to automated BPS model discovery using Simod, by providing a web service with an HTTP API and allowing scalable discovery of BPS models that can handle multiple simultaneous users, sustain efficiency at scale, and remain available during high demand. This will significantly simplify the usage of the tool by business analysts and give access to a wider community. End-users will benefit from removing the need to set up development environments for Python and Java in order to install Simod and its dependencies. As a side goal, this work aims to improve the development experience and prepare the code base for future changes. The main requirements of the new solution for scalable BPS model discovery include:

- the access to the tool through the HTTP API with the ability to submit discovery tasks by uploading an event log in the XES format and the Simod configuration in YAML;
- the ability to query the discovery job status and, when the job is finished, to retrieve the discovery results;
- serving multiple users simultaneously by utilising several machines and automatically instantiating Simod processing;
- storing of the user request-related data persistently.

The thesis author's contributions are summarised as the following:

1. An implementation of the distributed solution for scalable discovery of BPS models using Simod that exposes a web service with the Application Programming Interface (API) based on the Representational State Transfer (REST) architectural pattern.
2. Simod refactoring to enhance its code readability, maintainability, reduce coupling between internal modules, and improve packaging and distribution using an automated continuous integration (CI) pipeline and Docker.
3. An approach to scale out Simod (or other long-running task that can be packaged into a Docker container) to handle a large number of concurrent users employing the Kubernetes container orchestration technology.

The author thanks *Dr. Camargo* for the initial introduction into Simod; *Prof. Dumas* and *Dr. Chapela* for their contributions to the Simod's refactoring and guidance throughout the work on the thesis, their domain knowledge and consultations in process mining were invaluable in this research.

[2]https://github.com/AutomatedProcessImprovement/Simod/tree/v2.3.1

The rest of the thesis is structured as follows. Section 1 describes the context and several simulation concepts necessary to understand the goal and proposals of the thesis. This section also introduces business process discovery concepts and technologies for developing scalable software solutions. Section 2 introduces related work concerning process discovery and development of distributed and scalable software systems. Section 3 presents the approach to the problem, describes in detail the architecture of the solution for scalable BPS model discovery, and other methods and techniques applied during the work. Section 4 provides the validation framework for testing that the main goal of building a scalable solution has been achieved, the section also includes the results of the load testing experiments. Section 5 discusses the results of the validation tests, concludes if the goal has been achieved, and proposes further improvements to the system.

1 Background

This chapter introduces the background for the thesis by going deeper into three main contexts: BPS model discovery, Simod, and scaling of distributed systems. Section 1.1 describes the basic concepts and approaches in process mining to model discovery and their use in performance analyses. Section 1.2 establishes the starting point of the thesis. In the end, Section 1.3 develops a general approach to building highly scalable applications for modern deployment environments.

1.1 Business Process Simulation Model Discovery

Typically, BPS model discovery and other process mining techniques take an *event log* as an initial data source. An event log represents a series of events with attributes that provide required and additional information for process mining tools, and can come in various formats. The minimum required attributes are the timestamp denoting when the event occurred, the name of the executed activity (e.g., Check stock availability, Confirm order, Reject order), the name of the resource who performed the activity, and an identifier of the process case (i.e., the instance of the process this event belongs to), which usually either a customer reference number or other unique identifiers that group series of activities in a case. The series of such cases represent a single process. A process can be of a different kind, in business process management, it is often an accounting process, invoicing, recruitment, customer onboarding, product development and others. Table 1 shows an example with eight events from an event log of a purchasing process. In this example, the events store two timestamps, one denoting the executed activity's start and another denoting its end.

In BPS, the backbone of the process is the *control-flow*. It represents the structure of the process and is composed of *i)* a process model, a diagrammatic representation of the activities of the process and the transitions between them; and *ii)* the probabilities associated with each transition, modelling the probability of the flow to go through each path (e.g., either cancel or accept an order after assessing its validity).[3] The most frequent modelling languages used to represent a process model are *Petri nets* and *Business Process Model and Notation* (BPMN). Petri nets is a mathematical and graphical modelling technique that models and simulates dynamic, concurrent, non-deterministic information systems using a directed acyclic graph (DAG) [Mur89]. On the other hand, BPMN is a notation language designed explicitly for modelling business processes and workflows [Obj14], which provides a rich set of modelling elements (e.g., process, task, sub-process, gateway, event,

[3]Different approaches can be used to model the path flow, e.g., rules based on data associated to the case being processed. The tools used in the scope of this thesis work with a stochastic modelling of the control-flow.

Case ID	Activity	Start Time	End Time	Resource
		⋮		
512	Register order	03/11/2021 08:00:00	03/11/2021 08:31:11	Alice
512	Check stock availability	03/11/2021 08:31:11	03/11/2021 08:58:09	Bob
513	Register order	03/11/2021 08:31:11	03/11/2021 09:02:51	Bob
512	Confirm order	03/11/2021 09:00:00	03/11/2021 09:17:01	Alice
513	Check stock availability	03/11/2021 09:02:51	03/11/2021 09:35:50	Bob
513	Reject order	03/11/2021 09:17:01	03/11/2021 09:46:12	Bob
512	Notify customer	03/11/2021 15:17:01	03/11/2021 15:27:45	Alice
513	Notify customer	03/11/2021 15:46:12	03/11/2021 15:57:43	Alice
		⋮		

Table 1. Example of a fraction of an event log with eight events storing the case identifier, the executed activity, the start and end timestamps, and the resource.

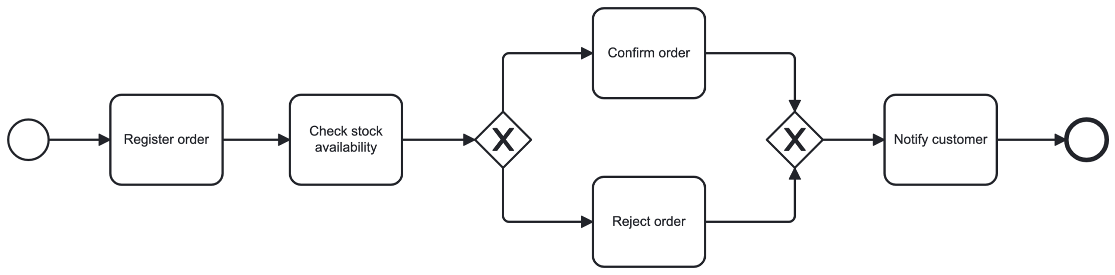


Figure 1. BPMN model example formed by five activities and two XOR gateways corresponding the event log of Table 1.

message, and others). Figure 1 depicts an example of a process model in the BPMN notation, corresponding to the event log in Table 1.

Process model discovery algorithms have to deal with noise and complexity. Event logs may contain incorrect data, poorly formatted values, incomplete cases, missing activities, unspecified resources, and other irregularities. Discovery algorithms must be able to identify the issues, remove the noise, fill in the incomplete data, and compose a sound and correct model that can represent reality in a general and valuable way. Models, discovered with automated algorithms, can be assessed by the four quality criteria: *fitness*, *simplicity* (complexity), *precision*, and *generalisation* [Van16]. Fitness measures how much behaviour observed in the event log can be reproduced by the process model. Complexity defines the ability to understand the model. Precision and generalisation help measure how balanced the resulting model is to avoid under-fitting and over-fitting.

Besides the control-flow, *temporal* data extracted from the event log can improve the BPS model by providing additional information that can be used in BPS to mimic the process in a more realistic way. Resources in the event log are usually either particular employees, roles of the employees, or automated systems. If automated systems can work around-the-clock, employees usually have a predetermined schedule which can be recovered from the event log even if such a schedule has not been provided explicitly. Timetable mining can significantly increase the accuracy of the model and subsequent simulations by providing the working hours of the resources. This, in its turn, allows further time analysis, more accurate performance measurement, and waste identification.[4]

Regarding the techniques employed in this thesis, the temporal perspective of the process is modelled by two elements: the arrival model and the resource allocation model. The first one models the arrival of new cases (i.e., executions of the process) to the system by defining their frequency and periods of the day in which they are created. The second one models the set of resources available to perform the activities of the process, which activities they can perform and how long they take, as well as the working schedules they follow.

BPS model discovery techniques aim to build a BPS model, with all its components, from the event log of a process. Subsequently, the discovered BPS model can be used by a simulation engine to generate a set of cases resembling the behaviour of the modelled process. These simulated logs are useful for analysing the performance of the process under the scenario described by the BPS model. The discovery of the BPS model of a process enables modification of scenario characteristics, e.g., changing the resource schedules and analysis of such modifications' impact on the performance of the process.

1.2 Simod

The current work starts with *Simod v2.3.1*, a tool for the automated discovery of BPS models from event logs, initially developed by Camargo et al. [CDG20]; [Cam21] as part of the Process Improvement Explorer (PIX) project funded by the European Commission[5]. Simod v2.3.1 discovered accuracy-optimised process models by introducing an optimisation feedback loop into its architecture with additional model enhancement techniques.

Figure 2 illustrates the approach employed by Simod. The control-flow discovery was implemented using *Split Miner* [Aug+19], the model discovery technique controlled by two parameters, parallelism and frequency thresholds. The parallelism threshold ϵ adjusts concurrent relations between captured events, and the frequency threshold η is used to retain the best incoming and outgoing edges above the threshold value [CDG20]. These two parameters were subjects to optimisation in

[4]Waste refers to time not spent on the execution of the activity assigned to the resource. Waiting time is one example of such process waste that can be analysed further down [Las+22].

[5]https://cordis.europa.eu/project/id/834141

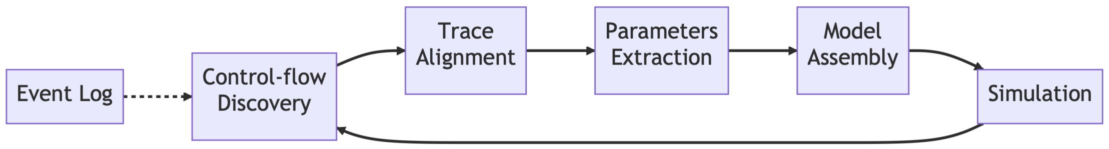


Figure 2. Simod BPS model discovery approach [CDG20]

Simod in order to discover process models that capture the input event log with the best accuracy.

Trace alignment was necessary for computing branching probabilities which are essential simulations parameters. To compute the probabilities, each trace has to be replayed to count how many times each conditional flow in a process model was reached. Trace alignment algorithms do such replaying of an event log against a process model and handle possible misalignments. The misalignments occur because of imprecisions in discovered process models. In Figure 1, for example, the process model assumes each case follows the sequence *Register order—Check stock availability—Confirm order OR Reject order—Notify customer*. If the event log has a different sequence, *Register order—Confirm order—Notify customer*, the trace alignment algorithm must be able to figure out that the *Check stock availability* activity has been skipped, so that branching probabilities can be computed correctly.

Simod used *ProConformance 2.0* [Rei+17], a technique and corresponding tool for computing misalignments between an event log and process model. The misalignments were resolved either by removing, replacing or repairing non-conformant log traces [CDG20]. It is worth noting that traversal of each gateway in the event log is a computationally intensive task which can introduce significant overhead during the BPS model discovery.

The parameters' extraction step used different process mining techniques to mine simulation parameters from the log. For timetable mining (calendar discovery), Simod used a technique that accounted for resource availability constraints [Est+21]. For example, Simod may discover that one resource pool performs tasks *Register order* and *Check stock availability* while the other one performs *Confirm order*, *Reject order*, and *Notify customer* from Table 1. Calendar discovery was implemented using the tool by Mahdy I.[6]. The tool provided resource pools with specific week days and times. Using the previous example, the tool may return two resource groups. *Role 1* where resources work on Monday-Friday 09:00-18:00, while *Role 2* worked from Wednesday to Friday between 14:00 and 18:00. Gateway branching probabilities and their distributions, inter-arrival times were among other simulation parameters mined by Simod during the parameter's extraction.

[6]https://bitbucket.org/Ibrahim_Mahdy/calendar/src/master/

Assembly is the final stage of the BPS model discovery in Simod, during which a BPMN model was constructed and written to disk as an XML file. The *QBP/BIMP* simulator[7] was used for simulation. A simulation generates a synthetic event log (also known as a simulated log). In order to assess the accuracy of the simulation, the simulated log is compared with the original log to assess their similarity. The higher the similarity, the more evidence there is that the BPS model is able to accurately replicate the observed reality. The similarity assessment used the Damerau-Levenshtein (DL) distance metric, which measures the edit distance in two strings. After calculating the distance metric, a decision was made whether to start another loop of the control-flow discovery, trace alignment, and parameters extractions or finish the discovery process and return the final model. The DL distance metric has to be computed for every pair consisting of a trace from the simulated log and from the original log. This pair-wise computation also leads to significant computation overhead in the BPS model discovery.

Even though the above approach to BPS model discovery used in Simod provided high-quality and simple process models, the software had its issues and drawbacks. The execution time of Simod varied greatly depending on the event log size—the optimisation loop forced to rerun the entire discovery pipeline many times, which slowed down the processing. For example, an event log in XES format of 180 MB in size, if run with 60 optimisation cycles on a machine with 20 CPU cores and 40 GB of RAM, took around two hours to process. Further, the installation and use of Simod required deep knowledge of how the tool operates, which hindered the accessibility of the tool for the research community.

Besides these limitations, underlying tooling, such as Split Miner, ProConformance, and QBP/BIMP, were written in Java, while Simod was a Python application. The reliance on two different development environments, Java and Python, introduced additional difficulties for end-users, potential contributors, and developers of Simod. Furthermore, two components that were under development in the PIX project had to be integrated into Simod to improve the quality of the discovered BPS models: *i) Prosimos* [LHD23], a new simulation engine supporting a more complex resource allocation model (replacing QBP/BIMP); and *ii)* an approach to discover a component to model extraneous activity delays [CD22]. These were the main reasons behind the decision to revise the architecture and implementation of the discovery stages of Simod, which led to the subsequent refactoring and scaling of Simod (the purpose of this thesis).

[7]https://www.qbp-simulator.com

1.3 Scaling Network-Based Distributed Systems

Providing public access to an application brings many challenges to an organisation. Modern software solutions often depend on many other applications and libraries, which sometimes have conflicting dependencies. Software portability and cross-platform distribution bring additional overhead and make continuous integration and delivery (CI/CD) solutions almost a requirement. Organising a software development process to make multiple teams work together on a single product effectively is a challenge on its own. Unpredictable demand forces system architects to design scalable and flexible systems that can automatically manage their operational costs. This work focuses on the scalability challenge, so the following discussion will delve into various strategies and approaches to ensure the adaptability of an organisation's software infrastructure.

When designing a system, it is important to consider a *deployment environment* since the beginning, because this environment will dictate most of the properties of the final solution. What are the target operating systems and platforms? What kind of hardware is available? How is data stored and distributed? What is the approximate scale of workload? Does the system require a single machine or a cluster? Is it a multi-location system, and data has to be stored and processed in multiple countries to comply with the legislation? Because of the many questions, a handful of deployment environments emerged to this day.

On-premises environment is a traditional approach when the software solution is installed and operated on the hardware available in-house by the organisation itself. Running in *data centres* is a similar approach, which assumes a more extensive hardware and software infrastructure, managed either by the organisation or by a managed service provider. Private data centres may provide more control and privacy with the cost of managing the operational overhead of running a data centre. *Cloud computing,* or infrastructure as a service (IaaS), is an emerging solution that provides on-demand computing resources. Clouds remove the need for the organisation to maintain the hardware and provide API-based access to computing, storing, networking and security. It is the most versatile way of deploying an application because of the hundreds of services cloud providers make available, and it is changing how software is designed, built, deployed and operated.

A *cloud architecture* leverages cloud capabilities to build scalable, flexible, and efficient applications. Because computing resources and storage are now available with a network request, cloud architecture enables solutions that can leverage on-demand resources, and which encourages isolation, independent deployment and scaling of system components. That is why *microservices* is one of the most widely adopted patterns for cloud-native applications [Lin16]. Microservices are the evolution of the service-oriented architecture (SOA) for cloud settings. The most important distinction is greater isolation and independent deployment of system components. Microservices pattern is also an organisational tool which allows arranging development teams around the software product in a more manageable way, with stricter boundaries. Because of the increased isolation and loose coupling of the components, the emphasis is on communication between services rather than state and data sharing. JSON over HTTP is the most widespread way to implement messaging between cloud-based applications [Sil16].

Containerisation is the enabling technology for microservices and cloud environments. Before containers, virtual machines (VMs) played a similar role and helped to isolate different development and production environments for teams. Containers improve performance, providing much less overhead than VMs, and allowing for more efficient resource utilisation. They also have faster start-up times, which makes the automated scaling of software containers more rapid. Containerisation also solves the *dependency hell* problem by isolating applications with their dependencies and runtimes into containers [Cas19]. Finally, the application portability and distribution are greatly improved by containerisation technology, which provides better development and distribution experience. The Open Container Initiative[8] (OCI) develops the container image, runtime, and distribution specifications, removing the reliance on vendors and improving long-term support and adoption of the technology.

