technische universität
dortmund

Bachelor's Thesis

Automata Learning at Scale: Evaluation of Query Parallelization Strategies in the Context of Clustered Systems

Julien Saevecke

November 10, 2022

Reviewer:
Prof. Dr. Bernhard Steffen
M.Sc. Julius Alexander Bainczyk

TU Dortmund University
Department of Computer Science
Chair of Programming Systems (Chair 5)
http://ls5-www.cs.tu-dortmund.de

Contents

Chapter 1

Introduction

Today, computer software is used in many items around us. It starts with the obvious computer that helps us in our daily lives, moves to important medical devices that preserve or save lives, and ends with self-driving cars for transporting goods or people. Regrettably, even the simplest software is prone to bugs that manifest themselves in different ways. A bug in software can manifest itself in a simple button on a website not working. But it can also lead to catastrophic financial loss or, even worse, loss of life. [65] Therefore, it is important to prevent software failures or bugs that can negatively impact or even endanger lives. Therefore, between 30 and 60 percent of the software development process consists of testing. [?]

An effective approach to prevent or reduce software defects and increase software quality is Model-Based Testing (MBT). In this approach, a model, such as a finite-state machine, is used to represent the