However, microservices and containerisation also have downsides; the biggest is the cloud-native applications' complexity. Microservices significantly increase the number of moving parts in a system, relying heavily on communication and networking, which is known to be unreliable. Thus, a more robust and fault-tolerable approach is necessary to build microservice components. Testing becomes harder, requiring stubs, fakes and mocks of other components to run the component in a testing environment during the development. Orchestrating deployed microservices and ensuring fine-grained control and observability is a big challenge. Container orchestration solutions, such as Kubernetes, help to tackle this issue.

Container orchestration provides a platform for managing the whole cycle of container-based applications, starting from their deployment. Resource control, scheduling, load-balancing, fault-tolerance, self-healing, and autoscaling are the main features of orchestrators [Cas19]. This also gives software development teams a way to run the software themselves (NoOps) without resorting to a development operations team (DevOps). Kubernetes is one of such container orchestrators initially developed by Google based on its predecessor Borg [Ver+15].

Kubernetes is the platform that pioneered the application-oriented infrastructure model for data centers instead of being machine-oriented [Bur+16]. The application-oriented infrastructure abstracts away hardware details from developers and simplifies application deployment and management. Kubernetes offers

[8]https://opencontainers.org

a declarative job specification language to describe workloads, where workload is an application running in a cluster using one or more pods. Pod is a single instance of a container with additional configuration parameters related either to the container itself or to Kubernetes. Moreover, Kubernetes provides an API for applications to leverage the power of automated scaling and launching workloads in a programmed and dynamic manner. Docker Swarm and Red Hat's OpenShift are among the alternative solutions for container orchestration. However, Kubernetes is probably the most feature-rich platform with support from major cloud providers which eliminates the need to install and manage Kubernetes clusters by the user.

2 Related Work

The scaling of BPS model discovery can be achieved in several ways. One of those is **reimplementation of the discovery algorithm** to work on large-scale distributed data so that the discovery job can be run in a compute cluster, which would imply to reimplement the discovery of all components (e.g., control-flow). Focusing solely on the control-flow discovery, Evermann presented an approach applying the *Map-Reduce* framework using Apache Hadoop to a distributed event log by reimplementing two well known process model discovery algorithms (the Alpha algorithm and the Flexible Heuristics Miner) [Eve16]. However, this approach requires deep knowledge of process model discovery algorithms, and not all of them can be efficiently parallelised. The Map-Reduce model requires the implementation to convert all the processing to mapping and reducing operations, which can lead to complex and unmaintainable code. The overhead of setting up Map-Reduce tasks and distributing data across the cluster causes network I/O contention and introduces additional latency, which might not be suitable for low-latency use cases [WKK15]. Unbalanced, non-uniformly distributed data (the data skew) causes some Hadoop workers to have a disproportionate amount of work, leading to worse performance [WKK15].

Apache Spark addresses some of the limitations of the Hadoop Map-Reduce model that made it a widely used framework for distributed big data cloud computing. Cheng et al. presented a scalable discovery of hybrid process models (f-HMD) using Spark in cloud settings [CDA20]. As a result, the Spark framework proved to be an excellent choice for building efficient and scalable process model discovery solutions for large event logs. However, if one wants to apply such an approach to small- and mid-size event logs, there is a high overhead cost. Spark requires at least 8 GB of memory and at least 8–16 CPU cores per machine[9], which makes it less reasonable to use such discovery techniques for not so big data. Furthermore, all the above-mentioned limitations would also hold for the parallelisation of the rest of the BPS model discovery components (e.g., resource profiles). The purpose of this approach is to discover BPS models over large datasets, which cannot be processed in a single computing node.

Another approach to BPS model discovery, instead of big data processing frameworks, is to use **workflow engines** and **job orchestrators** to handle discovery of many BPS models in parallel. Workflow engines provide more flexible data processing than big data frameworks. Not all steps in BPS model discovery are scalable and can run in parallel. In addition, BPS model discovery is a more complex and composite process than process model discovery, which may require multiple process mining tools. Simod, for example, used Split Miner for process model

[9]https://spark.apache.org/docs/3.4.0/hardware-provisioning.html

discovery, ProConformance for conformance checking, a custom-built calendar discovery application for resource pooling, and QBP/BIMP simulator for BPS model simulation. Such a workflow would be difficult to build with big data frameworks. Workflow engines and job orchestrators seem like a better fit for a composite application like this.

Apache Airflow, *Luigi*, *Dagster* belong to a new generation of data processing workflow engines with scheduling and distributed computing capabilities. Initially built for extract-transform-load (ETL) data pipelines, they are flexible and versatile frameworks that may integrate with big data frameworks like the one mentioned above. Apache Airflow, for example, uses Python to describe its data ingestion, processing, and extraction tasks. The tasks are assembled into a directed acyclic graph which can be scheduled and executed on a cluster of workers. Data transformation is usually done with operators, such as PythonOperator, BashOperator, EmailOperator, and others, splitting a workflow into small and manageable pieces of work. One more benefit of using such workflow engines is that they usually come with a graphical user interface (UI) to monitor the execution process, because some jobs may take a long time to finish. No literature is available on applying such workflow engines to BPS model discovery tasks.

Regarding highly scalable workloads, the cloud is often the default environment for such deployments. Not all workflow engines are built to run in cloud settings. Even though Apache Airflow currently has an integration with Docker, and can run containerised applications, it was not designed for this from the beginning, and the development experience can still be improved. Besides, Airflow is Python-centric; operators in other languages can be used with the help of BashOperator, but the rest of the framework is in Python. Thus, Apache Airflow might not be the best choice if a data processing pipeline has many tools that use different technologies. Here is where **cloud-native workflow engines** turn out to be useful. *Argo Workflows*[10] is a workflow engine for Kubernetes that supports data processing, machine learning, ETL, and other use cases. Native integration with Kubernetes makes the platform autoscalable, resilient, and technology-agnostic, because a task in a workflow is just a containerised application which can be implemented using any technology.

Although Argo Worfklows is already a cloud-native solution for orchestrating data processing or other workflows, there is an even more fundamental way of achieving a similar outcome. *Kubernetes* provides different kinds of workloads[11], like Jobs and CronJobs. Job is a basic workload type that provides a one-time execution of a single task. As a Kubernetes workload, a job inherits such features as scaling, resilience, and automated clean-up of finished jobs. If a job has to

[10]https://github.com/argoproj/argo-workflows/

[11]https://kubernetes.io/docs/concepts/workloads/controllers/

be run on schedule, CronJob is used with the schedule syntax similar to *cron*, a command-line utility on Unix-like operating systems. Because it is not possible to know a schedule of BPS model discoveries submitted by users, CronJob is not considered and the Job workload type is used to implement scalable BPS model discovery in this thesis.

3 Approach

This chapter describes the proposed approach for developing a system for scalable discovery of BPS models using Simod at its core. Having only Simod is not enough to serve end-users. Public interface, task management, resource allocation, persistent data storage, caching, and monitoring are all essential components of a well-engineered system ready to serve end-users reliably.

Depending on the input event log and optimisation parameters, BPS model discovery can take a significant amount of computing resources. It would be impossible to design such a system as a desktop application, as end-users might not have enough computing capacity for a job, or desire to run long-running tasks on their machines. Thus, remote execution is the most appropriate method for such jobs that removes the burden from the end-user and places it on the organisation. This allows the organisation to utilise scheduling and resource allocation more efficiently, and run the operations cost-effectively. A web service would allow the control of the remote execution on behalf of the user.

The main focus of this thesis is to extend Simod to handle unpredictable demand in BPS model discovery. One of the challenges is to scale each system component differently, because they might be subject to different loads with varying load profiles. Besides running at scale, there is a need to remain responsive and available under peak loads, while having some response time degradation is acceptable.

Another concern is to refactor Simod to increase its maintainability and prepare it for future change. Software maintenance and support are critical activities in the software development lifecycle [BF14, ch. 5] that take the most time during the software lifetime. Having self-contained loosely coupled system components simplifies software testing, modification, independent deployment, and operation.

In this project, safety and security are not primary concerns, because the service is not intended for use in critical, life-threatening situations. The input data is not considered private at this stage of development. Furthermore, the service is expected to be integrated into a larger software suite with an API gateway, making the BPS model discovery system an internal component without direct exposure to the internet. Thus, the security aspect is neglected in this study.

The following sections describe fundamental ideas on how to build a scalable system (Section 3.1), present the work done on refactoring Simod to prepare it for the primary goal of this thesis (Section 3.2), introduce the system with its components at different levels of detail (Section 3.3), describe an approach to scaling these services (Section 3.4 and Section 3.5), and present the use of language models and other intelligent assistants in this work (Section 3.6).

3.1 Design Principles for Scalability

Scalability is the capacity of a system to maintain its *productivity* as load increases [Dai12, p. 289]. Productivity, in its turn, is defined by business goals and can usually be described with performance metrics like response time and throughput.[12]

One of the design principles for scalability is the *distribution* of components across several computing nodes. It is rarely so that the system load is spread evenly across its subsystems. Usually, some components become bottlenecks and hinder the whole system's performance. Splitting such a system into loosely coupled parts that communicate with each other through message passing makes it possible to address each component individually by increasing the resource capacity (vertical scaling) or the number of instances (horizontal scaling).

Modular software is easier to split into standalone components by its boundaries. Therefore, designing for loose coupling improves maintainability and increases the potential for scalability. *Microservice* architecture pattern also encourages such loose coupling during the deployment stage. Other architectures do not require the independent deployment of components, but the microservice style emphasises it. Independent deployment is enabled by CI/CD pipelines configured for each component separately.

Containerisation of services and applications enables scalability by simplifying software packaging and delivery. It removes the dependency management problem by utilising Linux kernel features such as *namespaces* and *cgroups*, which allow for application runtimes isolation. However, containerisation technology increases the dependency on the internet with the introduction of image registries that store and distribute images. It also increases the load on the network during new components' version rollout.

Caching and *memoisation* are universal techniques that reduce the load on the system by avoiding redundant computation for similar requests, leading to a speed-up. Caching also reduces the load on the internal network by reducing the amount of data passing through the network from service to service. Besides that, caching is vital in containerisation technologies for the same reasons. It may take a long time to build an image of a service because of many dependencies to compile and additional packages to install on the OS level. Image build caching speeds up the whole containerisation process and reduces the network load.

Load balancing improves scalability by distributing the load across multiple instances of the same component. It is an essential technique that enables the horizontal scalability of services to handle peak-hour loads. The main role of a load balancer is to relay requests or tasks to downstream services employing

[12]The efficiency metric used to measure the scalability is described more precisely in the *Validation and Results* chapter (Section 4).

various scheduling strategies, e.g., round-robin, the least response times, or even randomly. The concept of *stateless* services, i.e., services that either do not store state information or delegate it to other components, complements load balancing by simplifying the load distribution.

Autoscaling is an emerging technique that originated in cloud computing. It improves scalability and resource management and reduces operating costs through programmatic scaling of the system components based on their usage. Autoscaling can scale from zero to n-number of instances. The ability to go to zero is often called *serverless* computing. However, at least one server which is responsible for the monitoring and scaling is required. Scaling also amplifies the need for observability and monitoring, because autoscaling is enabled by resource utilisation metrics or access to historical load data for scheduled autoscaling.

Resilience and *self-healing* are the principles of automatic recovering from failures. In non-critical software, service failure is accepted and often expected, which motivates system architects to design for failure. Designing services to handle failures of other downstream services increases the overall robustness. Self-healing is when the system can monitor its services' up- and downtimes through health checks, and act upon downtimes by restarting unresponsive or failed components.

Asynchronous communication improves responsiveness and scalability of the system by removing the need for one service to wait for a response from the other. Non-blocking event-based communication, besides increasing the throughput, also makes it possible to introduce parallel message processing when a message has multiple destinations. Choosing standard communication protocols like HTTP/REST, gRPC for blocking communication, or AQMP, and MQTT for message-based communication simplifies the development because of the community support and makes it easier to maintain services in future.

All of the above-mentioned principles and techniques are applied to the current system with the exception of memoisation. Memoisation works by storing computation results in memory and returning them directly when the same computation is submitted. Also, caching is limited only to the user's input to optimise the disk usage in this work. There is less emphasis on memoisation and caching, because it is unlikely that several concurrent sessions will require a BPS model for the same event log. Thus, the impact of memoisation and caching would be limited.

The microservice architectural pattern with independent services and asynchronous decoupled communication is applied to the current design of the extended version of Simod. Containerisation is used to package Simod and other system components into standalone applications for independent versioning and distribution. Load balancing, autoscaling, and self-healing are provided by the Kubernetes orchestration platform. The following chapters describe the role of Kubernetes in more detail.

3.2 Prerequisites for Scalability

Simod is at the core of the current system for scalable BPS model discovery. The starting point for this project is Simod v2.3.1, initially developed by Camargo et al. [CDG20]; [Cam21]. This chapter describes the challenges presented by Simod and how they have been resolved to proceed with the development of scalable and distributed BPS model discovery.

3.2.1 Simod Challenges

As mentioned in Section 1.2, Simod, while being written in Python, used multiple external Java applications for process model discovery, conformance analysis, calendar discovery, and simulation. The need for two development environments significantly complicated the installation and distribution of the software. Neither of the software components had automated builds, tests, packaging, or documentation in the repository, which slowed down any further modifications of Simod. Also, the components' high cohesion and tight coupling made new integrations difficult.

As part of the thesis, the QBP/BIMP simulator (written in Java) was replaced with the Prosimos simulator, a fully Python-based BPS model simulator developed in parallel with this Master's project. Besides being Python-based, which makes the integration with Simod more seamless, Prosimos offers additional functionality not present in QBP/BIMP. For example, the ability to capture calendars assigned to each resource separately, instead of attaching calendars to entire resource pools is a prominent feature of Prosimos [LHD23].

Simod also had an unexpected hardware dependency. Some of the Java libraries in external dependencies used the X11 windowing system even though Simod did not have UI at the moment, nor it or its dependencies drew graphics in OS windows. However, having X11 in the library forced Simod to be connected to a computer display. Without an actual physical display, Simod was not able to run. Therefore, it was impossible to use Simod on servers that usually do not have displays unless a user set up a display and made it available through the network.

3.2.2 Simod Refactoring and Maintenance

To address the above issues, this Master's project started with a major refactoring of the Simod tool, which took it from version 2.3.1 (in July 2021) up to version 3.3.0 (in February 2023). However, not all time was allocated for the work on the Simod's refactoring. It took approximately two semesters of part-time work to refactor Simod (and one more full-time semester to design and implement the extension).

Out of 775 total commits to the code repository on the date of writing, 625 commits have been submitted since v2.3.1, which means that 80% of the contributions to Simod are attributed to refactoring and further modifications of the initial version of Simod. The code base, however, has not grown significantly, from 81.5k lines of code (LOC) to 85k LOC, which is about 4% growth in size. During the course of the refactoring effort, as part of the thesis, the general code quality increased from 3.33/10 to 6.96/10 according to *PyLint v2.17.2* that was run on the Simod v2.3.1 and v3.3.0 with default PyLint settings. The main achievements are summarized in Table 2.

To improve the development process of the Simod system, an *issue management* system has been introduced using *GitHub Issues*[13]. The issue management helped to organize modification requests and track their progress. It also provided a place for discussions related to particular issues and system's components. Together with weekly stand-up meetings introduced to the development team, it made the development process more transparent and measurable.

As part of the thesis, a *continuous integration* (CI) pipeline has been implemented using *GitHub Actions*[14] to automate builds, tests, and packaging of Simod into a Docker image. Containerisation greatly simplified the release and distribution process, and made the installation of Simod effortless and reproducible. *Automation* has been improved with *Ansible* scripts to deploy Simod to testing environments for benchmarking and long-running tests using on-premises servers. GitHub Actions and Ansible scripts also serve as *documentation* source for other developers on how building, testing, packaging and possible deployment can be implemented.

The introduction of *regression tests* helped to recover the knowledge of how to use different modules of Simod. It established the baseline, the minimum level of quality for Simod and increased the confidence in refactoring and releasing new versions. When it made sense, the *test-driven development* (TDD) approach was applied to further Simod modifications. Test coverage has increased from 0 to 73% at the moment of writing, with 91% of files and 77% of lines covered.

Code readability has increased notably with a more careful approach to naming. It is important to be as clear as possible with one's intentions while writing code because developers spend considerable time reading it. So, many variables and Simod configuration parameters received full names to increase comprehension. Removing unnecessary, redundant, and dead code reduced the effort in understanding the code base.

The Simod's maintenance included not only the improvement and refactoring of the existing code base but also the introduction of new changes. ProConformance, a Java tool for conformance checking, was used in Simod to perform the trace alignments to calculate branching probabilities as part of the BPS model discovery. To reduce the dependency on Java, this tool has been replaced by a functionally

[13]https://github.com/AutomatedProcessImprovement/Simod/issues
[14]https://github.com/AutomatedProcessImprovement/Simod/actions

Metric, improvement	Simod v2.3.1	Simod current	Notes
LOC	81470	84996	4% increase
Commits	150	775	80% increase
Code quality	3.33/10	6.96/10	PyLint v2.17.2
Test coverage	0	73%	PyCharm using PyTests
CI	–	+	GitHub Actions
Issues Management	–	+	GitHub Issues

Table 2. Main Simod refactoring metrics and improvements

equivalent Python-based module developed by another team member. Calendar discovery implemented using another custom-built Java application has been replaced by a Python-based module also developed by another team member. As mentioned before, the QBP/BIMP simulator has been replaced with the Prosimos simulator developed by other two team members. Because Simod is complex software with many configuration parameters, Simod's CLI has been improved to add more CLI options and usage hints to streamline the user's experience. Finally, new measures for assessing the quality of the discovered BPS models [Cha+23] have been integrated to Simod.

Overall, the refactoring of Simod enabled its extension from a batch CLI tool for BPS model discovery to a more maintainable, testable, and scalable solution with HTTP API ready for production deployment to serve end-users more reliably. Also, changes in automation, testing, code readability, and packaging most likely improved the development experience and shortened the delivery cycle time for new features.

3.3 System Architecture

Scalability principles mentioned in Chapter 3.1 are directly applied to the current solution. For the presentation of the software architecture in this chapter, the *C4 model*[15] is used. Different perspectives on the software are presented in the order from high- to low-level details. The C4 model initially consisted of four contexts: system context, container, component, and code. However, C4 is not limited to these diagram types and can also contain system landscape, deployment, and dynamic diagrams, for example. The implementation of the solution can be accessed through this meta-repository that has links to all underlying systems, https://github.com/AutomatedProcessImprovement/simod-on-containers. More about the repositories structure is available in the appendix at page 63.

[15]https://c4model.com/

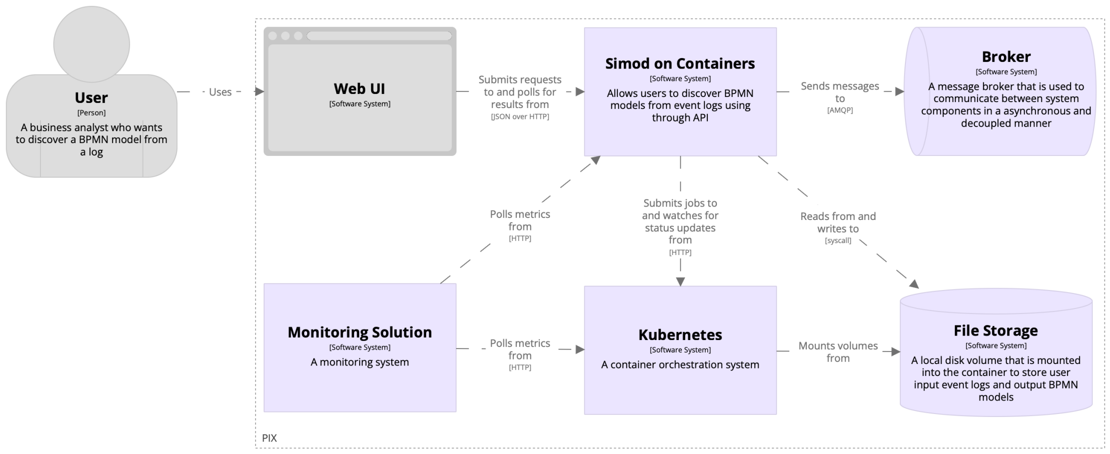

User
[Person]
A business analyst who wants to discover a BPMN model from a log
Uses
Web UI
[Software System]
Submits requests to and polls for results from
[JSON over HTTP]
Simod on Containers
[Software System]
Allows users to discover BPMN models from event logs using through API
Sends messages to
[AMQP]
Broker
[Software System]
A message broker that is used to communicate between system components in a asynchronous and decoupled manner
Polls metrics from
[HTTP]
Submits jobs to and watches for status updates from
[HTTP]
Reads from and writes to
[syscall]
Monitoring Solution
[Software System]
A monitoring system
Polls metrics from
[HTTP]
Kubernetes
[Software System]
A container orchestration system
Mounts volumes from
File Storage
[Software System]
A local disk volume that is mounted into the container to store user input event logs and output BPMN models
PIX

Figure 3. System context diagram of the solution, where the violet colour marks software systems which belong to the current solution

3.3.1 System Context

Figure 3 describes the solution from the highest level, the system context. The dashed box with the *PIX* label indicates the organisational boundary. The organisational context contains multiple software systems. Parts of the current solution are marked with a violet colour. The end-user interacts with the solution indirectly through a web user interface (Web UI) by browsing the web application, uploading event logs, and tuning discovery configuration in the web interface. The web application, which is not part of the current solution, sends the event log and discovery configuration to the current solution. Figure 3 is a simplified representation of the system. A load balancer will first meet the actual POST request before hitting the *Simod on Containers* system. Deployment is described in more detail later in this chapter.

The overall solution contains the aforementioned *Simod on Containers* software system, *Broker* for distributed and asynchronous communication, *Monitoring Solution*, *Kubernetes*, and *File Storage* for event logs and discovery configuration files. Kubernetes is essential to the current solution because it is a platform for running and orchestrating the system. Kubernetes addresses many of the scalability challenges mentioned before. Load balancing, resilience and self-healing, automated deployment, autoscaling, resource scheduling and allocation—these are the roles delegated to Kubernetes. *Simod on Containers* is the core system of the current solution that uses Kubernetes API to schedule, execute, and monitor the execution of Simod jobs described below.

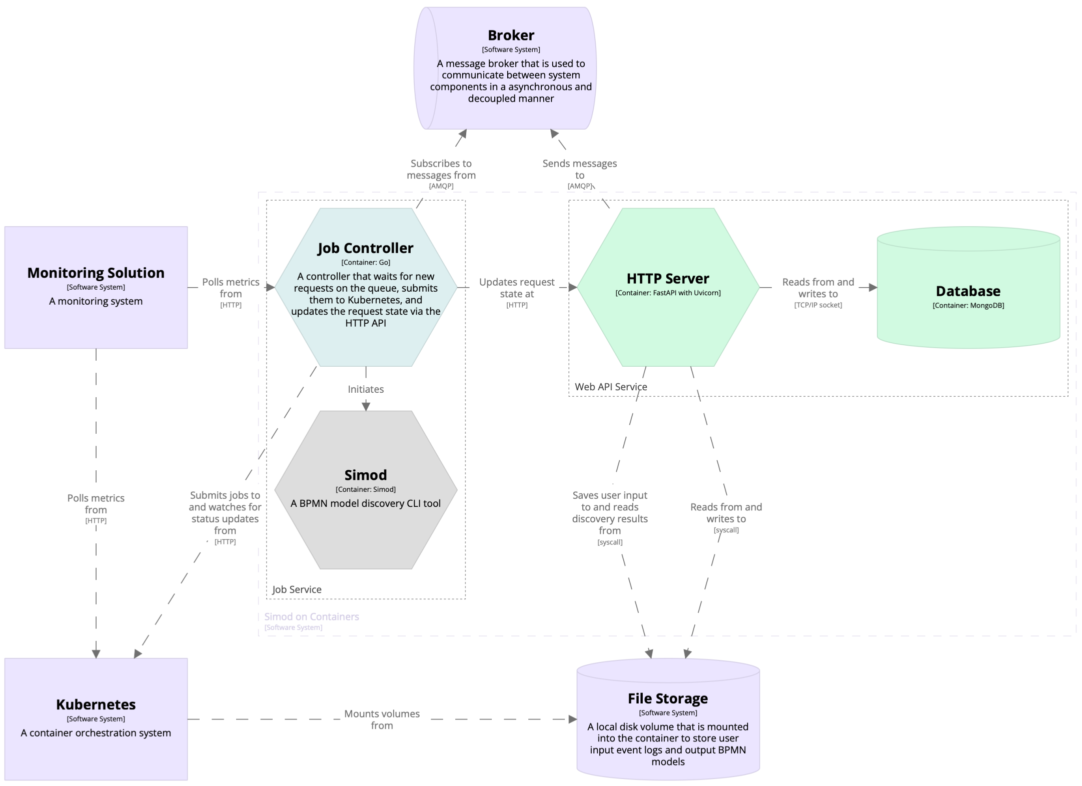

Broker
[Software System]
A message broker that is used to communicate between system components in a asynchronous and decoupled manner
Subscribes to messages from
[AMQP]
Sends messages to
[AMQP]
Monitoring Solution
[Software System]
A monitoring system
Polls metrics from
[HTTP]
Job Controller
[Container: Go]
A controller that waits for new requests on the queue, submits them to Kubernetes, and updates the request state via the HTTP API
Updates request state at
[HTTP]
HTTP Server
[Container: FastAPI with Uvicorn]
Reads from and writes to
[TCP/IP socket]
Database
[Container: MongoDB]
Web API Service
Initiates
Simod
[Container: Simod]
A BPMN model discovery CLI tool
Job Service
Polls metrics from
[HTTP]
Submits jobs to and watches for status updates from
[HTTP]
Saves user input to and reads discovery results from
[syscall]
Reads from and writes to
[syscall]
Simod on Containers
[Software System]
Kubernetes
[Software System]
A container orchestration system
Mounts volumes from
File Storage
[Software System]
A local disk volume that is mounted into the container to store user input event logs and output BPMN models

Figure 4. Simod on Containers' system boundary with its internal containers and related external systems

3.3.2 Container Context

The next level of abstraction in C4 is the container context, where each software system is depicted with its internal containers, i.e., its services, applications, databases, and other subsystems. This section examines each software component in more detail, starting from the core.

Simod on Containers depicted in Figure 4 has two services, *Web API* and *Job Service*. *Web API* is the web service responsible for managing user discovery requests. It takes the user input, records all the relevant information, stores it persistently, and then passes discovery jobs to downstream systems. It has been designed to be horizontally scalable and highly available with little side processing as possible, communicating mostly through the HTTP protocol in JSON format. *Job Service*, on the other hand, is a worker that listens to discovery requests on the queue. When one appears, it initiates a long-running Simod job with the help of Kubernetes' internal scheduler, while in parallel watching for state changes of the jobs through the Kubernetes API. Besides, Job Service exposes metrics

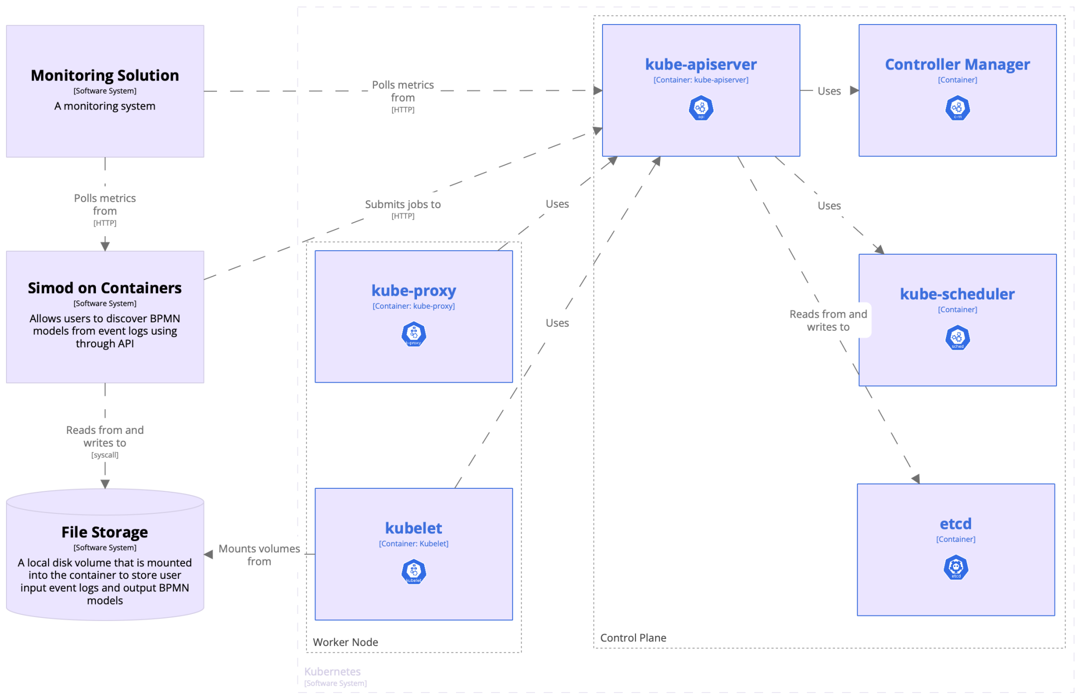


Figure 5. Kubernetes' system boundary with its internal containers and related external systems

such as job waiting time, processing time, and the current number of jobs in the queue, actively running and finished jobs. Broker enables distributed asynchronous communication between internal services in the current solution. Still, Job Service has direct access to Web API through the HTTP API if a discovery job state has been updated. Using the queue for that kind of communication is possible, but this introduces another long-running thread with a broker client and complicates the implementation. However, using the queue for Web API allows us to skip waiting for a response from downstream internal services while processing a user request.

Kubernetes is a well-described, mature, and extensible system[16], so only parts concerning this solution are covered here. Figure 5 depicts its internals in two distinct groups, *Control Plane* and *Worker Node*. *Control Plane* consists of the API server, Controller Manager, Scheduler, and etcd—an internal storage for Control Plane. Any application that integrates into Kubernetes does it through the API server by communicating directly through HTTP or community-developed client libraries. The roles of Kubernetes are manifold in the current solution. Simod on Containers uses Kubernetes to run Simod jobs in a resource-aware manner. Job Service submits jobs through the Kubernetes API server, which are planned,

[16]https://kubernetes.io/docs/concepts/overview/components/

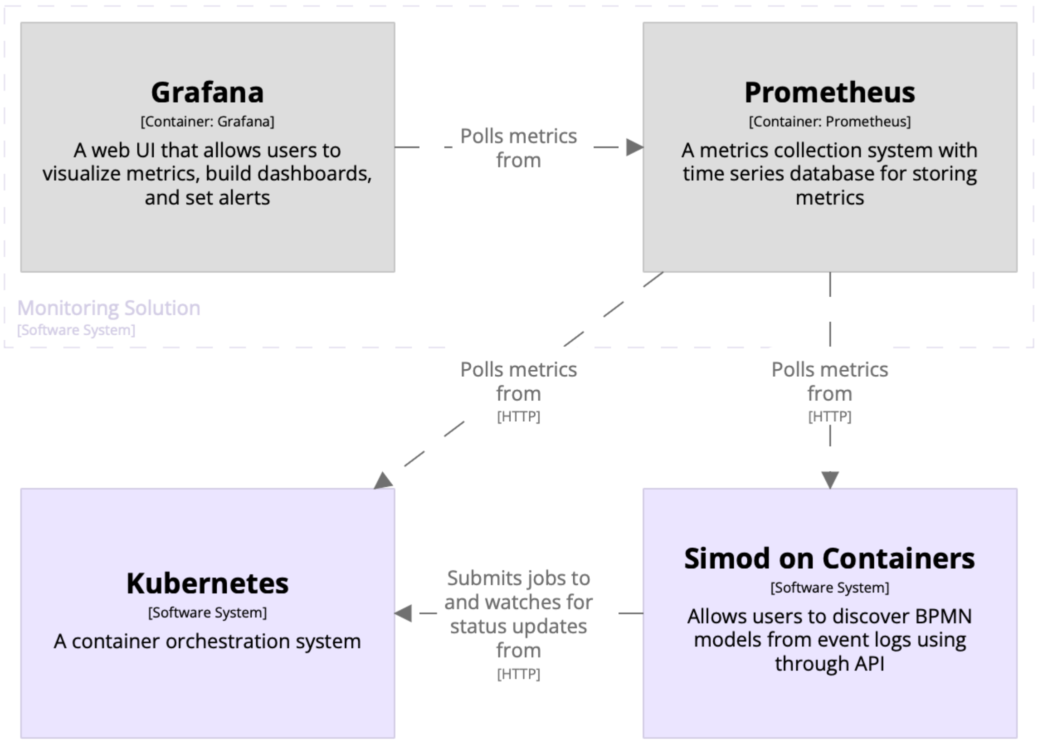

Grafana
[Container: Grafana]
A web UI that allows users to visualize metrics, build dashboards, and set alerts
Polls metrics from
Prometheus
[Container: Prometheus]
A metrics collection system with time series database for storing metrics
Monitoring Solution
[Software System]
Polls metrics from
[HTTP]
Polls metrics from
[HTTP]
Kubernetes
[Software System]
A container orchestration system
Submits jobs to and watches for status updates from
[HTTP]
Simod on Containers
[Software System]
Allows users to discover BPMN models from event logs using through API

Figure 6. Monitoring Solution's system boundary with its internal containers and related external systems

scheduled, and deployed by Kubernetes Scheduler to an available compute node with such capacity. One relation is not shown on the diagram to avoid cluttering, but is ubiquitous, from Kubelet to each service and application in the current solution. Because Kubernetes orchestrates the deployment and running of the solution, Kubelet, installed on each compute node, ensures each application is healthy and running. In case of failures, applications are restarted automatically. Kubernetes also provides load balancing and autoscaling services based on resource utilization for applications that require it. *Worker Node* is a set of daemons that run on any other node except the Control Plane to establish communication between each node of the cluster running user applications and the Kubernetes management software.

Monitoring Solution provides tools for ensuring system observability. The core of the monitoring solution is *Prometheus*[17], the open-source monitoring and alerting software that collects and stores metrics as time-series data. There are

[17]https://prometheus.io

two possible ways of collecting the metrics: serving them with a web server, or the push gateway. In the current solution, *Job Service* exposes job duration and the number of pending, running, succeeded, and failed jobs through a web server running at the *http://localhost/metrics* endpoint. Kubernetes also exposes metrics such as CPU usage, RAM consumption, I/O throughput, and others by default. Prometheus, in its turn, polls these endpoints periodically and stores metrics in the time-series database. *Grafana* is another ready-made software integrated into the system to enhance observability by visualising metrics collected by Prometheus. It allows us to build a dashboard for the project, run queries against the Prometheus database, and visualise the results at near real-time resolution.

Broker, as mentioned before, decouples communication between Web API and the rest of the system by providing a durable message queue that survives system reloads. Job Service is subscribed to the queue and starts a lightweight co-routine for each message processing, allowing for concurrently serving many requests.

File Storage for storing user inputs is implemented using an ordinary Linux file system. It can be replaced with blob storage if deployed in a cloud-native environment. However, it was not necessary for this project, so the most straightforward and still reliable solution has been chosen. File Storage must be attached to Web API and Kubernetes worker nodes that run Simod jobs.

3.3.3 Components Context

The component's context level examines each container closer. C4 modelling also has code-level diagrams, but they are omitted in this description because components in Simod on Containers are not very extensive and often directly correlate with classes in the code. Thus, to avoid unnecessary details, the code-level diagrams are replaced with dynamic diagrams, which are more helpful in understanding the system's behaviour at runtime.

Web API consists of the **HTTP Server** and the internal database. *MongoDB v6.0.5* plays the role of the NoSQL database. The NoSQL or document type of the database helps to avoid strict database schemas in this project that introduce unnecessary constraints at the beginning of the development. Also, document databases suit more the distributed context and scale horizontally more easily than row-based tables. HTTP Server is the core of the web service, and its components are shown in Figure 7. The web application is implemented in Python using *FastAPI v0.92.0*[18] and runs on top of the *Uvicorn*[19] web server which runs multiple instances of the same application in parallel. The web application ensures that the asynchronous run of its internal components is safe and will not corrupt the data or block underlying resources.

[18]https://fastapi.tiangolo.com
[19]https://www.uvicorn.org

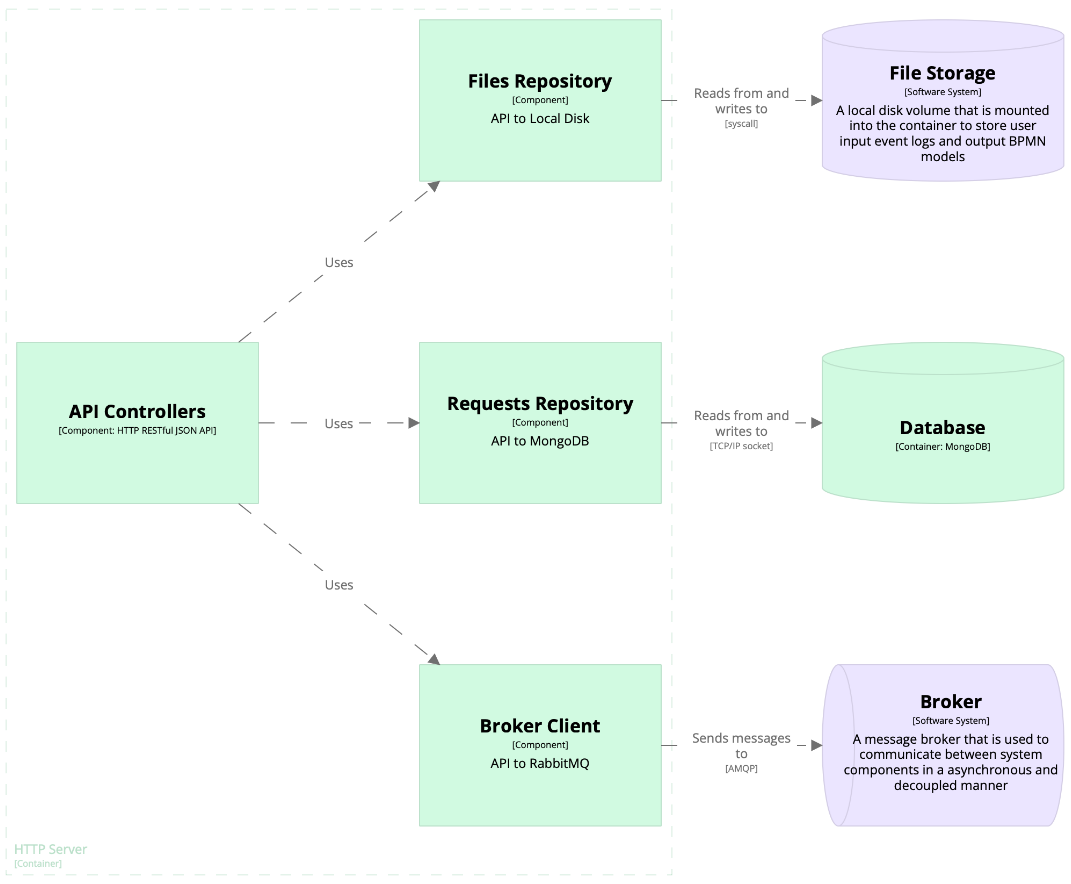
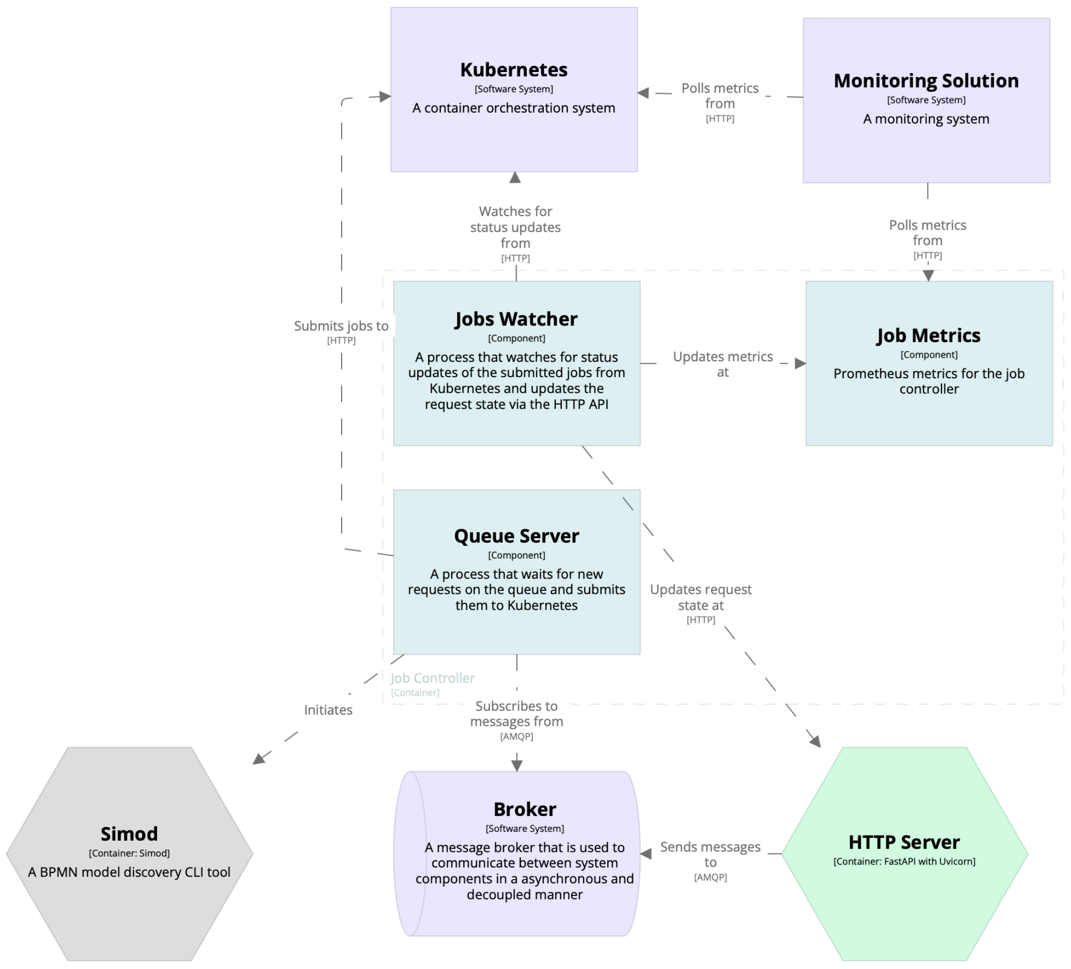


Figure 7. HTTP Server's boundary with its internal components marked in green and related external systems in violet

The Web API service has three underlying resources: the broker's queue, database, and file storage. The broker, *RabbitMQ v3.11*, and Mongo database can sustain multiple asynchronous connections through client libraries developed by the community. The Linux OS file system is used as the file storage for user inputs and does not require any additional connections, but the disk volume must be attached to containers that use the file system. The database contains only the metadata about user requests and paths to user inputs in the file system. There is a possibility of data races when using the pure file system. To avoid that, Web API treats input data as immutable writing bytes to disk only once and then uses it for read-only operations. Besides, Web API computes a SHA-256 hash of the file content and uses it as a file name, which helps with caching. If the file is already in the file system, its processing is stopped in the service to avoid redundant writes.

Job Controller shown in Figure 8 is the core part of the Job Service. The



Kubernetes
[Software System]
A container orchestration system
Polls metrics from
[HTTP]
Monitoring Solution
[Software System]
A monitoring system
Watches for status updates from
[HTTP]
Polls metrics from
[HTTP]
Submits jobs to
[HTTP]
Jobs Watcher
[Component]
A process that watches for status updates of the submitted jobs from Kubernetes and updates the request state via the HTTP API
Updates metrics at
Job Metrics
[Component]
Prometheus metrics for the job controller
Queue Server
[Component]
A process that waits for new requests on the queue and submits them to Kubernetes
Updates request state at
[HTTP]
Job Controller
[Container]
Initiates
Subscribes to messages from
[AMQP]
Simod
[Container: Simod]
A BPMN model discovery CLI tool
Broker
[Software System]
A message broker that is used to communicate between system components in a asynchronous and decoupled manner
Sends messages to
[AMQP]
HTTP Server
[Container: FastAPI with Uvicorn]

Figure 8. Job Controller's boundary with its internal components marked in light blue and related external systems in violet

Verb	Endpoint	Description
GET	/discoveries	List discovery requests.
POST	/discoveries	Create a discovery request. Simod configuration and event log must be provided in the body as form data.
DELETE	/discoveries	Delete all discovery requests.
GET	/discoveries/{request_id}	Get general information about the request.
GET	/discoveries/{request_id}/ {file_name}	Get a file attributed to the discovery request. Used to fetch discovery results when ready.
GET	/discoveries/{request_id}/ configuration	Get configuration of the discovery request.
PATCH	/discoveries/{request_id}	Update the request metadata. Used to update the request's status by *Job Service*.
DELETE	/discoveries/{request_id}	Delete the request.

Table 3. HTTP Service API endpoints

application is written in Go and is split into several internal lightweight workers (servers) that run concurrently. These workers are *Queue Server*, *Jobs Watcher*, and *Job Metrics*. *Queue Server* is subscribed to the Broker's queue and waits for discovery requests. As a request is published to the queue, the server receives the message, composes a description of a Simod job, and submits it to Kubernetes for scheduling and execution. In parallel, *Jobs Watcher* is subscribed to events related to Simod jobs through the Kubernetes API. If there is any change in a Simod job, the worker gets notified and updates the state of the discovery request at the HTTP Server. Besides, Jobs Watcher updates job-related metrics (job duration, and the number of pending, active, succeeded, and failed jobs) in the *Job Metrics* worker. *Job Metrics* is a web server that exposes metrics at the well-known endpoint for Prometheus at *http://localhost/metrics*.

Simod is marked in grey in Figure 8 because it is not an active part of the Job Service. Job Controller composes a description of a Simod job specifying the particular version of the Docker image of Simod, command arguments for the Simod start-up, and path to the attached disk volume to access user inputs. Job Controller also specifies the number of computing resources needed for a job so that Kubernetes can better schedule the deployment of the job to nodes that can accommodate such a load.

3.3.4 Collaboration between Services

The current solution has one obvious potential hot spot and one bottleneck. The hot spot is the HTTP Server that accepts incoming requests from users. Because user requests most likely would have exponential distribution, e.g., many visitors come simultaneously after a link to the website is published in a blog article on

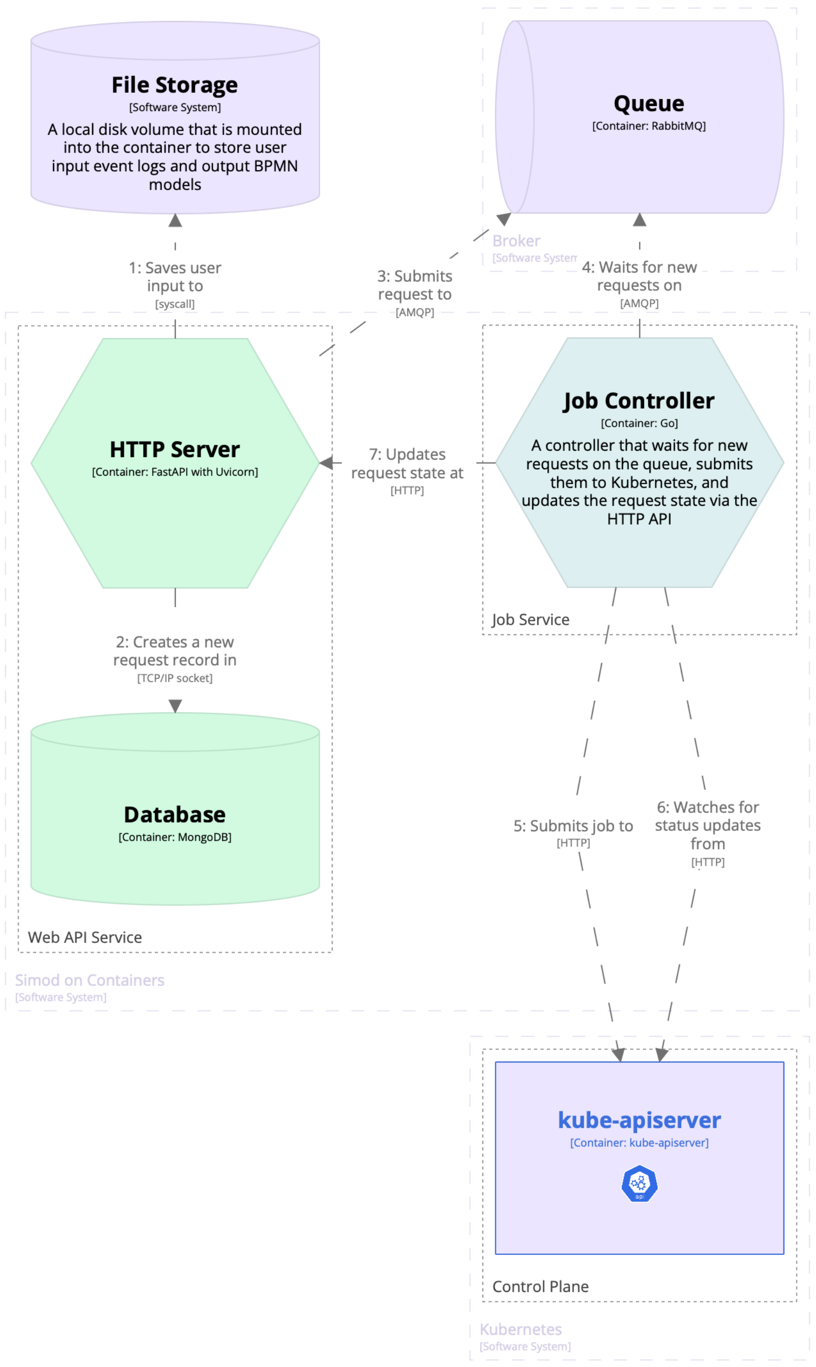

File Storage
[Software System]
A local disk volume that is mounted into the container to store user input event logs and output BPMN models
Queue
[Container: RabbitMQ]
Broker
[Software System]
1: Saves user input to
[syscall]
3: Submits request to
[AMQP]
4: Waits for new requests on
[AMQP]
HTTP Server
[Container: FastAPI with Uvicorn]
7: Updates request state at
[HTTP]
Job Controller
[Container: Go]
A controller that waits for new requests on the queue, submits them to Kubernetes, and updates the request state via the HTTP API
Job Service
2: Creates a new request record in
[TCP/IP socket]
Database
[Container: MongoDB]
Web API Service
5: Submits job to
[HTTP]
6: Watches for status updates from
[HTTP]
Simod on Containers
[Software System]
kube-apiserver
[Container: kube-apiserver]
Control Plane
Kubernetes
[Software System]

Figure 9. Dynamic diagram visualising communication steps when a new discovery request comes in from the upstream Web UI. Communication technology is specified in square brackets on relationship arrows

a university's website, HTTP Server is the first component that faces the peak load. As mentioned in Section 3.1, asynchronous message-based communication is good practice to increase service availability. Thus, the communication between Web API and Job Service is organised in a distributed manner using RabbitMQ as a message broker, which implements *AMQP 0-9-1* protocol.

However, the communication from the Job Service to Web API is implemented through the HTTP REST API. Job Service is the system's bottleneck because discovery jobs with Simod can take a significant amount of time, from minutes to hours. Job Service sends messages to Web API updating discovery request statuses. As jobs take a long time, there would be few simultaneous updates to Web API. Therefore, the frequency of communication initiated by Job Service is low and limited by the number of available compute nodes. If there are 4 nodes, there would be a maximum of 4 concurrent updates to Web API. That is why the HTTP REST communication has been chosen for Job Controller instead of the queue. This simplifies the development because blocking communication is easier to implement and debug than distributed message-based communication.

Activity diagrams help examine a system's dynamic nature by visualising the order of interactions between subsystems. Figure 9 is a dynamic diagram for a use case when a discovery request comes in from Web UI. The order of operations is labelled with numbers on the arrows. (1) The HTTP Server accepts a POST request, and the user input is saved to the file storage. (2) The HTTP Server creates a record in the database for the new request with file paths to user inputs in the file system, also a timestamp, and the current request status (ACCEPTED at this step). Because a discovery job can take a considerable amount of time, the *Request–Acknowledge–Polling* communication pattern has been chosen for the web service. After processing the user input successfully, the HTTP Server returns the 202 ACKNOWLEDGED status code. From this point, the client application (Web UI) can start polling the web service for updates and results of the BPS model discovery. Table 3 shows the complete list of HTTP Server API endpoints.

After processing the user input, (3) the HTTP Server publishes the new request message to the broker in a separate background thread. The message is a JSON object encoded as an array of bytes. It contains the new request identifier and path to the Simod configuration file, with all the information needed for Simod to discover a BPS model. Then, (4) the Job Controller gets the message, prepares a Simod job description for Kubernetes specifying the path to the configuration file, and (5) submits it for execution through the Kubernetes API. (6) At the same time, there is already a running process in a separate thread that watches updates for Simod jobs from Kubernetes. If a status update happens, Job Controller updates the metrics values, archives the results, and (7) updates the status through the HTTP Server.

3.4 Scalability of Web API

Scalability of Web API is enabled by *asynchronous (concurrent) programming* for requests processing and by Kubernetes *Horizontal Pod Autoscaler*[20]. Endpoints in the HTTP Server are declared using the *async/await* Python syntax supported by the FastAPI web framework. This allows the Python interpreter to suspend long-running functions usually related to I/O operations, switch to other tasks, and then resume the suspended execution finishing the operation.

Concurrency can help increase processing efficiency and give a certain speed-up, but it is not enough for highly scalable web servers. Another way to improve the scalability is by running several web servers in parallel. Web API uses Uvicorn v0.20 as a web server for running the FastAPI application. Uvicorn implements *Asynchronous Server Gateway Interface* (ASGI)[21] that enables asynchronous processing of multiple events simultaneously. Additionally, it introduces support for background tasks for a web server, which are used in the HTTP Server to send messages to the broker. Background tasks remove the need for the user to wait for the completion of downstream calls before receiving a response.

Kubernetes utilises ASGI implemented by Uvicorn and plays the role of a process manager. This brings us to the *Horizontal Pod Autoscaler* (HPA), a Kubernetes component responsible for horizontal scaling in response to certain metrics. For the autoscaler to work, Kubernetes requires the *Metrics Server*[22] to be deployed to the Kubernetes cluster. The Metrics Server collects Kubernetes Pods resource metrics such as CPU or memory utilisation, which can be used for scaling. For example, if the average CPU utilisation goes above 50%, then scale up to n-amount of nodes. Scaling down is also can be additionally configured.

```yaml
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: simod-http
spec:
  maxReplicas: 6
  minReplicas: 2
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: simod-http
```

Listing 1. Horizontal Pod Autoscaler manifest in YAML for *HTTP Server*

[20]https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/
[21]https://asgi.readthedocs.io/en/latest/specs/main.html
[22]https://github.com/kubernetes-sigs/metrics-server

The Web API uses the default settings for scaling up and down: if the CPU utilization goes above 50%, HPA scales up to a maximum number of replicas; if the CPU utilization goes below 50%, HPA scale down. Kubernetes accepts configuration as a manifest file in YAML format. Listing 1 shows the HPA manifest for autoscaling the HTTP Server. It includes the minimum and maximum number of replicas that result in the same number of parallel Uvicorn web servers. Also, for HPA to find what should be automatically scaled, the property `scaleTargetRef` points at the deployment, which is marked with `{name: simod-http}`. The entire manifest for the Web API deployment is presented at page 64.

3.5 Scalability of Job Service

The Job Service is not a hot spot like Web API, and it does not face external systems or users. It processes discovery requests from the queue at its own pace depending on the available computing resources. A single Simod discovery can take from minutes to hours. If it takes about a minute to discover a BPS model using Simod for a lightweight event log, and there are 100 pending requests in the queue, then it would take about 1.7 hours to run all the discovery jobs. Thus, it does not matter if the Job Service processes pending requests in 1, 5, or 15 seconds.

Nevertheless, the Job Service is a performant application. It is written in Go to utilize Go's well-known concurrency model[23] of the language based on *goroutines*. Goroutines allow cheap and lightweight concurrent execution in the same address space with a small stack, dynamic heap allocation and freeing. One can start hundreds of goroutines without overwhelming the Go runtime. Goroutines are used to start the Queue Server, Jobs Watcher, and Job Metrics components shown in Figure 8. It means that all those workers run concurrently, most likely in parallel threads (Go runtime manages it under the hood). When a new request comes in, a new goroutine launches without blocking the processing of the subsequent request. This concurrent model is similar to the process manager with ASGI mechanism described in the section about the scaling of Web API.

It is possible to use HPA for autoscaling the Job Service too. However, there is no need at the moment because the system bottleneck is still downstream, and the current architecture of the service can handle peak loads without issues. The entire manifest for the Job Service deployment is provided at page 66.

[23]https://go.dev/doc/effective_go#concurrency

3.6 Usage of Language Models and Smart Assistants

During the work on the thesis, such large language models (LLMs) as *GPT-3.5*, *GPT-4*, and *Codex* were used[24]. GPT models were used as conversational assistants that helped to explore a topic, suggest some starting points for the research, relevant tools and technologies. LLMs were used in the same manner as search engines for the web to do the initial exploration of the field before going deeper. LLMs were not used to generate text for this thesis or for paraphrase. *Grammarly*[25] was used instead as a spell-checker and tool to improve the writing. Codex in the form of the *GitHub Copilot* plugin to code editors (JetBrains' PyCharm, GoLand, and Visual Studio Code) was used as an intelligent auto-completion during the implementation of applications mentioned in the thesis.

[24]https://platform.openai.com/docs/models
[25]https://www.grammarly.com

4 Validation and Results

There might be multiple definitions of scalability, and often an owner of a system adjusts those definitions to the needs of a company. This work defines *scalability* as the ability to handle an increased workload with the extended system's capacity while maintaining efficiency [WG06].

In modern cloud environments, we also define several types of application scaling—*vertical*, *horizontal*, and *cluster scaling*[26]. Vertical scaling is aimed at adjusting a single component of a resource for the application, e.g., increasing RAM or CPU cores. Horizontal scaling increases the number of instances of the application to match higher demand. Cluster scaling adds new resources to the cluster, e.g., by changing the number of compute nodes.

This chapter establishes the validation approach to test the system's scalability using different scaling techniques for load management. Also, it describes another important aspect of the validation, the implementation of the system's observability, which is required for collecting measurements for the validation. Finally, the results of the validation experiments are presented.

4.1 Validation Framework

The validation of the current solution focuses on horizontal and cluster scaling. Scalability can be achieved either by horizontal scaling of a system component or cluster scaling, which usually goes together with horizontal scaling because adding more compute nodes allows deploying more application instances to handle more load.

Two experiments have been designed to validate if scalability is achieved. One of the experiments ensures that the system maintains its efficiency under an increased load with an extended capacity by employing cluster and horizontal scaling. The other experiment shows that the system can sustain peak loads without collapsing using horizontal scaling. The peak loads are achieved by ever growing number of concurrent user sessions leading to a constant increase in requests rate.

4.1.1 Scalability Experiment

Design. The scalability experiment aims at validating if the system maintains its efficiency under an increased load with an extended capacity. *Efficiency* is defined as the ratio of the real capacity of the system over the ideal capacity [WG06, p. 14]:

$$E_n = \frac{\sum_{i=1}^{n} D_i}{nD_1} \tag{1}$$

[26]https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/

where D_i stands for the maximum added demand (capacity) by the i-th unit of the system. For example, one compute node can satisfy the maximum demand D_1, the second node satisfies the maximum demand D_2, and then the real capacity of the system is $D_1 + D_2$. The ideal capacity is D_1 because doubling the system's resources, e.g., adding a second compute node, usually does not double the capacity because of additional overhead. In that case, D_2 is expected to be less than D_1. Thus, nD_1 is considered the ideal system capacity where n is a number of scaled resources.

Additionally, the scalability experiment requires a response metric that must be met by the system [WG06, p. 13]. In a more complex setting, it can be an elaborate performance model. In this work, the *cycle time* of a Simod job is the response metric, i.e., the time since the job has been submitted to the queue and received the PENDING status, until it has successfully finished (SUCCEEDED) or failed (FAILED). Further in the text, the cycle time is also called the *response time.*

The *Throughput* or the number of processed discovery jobs per unit of time is used as a measure of the demand in this experiment. The maximum number of processed jobs given n-nodes is the capacity of the system. To determine the capacity of the system with n-nodes, the system will be under load until it does not meet the target response time, which is calculated below, or the experiment times out.

This experiment considers mainly the Job Controller, Simod, and Kubernetes from Figure 4 while intentionally omitting the solution's Web API component. This is because the response times of the two main services, Web API and Job Service, are on different scales. It takes about 500 *ms* to process an HTTP request and about 5 *min* to discover a BPS model. Therefore, scalability is judged by the system's bottleneck, its *Job Service* and related components. The *Web Service* is evaluated in the other experiment where robustness under load is tested.

Workload. The workload for the test consists of a series of POST requests with two files as a payload, an event log in the XES format, and a configuration file in YAML for Simod which specifies how the BPS model discovery should be conducted. The configuration file is the same for each test (a complete copy of this file can be found at page 68). *PrepaidTravelCost.xes* (7.8 MB) [Don20a] and *RequestForPayment.xes* (15.2 MB) [Don20b] are the two event logs used in the experiment. **Challenge:** Because of the overhead of running the whole Kubernetes cluster in the testing environment using a single laptop, it is impossible to run logs of any size without crashing the cluster. This is why bigger event logs are not considered in the experiments.

Target response time. The response time target is established as the mean response time given the two different payloads used in the tests. Mean response times for the payloads were measured in the testing environment by running POST requests three times without additional load. The target times are summarized in Table 4. The *HTTP RT* column specifies the time it took to receive the first acknowledgement response from the HTTP Server. The *Job RT* column shows the cycle time of a job. From the table, we see that it took, on average, 298.44 seconds to process the PrepaidTravelCost event log and 510.93 seconds to process the RequestForPayment log.

Event log	**Run no.**	**HTTP RT, s**	**Job RT, s**
PrepaidTravelCost	1	0.63	326.00
	2	0.29	149.00
	3	0.39	419.00
	mean	0.44	298.00
RequestForPayment	1	0.82	574.00
	2	0.56	483.00
	3	0.42	474.00
	mean	0.60	510.33

Table 4. Baseline measurements for both workloads where *RT* is the response time

Based on the measurements from Table 4, the target response time is taken as the double of the baseline mean time because the job processing varies widely. Also, it is expected for the system to experience some degradation in response time under high load. Thus, the target response time for PrepaidTravelCost is 596 *s* or 10 *min*, and for RequestForPayment it is 1021 *s* or 17 *min*.

Test implementation. To validate the scalability using the efficiency equation (1), the maximum capacity is required for each additional compute node. In this test, the cluster of 1, 2, 3, and 4 worker nodes is considered. In the Kubernetes terminology, there are two types of nodes, the *Control Plane* and *Worker Node*. It is highly recommended[27] to keep the control plane on a separate node to ensure the load on the cluster does not hinder the cluster management applications of Kubernetes. It means if the cluster has one worker node, it actually has two compute nodes, because one of them is the control plane node.

This experiment uses the *Locust v2.15*[28] load testing tool to implement the workload and simulate users posting requests to the system[29]. Locust has been launched with the following test parameters:

- the maximum *number of users*: 30
- *spawn rate* of users per second: 1

[27]https://kubernetes.io/docs/concepts/overview/components/
[28]https://locust.io
[29]Locust load profiles are available as Python scripts at https://github.com/AutomatedProcessImprovement/simod-load-testing/tree/0.2.1/src

- *running time* of the load test: 30*s*

These parameters give up to 100 discovery jobs for the experiment, enough to calculate the capacity. Because a job is expected to take at least 5 *min* each, and each experimental run was limited to about 10 *min*, most jobs were not awaited and cancelled after the timeout. Thus, larger parameter values are not needed.

Additionally, each test has a load profile (user profile) that describes how and when to apply the load for a single simulated user. User profiles may differ for simulating more complex behaviour. The simplest user profile with constant waiting time between user requests is used in this experiment because it is sufficient to generate enough load and calculate the throughput of the system with different number of worker nodes.

4.1.2 Robustness under Load Experiment

Design and implementation. Stress testing of the HTTP component of the solution helps to analyse if the system can handle peak loads gracefully without collapsing. To do that, the system is a subject to high load for 3 *min* with the following Locust parameters:

- the maximum *number of users*: 200
- *spawn rate* of users per second: 5
- *running time* of the load test: 3*min*

Depending on the network throughput, Locust can generate more than a thousand requests to the system with these parameters. Besides Locust parameters, the load profile is switched to the one with the exponential waiting time between subsequent POST requests. That helps generate a peak load at one point and continues to generate moderate loads the rest of the time.

In the experiment, request-related metrics such as *requests rate*, *total request count*, and *average response time* are collected. To check if Kubernetes can apply automatic horizontal scaling to Web API, the *number of running HTTP Server pods* is measured. Also, *failure rate* is used to judge the overall robustness under the load.

Workload. Similarly to the scalability experiment, the workload for the robustness experiment consists of a series of POST requests with two files as a payload, an event log in the XES format, and a Simod configuration file in YAML. However, only one event log is used for the experiment, PrepaidTravelCost (7.8 MB) [Don20a]. The travel log is a smaller event log that suits the experiment better because of the specifics of the testing environment that does not allow for much higher loads.

4.2 Testing Environment

Considering the testing environment during the load testing is important because it directly impacts measurements collected from the system under test (SUT). The primary goal of the testing environment for this study is to provide a way to execute the whole system in a similar but not exact manner how that can be done in production. Because the system has not been deployed to production yet, there is no easily available Kubernetes cluster for testing. Also, creating or renting an already existing cluster incurs significant costs. For this reason, the local Kubernetes cluster has been set up using *kind*[30], a tool for setting up Kubernetes clusters for development purposes.

All tests are conducted on a laptop with the macOS 13.3 operating system, Apple M1 Pro chip with 10 cores, and 16 GB LPDDR5 RAM using Docker Desktop v4.17.0 as a container runtime. However, only 6 cores, 10 GB of RAM, 1 GB of swap, 64 GB of disk are allocated for Docker.

Testing scenarios[31] are implemented using Locust v2.15[32] and run in Kubernetes v1.25. Measurements are collected using Locust built-in reporting functionality, the Kubernetes Metrics Server, Prometheus[33], and visualised using Grafana[34] and ad-hoc Python scripts.

4.3 Observability

The main purpose of the Monitoring Solution from Figure 6 is to provide observability of certain aspects of the system needed for the experiments. Kubernetes exposes many metrics by default[35], such as the number of running pods, nodes, resource utilization per container, per pod, network and disk I/O, and others. The rest of the components of the system require the metrics to be defined and exported by the system so that Prometheus can collect them for further querying and processing.

Locust collects its own metrics during the load testing that are sufficient for the tests. However, in the Job Service, the metrics such as pending, running, succeeded, and failed job counts and job cycle times are instrumented using the Prometheus SDK[36] for the Go language. The metrics are exposed through a web server at the endpoint *http://localhost/metrics*. This well-known location for Prometheus metrics gets polled automatically if available. Also, such components as RabbitMQ,

[30]https://kind.sigs.k8s.io

[31]https://github.com/AutomatedProcessImprovement/simod-load-testing/tree/0.2.1/src

[32]https://locust.io

[33]*Prometheus* is an open-source systems monitoring and alerting toolkit, https://prometheus.io

[34]*Grafana* is a metrics visualisation toolkit, https://grafana.com/grafana/

[35]https://kubernetes.io/docs/reference/instrumentation/metrics/

[36]https://github.com/prometheus/client_golang

MongoDB, and the HTTP Server can expose their metrics to increase observability, but it was unnecessary for the current evaluation.

4.4 Results

This section presents the results of the experiments described in Section 4.1. The scalability experiment results demonstrate if the scalability has been achieved, while the robustness experiment shows the system can handle peak loads and remain available.

4.4.1 Scalability Experiment

The results of the scalability experiment are summarised in Table 5. The table shows the cluster and workload configuration, target response times, and measures for eight experimental runs for both event logs. The experiments were executed in the testing environment using Kubernetes clusters with 1, 2, 3, and 4 worker nodes. Additionally, the system's capacity in requests per minute was calculated, which allowed it to compute the efficiency metric needed for the evaluation.

The efficiency metric for both event logs is visualised in Figure 10. The plot shows that for the PrepaidTravelCost event log, the efficiency is maintained with the extended capacity of the system. There is almost no efficiency drop when extending from 1 to 2 worker nodes, 30% drop when extending further to 3 nodes, and only 8% drop when extending to 4 nodes compared to the original efficiency of 100% using only one node.

However, the results are different for the other event log, RequestForPayment, which is twice the size of the Travel log. There is constant degradation of efficiency with more worker nodes added. The most significant drop of 34% is when extending from 1 to 2 nodes, and then the degradation is more constant but significant, plus 15% with each additional node.

4.4.2 Robustness under Load

The results of the robustness under load experiment are summarised in Table 6 and visualised in Figures 11 to 15. PrepaidTravelCost, and four experimental runs were conducted with only one event log. The table presents the measures collected with Locust: the total amount of requests posted to Web API, the run time of each experimental run (3 min), the total amount of failures, average response time, throughput in requests per second, and the failure rate.

In Table 6, the number of total requests posted during the 3 min period varies depending on the number of nodes. For the 1-worker node cluster, there were 885 POST requests submitted to *Web API*. There was a drop from 885 to 246

ID	Nodes	Log	Jobs	Elapsed time	Target RT	Mean RT	Capacity	Efficiency
1	1	I	2	9.48 min	10 min	6.33 min	0.22 rpm	1.00
2	2	I	4	9.52 min	10 min	6.10 min	0.42 rpm	1.00
3	3	I	5	11.15 min	10 min	7.45 min	0.45 rpm	0.71
4	4	I	7	9.05 min	10 min	6.21 min	0.77 rpm	0.92
5	1	II	2	14.30 min	17 min	10.65 min	0.14 rpm	1.00
6	2	II	2	11.18 min	17 min	9.22 min	0.18 rpm	0.64
7	3	II	2	9.55 min	17 min	8.96 min	0.21 rpm	0.50
8	4	II	2	11.32 min	17 min	10.21 min	0.18 rpm	0.32

Table 5. Results of the scalability experiment on both event logs where *Nodes*—number of worker nodes; *Log*—event log (I: PrepaidTravelCost.xes, II: RequestForPayment.xes); *Jobs*—number of successfully finished jobs; *Elapsed time*—duration of the experimental run; *Target RT*—target response time; *Mean RT*—mean cycle time of a job; *Capacity*—throughput in requests per min

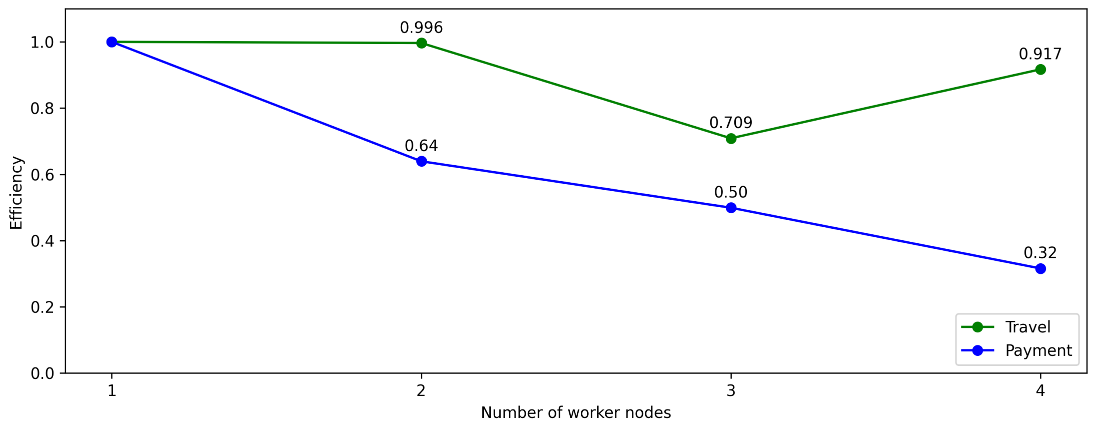


Figure 10. Efficiency with the extended capacity during the scalability experiment for both event logs where *Travel* is the PrepaidTravelCost.xes log, *Payment*—RequestForPayment.xes

requests for the 2-worker nodes cluster. Then, the number of requests stayed more consistent, with 438 requests for the 3-node cluster and 236 requests for the 4-node cluster. The number of failures, in general, rose with more nodes added. The throughput dropped significantly for the clusters with more than one node and remained consistent.

Nevertheless, the average response time improved with more nodes in the cluster. Figure 11 visualised the average and median response times and failure rates for the PrepaidTravelCost log. Response time per request goes down with more nodes added. It drops from the average response time of 19 s per request on the 1-node cluster to 11 s on the 2- and 3-node clusters, finally dropping to 4.8 s on the 4-node cluster. The median response time is even better, dropping from 16 s for the 1-node cluster to 1.4 s for the 4-nodes cluster.

The failure rate rises with more nodes added to the cluster but stays relatively low with a maximum of 4% failures on the 4-nodes cluster. There were two types of failure, *connection refused* and *timeout*. Also, in some cases, the database had more submitted discovery requests than reported by Locust, which means that the request was sent and processed, but the response with acknowledgement was not sent.

Figures 12 to 15 provide supplementary visualisation of the four experimental runs with the running time on the x-axis and metrics on the y-axis. Among the metrics, there are the request rate (requests per second), total request count over time, number of running *HTTP Server* pods, average response time in seconds, and failure rate in requests per second. These plots have similar shapes for all the metrics, with a significant spike in requests during the first minute of the experiment followed by a moderate request rate afterwards, which is expected because the same load profile is used for all the runs. Then, looking at the number of running *HTTP Server* pods, it is visible how Kubernetes managed to autoscale the web server from 2 to 3, 4, and up to 6 concurrent web servers depending on the number of available nodes. It also takes about 15 s to scale from 2 to 4 pods, or from 4 to 6, or even from 2 to 6. However, we also see that the always-running two web server pods usually handle the peak, and scaling happens after the load peak occurs. It is caused by the fact that Kubernetes needs to record the spike in the metrics server first before responding with scaling. The failure rate is usually non-zero during the highest load or the scaling of the web server component.

ID	**Nodes**	**Requests**	**Run Time**	**Failures**	**Avg. RT**	**Throughput**	**Failure rate**
9	1	885	3 min	1	19.0 s	4.92 rps	0.00
10	2	246	3 min	5	11.0 s	1.37 rps	0.02
11	3	438	3 min	12	11.0 s	2.43 rps	0.03
12	4	236	3 min	9	4.8 s	1.31 rps	0.04

Table 6. Results of the robustness experiment on PrepaidTravelCost.xes where *Nodes*—number of worker nodes; *RT*—response time; *rps*—requests per second

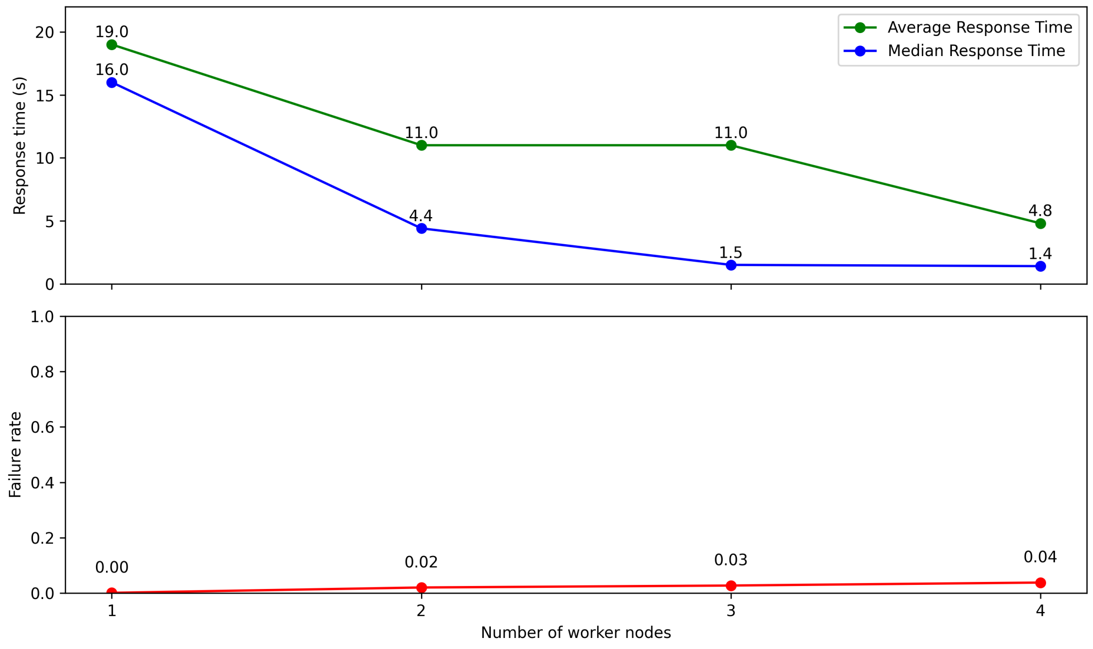


Figure 11. Response time of the HTTP Server with the extended capacity during the robustness experiment for the PrepaidTravelCost.xes event log

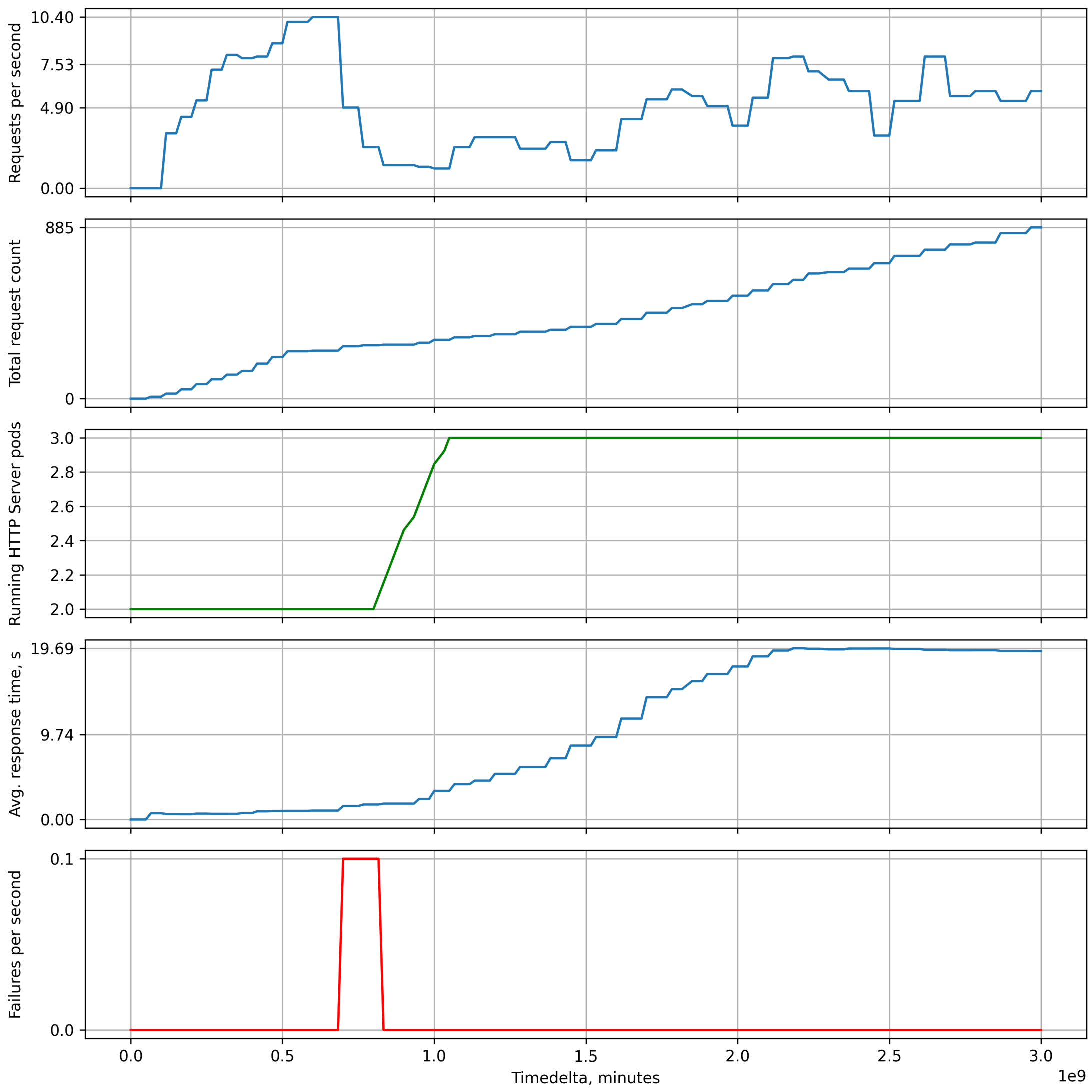


Figure 12. Visualisation of measurements collected during the robustness experiment with 1 worker node

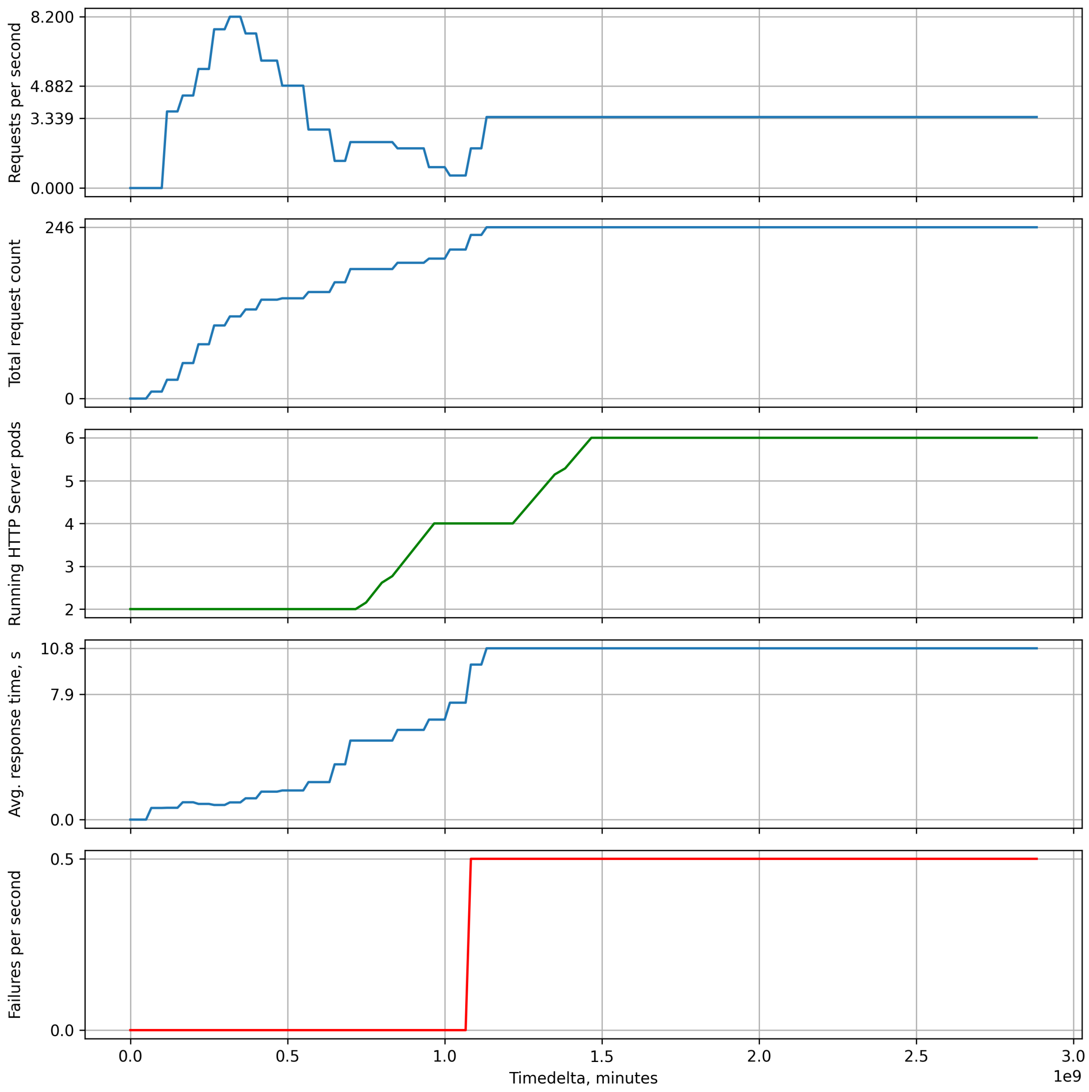


Figure 13. Visualisation of measurements collected during the robustness experiment with 2 worker nodes

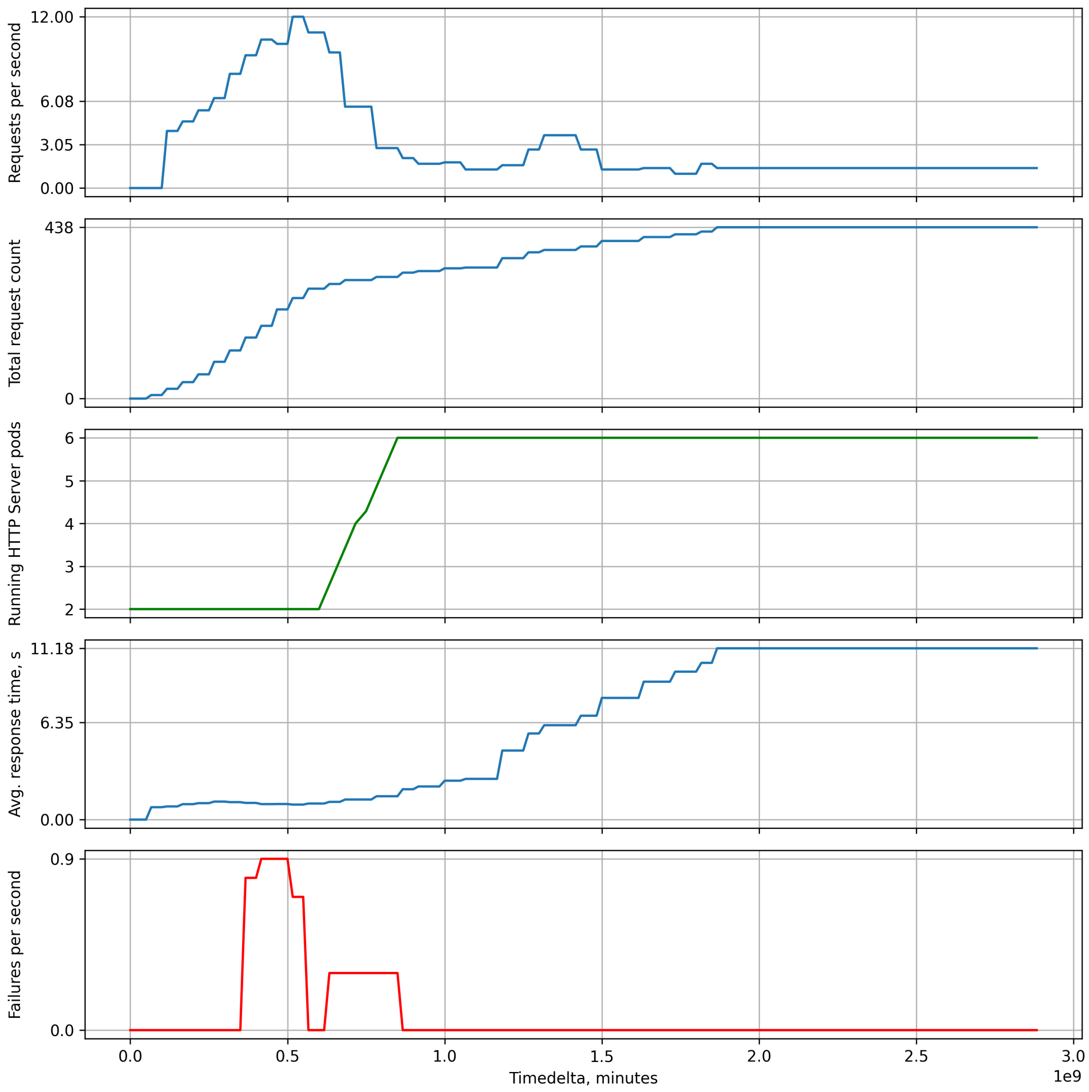


Figure 14. Visualisation of measurements collected during the robustness experiment with 3 worker nodes

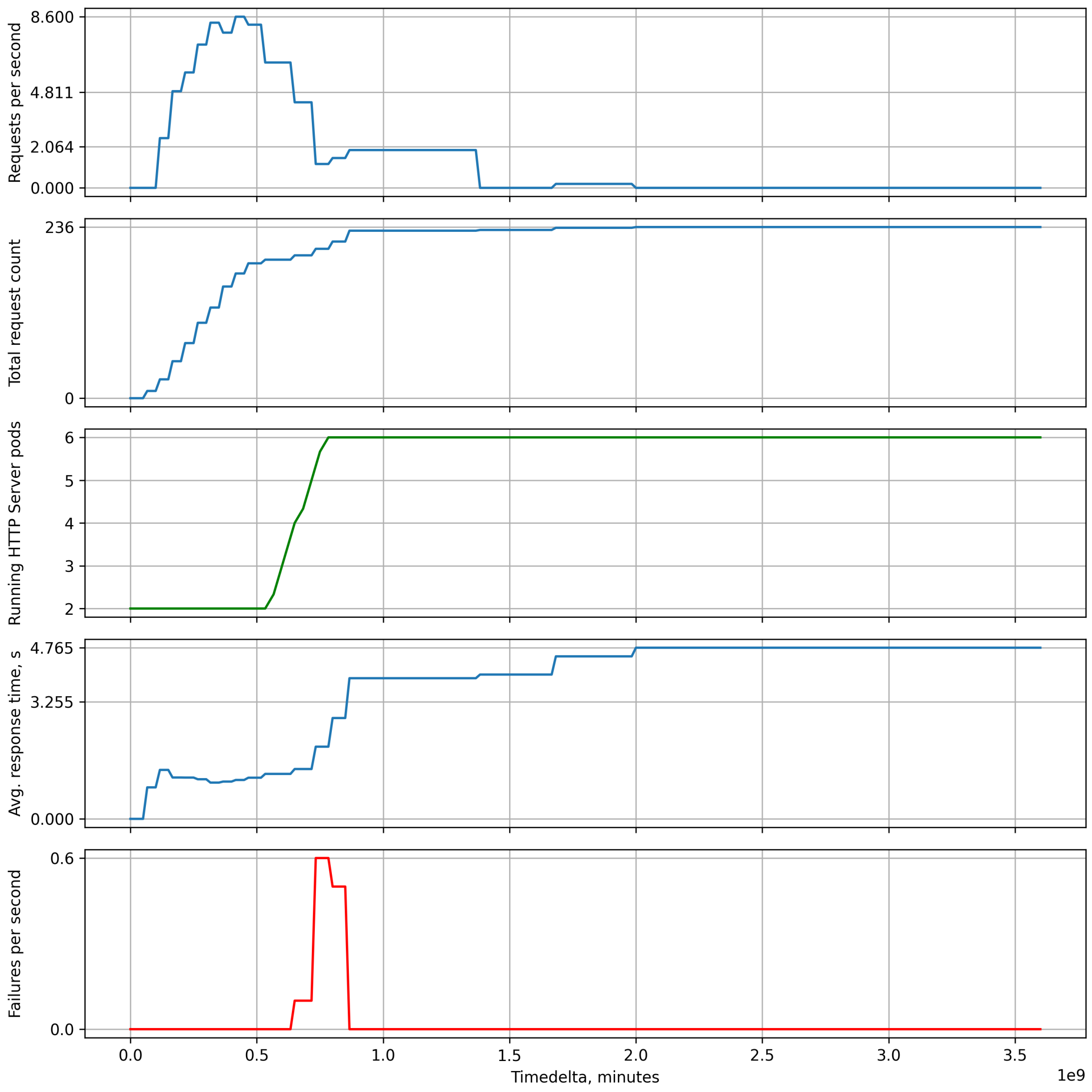


Figure 15. Visualisation of measurements collected during the robustness experiment with 4 worker nodes

5 Discussion

This study aimed to design and implement a system for the scalable discovery of BPS models using Simod by increasing the number of parallel discoveries. The system's architecture with its components and communication was described in Section 3.3. The architecture proposed a distributed microservice-based system tightly integrated into the Kubernetes platform for container orchestration, load balancing, autoscaling, and self-healing. The solution used the job-type workload of Kubernetes to launch BPS model discovery jobs using Simod in a serverless-like manner. It can be compared to how Apache Airflow manages its jobs with the difference that a task is implemented not as a Python script but as a container application that can use any suitable technology.

Scalability is the ability to handle an increased workload with the extended system's capacity while maintaining efficiency [WG06]. Two experiments described in Section 4.1 were executed to validate if the scalability has been achieved. The efficiency degradation depending on the number of worker nodes in the Kubernetes cluster has been shown in Figure 10. From the results, the efficiency is maintained for the lightweight event log, PrepaidTravelCost.xes (7.8 MB). However, the other event log, RequestForPayment.xes (15.2 MB), twice the size, was not handled efficiently by the extended cluster. There is a strong belief that it is related mostly to the testing environment and its limitations.

The testing environment implemented with the *kind* tool uses Docker containers as compute nodes. It means that worker nodes of the cluster are not actual isolated compute nodes but running Docker containers that share the same underlying resources allocated to the Docker runtime. Therefore, the more nodes added to the testing environment, the fewer resources each of them has. Suppose the intensive workload gets distributed to more nodes. In that case, nodes have to compete for resources, leading to the whole Kubernetes cluster becoming unresponsive or to significant latency in computation. Thus, a larger event log leads to more intensive computation, which, when scaled to more nodes in the testing environment, actually leads to worse performance. If this can be confirmed by another experiment in a proper production-ready Kubernetes environment, we can conclude that the decreasing efficiency in Figure 10 for larger logs is attributed to the testing environment rather than to drawbacks in the implementation.

The robustness under load experiment aimed at validating the horizontal autoscaling of the Web API by Kubernetes and ensuring the system can handle peak loads without collapsing. The results of the experiments summarised in Table 6 showed a significant drop in the throughput when moving from the 1-node cluster to 2 and more nodes. The throughput for 2-, 3-, and 4-node clusters remained consistent. The drop in the throughput can be explained by how the load testing was deployed. In the current testing environment, there are two possible ways of making a request to the system: (a) by launching the load testing script on a laptop and port-forwarding requests from the host machine to the *kind*-cluster that runs in the Docker network; (b) by deploying the load testing script alongside the SUT in the Kubernetes cluster. Using port-forwarding is possible, and maybe it can ensure better separation of the load testing framework from the SUT, but in practice, the port-forwarding technique could not handle so many simultaneous requests, and the connection was most of the time broken. This is why the second approach was used in the validation, and the load testing framework was deployed to the same Kubernetes cluster and shared the same resources. Inevitably, it leads to SUT performance degradation when the load testing framework is highly active, which was more of a concern during the robustness experiment. Thus, the drop in the throughput can be explained by the fact that the 1-node cluster deploys the SUT and load testing framework on the same node. Therefore, networking is much more efficient when all systems are side by side. However, when the cluster grows, the load-testing application has to make requests that cross the nodes' boundary using the Docker network to reach the SUT. This is most likely the reason behind such a high number of total requests (885) when using the 1-node cluster compared to multiple-node clusters (236, 438). If we exclude the 1-node cluster from the analysis, then the throughput did not degrade but was maintained at about 1.67 *rps* independent of the number of nodes in the cluster. Together with the improved response time, the robustness experiment can be considered successful if the assumption is correct.

Figures 12 to 15 demonstrated that Kubernetes was able to successfully autoscale the Web API when the load was applied. The current system's configuration had two web servers always running while additional four servers were automatically deployed when necessary. However, as seen from the plots, the scaling happened after the peak when the load decreased already to a moderate level. This is because the Kubernetes Horizontal Pod Autoscaler uses Kubernetes Metrics Server to register the increased load before acting, which is normal but can be improved. The first improvement is decreasing the CPU utilisation threshold, 50% by default, to a lower value so that Kubernetes could react faster. The second improvement is employing a proactive approach using a prediction-based scaling policy [HW21]. However, the prediction-based approach makes sense when the system is operated in the production environment for some time and historical data is available.

One of the study's limitations is the testing environment which is the weakest point and negatively impacts the ability to validate the implementation of the system, forcing it to accept assumptions about resource sharing, computation latency, and the speed of network I/O operations. Setting up and managing a highly available Kubernetes cluster can be a significant challenge on its own. The cluster should be bootstrapped securely, reliably, and cost-effectively. To allow for automated cluster scaling (autoscaling the number of compute nodes), the Cluster Autoscaler[37] component is required. The autoscaler can be integrated into a cloud provider or a custom infrastructure solution depending on the infrastructure. The other way to achieve a production-ready environment would be to use already existing Kubernetes services provided by cloud providers. The current study did not consider it because of the additional costs such a solution incurs.

Another limitation of the study is that it does not consider the system's operational costs. In real settings, one must be careful with automated scaling and choose a scaling strategy that includes the cost of running the system's components. The validation of the scalability and scaling would benefit an organisation more if implemented in a cost-aware manner.

In future, because most of the tasks in process mining can be and usually are implemented as batch processing tools with well-formed input and output, process mining applications can benefit from workflow engines such as Apache Airflow and Luigi. However, Kubernetes, as an orchestration platform, can offer even more. Kubernetes is a more general container-native platform already supported by major cloud providers. It is also technology-agnostic and does not depend on one language or technology. Even though Airflow now provides container support and a user can call non-Python applications through the BashOperator, Airflow has not been initially designed and built for containers. Kubernetes is a platform for highly parallel solutions suitable for a broader range of applications rather than only ETL, which better suits enterprise needs and is most likely to be a part of the enterprise infrastructure.

Argo Workflows[38] is an emerging Kubernetes-native workflow engine similar to Apache Airflow. The Job Service illustrated in Figure 8 uses Kubernetes API to submit Simod jobs and watch for their status updates. So, the current solution may benefit from transitioning this component to Argo Workflows. This would remove the need to develop and support the custom integration with Kubernetes API while providing a useful workflow monitoring UI. If Simod is split into smaller components and services, the whole Simod architecture can be revised and transitioned from a monolith to a modular workflow-based system, better fitting the complex and composite nature of BPS model discovery.

[37]https://github.com/kubernetes/autoscaler
[38]https://github.com/argoproj/argo-workflows/

Conclusion

This thesis presented a complete redesign and re-packaging of the Simod tool to discover BPS models from event logs. The redesign and re-packaging improved the tool with respect to the following dimensions:

1. Ease of use. The extension of Simod, when deployed to a Kubernetes environment, allows users to access Simod through the web service with an HTTP API.
2. Maintainability. Code quality improved from 3.33 to 6.96 out of 10 as measured by PyLint. The system was enhanced with regression tests providing 73% coverage. Automated builds, testing, and distribution were achieved using the CI pipeline.
3. Scalability. The system provided an architecture and implementation using the Kubernetes orchestration technology, increasing the BPS model discovery throughput while ensuring high service availability during peak loads.

The software solution employed the microservice style with tight integration into the Kubernetes orchestration platform through the Kubernetes API. System components were written in different languages and used underlying technologies depending on the task. Web API was written in Python, it used the FastAPI web framework, MongoDB, and the file system for persistent storage. The Go language was used to process the broker's queue concurrently and integrate with the Kubernetes API written originally in Go. Load-balancing, autoscaling, and self-healing were delegated to Kubernetes, while observability was implemented using Prometheus and Grafana.

One of the less emphasised contributions of the study is the refactoring of Simod described in Section 3.2 that was a prerequisite for building the scalable discovery of BPS models using Simod. The refactoring made it possible to modify Simod with more confidence with the help of regression tests and decreased interdependency of Python modules. The test coverage has increased from 0 to 73% while the code quality improved from 3.33 to 6.96 out of 10, according to PyLint. The distribution of Simod has been improved by removing unnecessary hardware and software dependencies and packaging it into a Docker container image.

Simod shows that BPS model discovery is a complex process, thus requiring a composite application that uses multiple underlying tools to execute a series of data processing tasks on the input data. In this case, a workflow-based architecture is a good fit and helps to improve the final solution's maintenance, scalability, robustness, and observability. Containerisation is the technology that introduces isolation of system components providing strict boundaries on one hand while improving the communication across the boundaries on the other hand by making the communication transparent and obvious through the use of API and CLI. Thus, container-based workflow engines make a great fit for complex data processing tasks, such as BPS model discovery.

In future work, Simod, used as a black box in this solution, may benefit from being split into smaller components and services that can be developed, deployed, and scaled independently. The current Simod architecture consists of several processing stages, where some can run in parallel and use the multi-threaded approach. Multi-threading limits parallelism to the number of CPU cores on the machine. This limitation can be overcome if such stages use a cluster of machines instead of only a single computer. Splitting Simod into smaller independent services and components would also benefit the software development process by reducing the system's scope, which must be considered during development. The usage of workflow-based engines may additionally contribute to improved observability, maintenance, and better scalability.

It is critical to continue researching the improvement of architecture solutions for BPS model discovery by utilising modern technologies for distributed cloud computing. The demand for new frameworks will grow as more enterprises transition to the cloud for data storage and processing. Containerisation and orchestration technologies enabled the new generation of workflow engines that can provide low- to no-code solutions for the composition of data processing.

Another direction for future work is to redesign Simod to handle larger datasets. The refactoring of Simod undertaken in this Master's thesis was aimed at making Simod scale out to handle concurrent discoveries. An equally important scalability dimension in this field is to handle larger dataset sizes. To achieve this, there is an opportunity to redesign Simod to use a map-reduce framework, such as Spark, or parallel data frames processing libraries like Dask or Polars.

References

[Aug+19] Adriano Augusto et al. “Split miner: automated discovery of accurate and simple business process models from event logs”. In: *Knowledge and Information Systems* 59.2 (May 2019), pp. 251–284. ISSN: 0219-1377, 0219-3116. DOI: 10.1007/s10115-018-1214-x. URL: http://link.springer.com/10.1007/s10115-018-1214-x (visited on 10/16/2022).

[BF14] P. Bourque and R.E. Fairley. *Guide to the Software Engineering Body of Knowledge, Version 3.0*. IEEE Computer Society, 2014. URL: https://www.computer.org/education/bodies-of-knowledge/software-engineering (visited on 10/16/2022).

[Bur+16] Brendan Burns et al. “Borg, Omega, and Kubernetes: Lessons learned from three container-management systems over a decade”. In: *Queue* 14.1 (Jan. 2016), pp. 70–93. ISSN: 1542-7730, 1542-7749. DOI: 10.1145/2898442.2898444. URL: https://dl.acm.org/doi/10.1145/2898442.2898444 (visited on 04/28/2023).

[Cam21] Manuel Camargo. *Simod: Automated discovery of BPS models. Source code at GitHub*. Version v2.3.1. Mar. 24, 2021. URL: https://github.com/AutomatedProcessImprovement/simod/tree/v2.3.1 (visited on 10/16/2022).

[Cas19] Emiliano Casalicchio. “Container Orchestration: A Survey”. In: *Systems Modeling: Methodologies and Tools*. Ed. by Antonio Puliafito and Kishor S. Trivedi. Series Title: EAI/Springer Innovations in Communication and Computing. Cham: Springer International Publishing, 2019, pp. 221–235. ISBN: 978-3-319-92377-2 978-3-319-92378-9. DOI: 10.1007/978-3-319-92378-9_14. URL: http://link.springer.com/10.1007/978-3-319-92378-9_14 (visited on 04/28/2023).

[CD22] David Chapela-Campa and Marlon Dumas. “Modeling Extraneous Activity Delays in Business Process Simulation”. In: *Proceedings of the 4th International Conference on Process Mining (ICPM 2022)*. Ed. by Andrea Burattin, Artem Polyvyanyy, and Barbara Weber. IEEE, 2022, pp. 72–79. DOI: 10.1109/ICPM57379.2022.9980544. URL: https://doi.org/10.1109/ICPM57379.2022.9980544.

[CDA20] Long Cheng, Boudewijn F. van Dongen, and Wil M.P. van der Aalst. “Scalable Discovery of Hybrid Process Models in a Cloud Computing Environment”. In: *IEEE Transactions on Services Computing* 13.2 (Mar. 1, 2020), pp. 368–380. ISSN: 1939-1374, 2372-0204. DOI: 10.1109/

TSC.2019.2906203. URL: https://ieeexplore.ieee.org/document/8669858/ (visited on 11/24/2022).

[CDG20] Manuel Camargo, Marlon Dumas, and Oscar González-Rojas. "Automated discovery of business process simulation models from event logs". In: *Decision Support Systems* 134 (July 2020), p. 113284. ISSN: 01679236. DOI: 10.1016/j.dss.2020.113284. URL: https://linkinghub.elsevier.com/retrieve/pii/S0167923620300397 (visited on 10/16/2022).

[Cha+23] David Chapela-Campa et al. "Can I Trust My Simulation Model? Measuring the Quality of Business Process Simulation Models". In: *CoRR* abs/2303.17463 (2023). DOI: 10.48550/arXiv.2303.17463. arXiv: 2303.17463. URL: https://doi.org/10.48550/arXiv.2303.17463.

[Dai12] Robert Daigneau. *Service design patterns: fundamental design solutions for SOAP/WSDL and RESTful Web services*. Addison-Wesley signature series. OCLC: ocn753468449. Upper Saddle River, NJ: Addison-Wesley, 2012. 321 pp. ISBN: 978-0-321-54420-9.

[Don20a] Boudewijn van Dongen. *BPI Challenge 2020: Prepaid Travel Costs*. In collab. with Department Of Mathematics {And} Computer Science Eindhoven University Of Technology. Medium: media types: application/x-gzip, text/plain, text/xml Version Number: 1 Type: dataset. Mar. 26, 2020. DOI: 10.4121/UUID:5D2FE5E1-F91F-4A3B-AD9B-9E4126870165. URL: https://data.4tu.nl/articles/_/12696722/1 (visited on 04/10/2023).

[Don20b] Boudewijn van Dongen. *BPI Challenge 2020: Request For Payment*. In collab. with Department Of Mathematics {And} Computer Science Eindhoven University Of Technology. Medium: media types: application/x-gzip, text/plain, text/xml Version Number: 1 Type: dataset. Mar. 26, 2020. DOI: 10.4121/UUID:895B26FB-6F25-46EB-9E48-0DCA26FCD030. URL: https://data.4tu.nl/articles/_/12706886/1 (visited on 04/19/2023).

[Est+21] Bedilia Estrada-Torres et al. "Discovering business process simulation models in the presence of multitasking and availability constraints". In: *Data Knowl. Eng.* 134 (2021), p. 101897. DOI: 10.1016/j.datak.2021.101897. URL: https://doi.org/10.1016/j.datak.2021.101897.

[Eve16] Joerg Evermann. "Scalable Process Discovery Using Map-Reduce". In: *IEEE Transactions on Services Computing* 9.3 (May 1, 2016), pp. 469–481. ISSN: 1939-1374. DOI: 10.1109/TSC.2014.2367525. URL: http://ieeexplore.ieee.org/document/6948229/ (visited on 11/24/2022).

[HW21] Tengfei Hu and Yannian Wang. “A Kubernetes Autoscaler Based on Pod Replicas Prediction”. In: *2021 Asia-Pacific Conference on Communications Technology and Computer Science (ACCTCS)*. 2021 Asia-Pacific Conference on Communications Technology and Computer Science (ACCTCS). Jan. 2021, pp. 238–241. DOI: 10.1109/ACCTCS52002.2021.00053.

[Las+22] Katsiaryna Lashkevich et al. *Why am I Waiting? Data-Driven Analysis of Waiting Times in Business Processes*. Dec. 2, 2022. DOI: 10.48550/arXiv.2212.01392. arXiv: 2212.01392[cs]. URL: http://arxiv.org/abs/2212.01392 (visited on 04/27/2023).

[LHD23] Orlenys López-Pintado, Iryna Halenok, and Marlon Dumas. “Prosimos: Discovering and Simulating Business Processes with Differentiated Resources”. In: *Enterprise Design, Operations, and Computing. EDOC 2022 Workshops*. Ed. by Tiago Prince Sales et al. Lecture Notes in Business Information Processing. Cham: Springer International Publishing, 2023, pp. 346–352. ISBN: 978-3-031-26886-1. DOI: 10.1007/978-3-031-26886-1_23.

[Lin16] David S. Linthicum. “Practical Use of Microservices in Moving Workloads to the Cloud”. In: *IEEE Cloud Computing* 3.5 (Sept. 2016). Conference Name: IEEE Cloud Computing, pp. 6–9. ISSN: 2325-6095. DOI: 10.1109/MCC.2016.114.

[Mur89] T. Murata. “Petri nets: Properties, analysis and applications”. In: *Proceedings of the IEEE* 77.4 (Apr. 1989), pp. 541–580. ISSN: 00189219. DOI: 10.1109/5.24143. URL: http://ieeexplore.ieee.org/document/24143/ (visited on 11/25/2022).

[Obj14] Object Management Group. *Business Process Model and Notation (BPMN), Version 2.0*. Version 2.0.2. 2014. URL: https://www.omg.org/spec/BPMN/2.0.2/PDF.

[Rei+17] Daniel Reißner et al. “Scalable Conformance Checking of Business Processes”. In: *On the Move to Meaningful Internet Systems. OTM 2017 Conferences*. Ed. by Hervé Panetto et al. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2017, pp. 607–627. ISBN: 978-3-319-69462-7. DOI: 10.1007/978-3-319-69462-7_38.

[Sil16] Alan Sill. “The Design and Architecture of Microservices”. In: *IEEE Cloud Computing* 3.5 (Sept. 2016). Conference Name: IEEE Cloud Computing, pp. 76–80. ISSN: 2325-6095. DOI: 10.1109/MCC.2016.111.

[Van16] Wil Van Der Aalst. *Process Mining*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016. ISBN: 978-3-662-49850-7 978-3-662-49851-4. DOI: 10.1007/978-3-662-49851-4. URL: http://link.springer.com/10.1007/978-3-662-49851-4 (visited on 04/27/2023).

[Ver+15] Abhishek Verma et al. "Large-scale cluster management at Google with Borg". In: *Proceedings of the Tenth European Conference on Computer Systems*. EuroSys '15: Tenth EuroSys Conference 2015. Bordeaux France: ACM, Apr. 17, 2015, pp. 1–17. ISBN: 978-1-4503-3238-5. DOI: 10.1145/2741948.2741964. URL: https://dl.acm.org/doi/10.1145/2741948.2741964 (visited on 05/08/2023).

[WG06] Charles B. Weinstock and John B. Goodenough. *On System Scalability*. Technical Note CMU/SEI-2006-TN-012. 2006. URL: https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=7887 (visited on 04/20/2023).

[WKK15] Jean-François Weets, Manish Kumar Kakhani, and Anil Kumar. "Limitations and challenges of HDFS and MapReduce". In: *2015 International Conference on Green Computing and Internet of Things (ICGCIoT)*. 2015 International Conference on Green Computing and Internet of Things (ICGCIoT). Oct. 2015, pp. 545–549. DOI: 10.1109/ICGCIoT.2015.7380524.

Appendix

I. Implementation of the Solution

The implementation of the solution consists of several repositories, with the meta-repository located at https://github.com/automatedProcessImprovement/simod-on-containers. The following code repositories contain corresponding implementations:

simod v3.3.0 Simod repository[39] contains the version of Simod used in the thesis with the Docker image publicly available at the Docker Hub[40].

simod-http v0.11.1 Web API service repository[41] contains the implementation of the HTTP Server written in Python using the FastAPI web framework. The Docker image is available at the Docker Hub with the corresponding tag[42].

simod-job-controller-go v0.6.0 Job Service repository[43] contains the implementation of the Job Controller with the Docker image available at the Docker Hub[44].

simod-load-testing v0.2.1 Load testing solution repository[45] that uses Locust for validation experiments. The Docker image is available at the Docker Hub[46].

simod-on-containers The meta-repository[47] that contains links to other repository and, most importantly, contains deployment manifests for Kubernetes.

[39]https://github.com/AutomatedProcessImprovement/simod/tree/v3.3.0

[40]https://hub.docker.com/layers/nokal/simod/3.3.0/images/sha256-74c3ab0bdac31b385a780168ca71f1eae3162631abd04e288cedc7bd6d7135a6?context=explore

[41]https://github.com/AutomatedProcessImprovement/simod-http/tree/0.11.1

[42]https://hub.docker.com/layers/nokal/simod-http/0.11.1/images/sha256-325603e551bf29fd2488d25d2bbeff77e7e2b37dcaa01bbd5849402fe929927e?context=explore

[43]https://github.com/AutomatedProcessImprovement/simod-job-controller-go/tree/0.6.0

[44]https://hub.docker.com/layers/nokal/simod-job-controller-go/0.6.0/images/sha256-bf77343dbbcf01838cbdac7598542796ae9778731be5336a5e25516cef117683?context=explore

[45]https://github.com/AutomatedProcessImprovement/simod-load-testing/tree/0.2.1

[46]https://hub.docker.com/layers/nokal/simod-load-testing/0.2.1/images/sha256-aef3f8ce9d44e16dd753f518ff5b1fead4faca2f336deab77d0557f29df83849?context=explore

[47]https://github.com/automatedProcessImprovement/simod-on-containers

II. Kubernetes Manifest for Web API

The full Kubernetes manifest in YAML that is used to deploy the Web API service to the testing environment in this thesis is presented below.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: simod-http
spec:
  selector:
    matchLabels:
      app: simod-http
  template:
    metadata:
      labels:
        app: simod-http
    spec:
      volumes:
        - name: simod-data
          persistentVolumeClaim:
            claimName: simod-volume-claim
      containers:
        - name: simod-http
          image: nokal/simod-http:0.11.1
          command: [ "/bin/bash" ]
          args: [ "run_uvicorn.sh" ]
          env:
            - name: BROKER_URL
              value: "amqp://guest:guest@rabbitmq-service:5672"
            - name: SIMOD_EXCHANGE_NAME
              value: "simod"
            - name: SIMOD_PENDING_ROUTING_KEY
              value: "requests.status.pending"
            - name: SIMOD_HTTP_HOST
              value: "0.0.0.0"
            - name: SIMOD_HTTP_PORT
              value: "8000"
            - name: SIMOD_HTTP_STORAGE_PATH
              value: "/tmp/simod-volume/data"
            - name: SIMOD_HTTP_LOG_LEVEL
              value: "info"
            - name: SIMOD_GUNICORN_WORKERS
              value: "4"
            - name: MONGO_URL
              value: "mongodb://mongodb:27017"
            - name: MONGO_DATABASE
              value: "simod"
            - name: MONGO_USERNAME
```

```
              value: "root"
            - name: MONGO_PASSWORD
              value: "example"
            - name: MONGO_REQUESTS_COLLECTION
              value: "requests"
          resources:
            requests:
              cpu: 256m
              memory: 256Mi
          volumeMounts:
            - name: simod-data
              mountPath: /tmp/simod-volume
---
apiVersion: v1
kind: Service
metadata:
  name: simod-http
  labels:
    app: simod-http
spec:
  selector:
    app: simod-http
  ports:
    - name: web
      port: 8000
---
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: simod-http
spec:
  maxReplicas: 6
  minReplicas: 2
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: simod-http
```

III. Kubernetes Manifest for Job Service

The full Kubernetes manifest in YAML that is used for deployment of the Job Service to the testing environment in the thesis is presented below.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: simod-job-controller
spec:
  selector:
    matchLabels:
      app: simod-job-controller
  template:
    metadata:
      labels:
        app: simod-job-controller
    spec:
      volumes:
        - name: simod-data
          persistentVolumeClaim:
            claimName: simod-volume-claim
      containers:
        - name: simod-job-controller
          image: nokal/simod-job-controller-go:0.6.0
          env:
            - name: BROKER_URL
              value: amqp://guest:guest@rabbitmq-service:5672
            - name: SIMOD_EXCHANGE_NAME
              value: simod
            - name: SIMOD_DOCKER_IMAGE
              value: nokal/simod:3.3.0
            - name: SIMOD_JOB_RESOURCE_CPU_REQUEST
              value: "1"
            - name: SIMOD_JOB_RESOURCE_MEMORY_REQUEST
              value: "4Gi"
            - name: SIMOD_HTTP_HOST
              value: simod-http
            - name: SIMOD_HTTP_PORT
              value: "8000"
            - name: KUBERNETES_NAMESPACE
              value: default
          resources:
            limits:
              cpu: "1"
              memory: 1Gi
            requests:
              cpu: 100m
              memory: 128Mi
          volumeMounts:
            - name: simod-data
              mountPath: /tmp/simod-volume
---
apiVersion: v1
kind: Service
metadata:
  name: simod-job-controller
  labels:
    app: simod-job-controller
spec:
  selector:
    app: simod-job-controller
  ports:
  - name: web
    port: 8080
---
apiVersion: monitoring.coreos.com/v1
kind: ServiceMonitor
metadata:
  name: simod-job-controller
spec:
  selector:
    matchLabels:
      app: simod-job-controller
  endpoints:
  - port: web
```

IV. Simod Configuration

Simod configuration for business process simulation model discovery is presented below. The log path is dynamically adjusted depending on the provided event log by the Web API.

```
version: 2
common:
  log_path: /Users/ihar/Projects/PIX/simod-load-testing/assets/
      PrepaidTravelCost.xes
  repetitions: 1
  evaluation_metrics:
    - dl
    - absolute_hourly_emd
    - cycle_time_emd
    - circadian_emd
preprocessing:
  multitasking: false
structure:
  max_evaluations: 1
  mining_algorithm: sm3
  optimization_metric: dl
  concurrency:
    - 0.0
    - 1.0
  epsilon:
    - 0.0
    - 1.0
  eta:
    - 0.0
    - 1.0
  gateway_probabilities:
    - equiprobable
    - discovery
  replace_or_joins:
    - true
    - false
  prioritize_parallelism:
    - true
    - false
calendars:
  max_evaluations: 1
  optimization_metric: absolute_hourly_emd
  resource_profiles:
    discovery_type: differentiated
    granularity: 60
    confidence: 0.1
    support: 0.7
    participation: 0.4
```

```
extraneous_activity_delays:
  num_iterations: 1
  optimization_metric: relative_emd
```

V. Licence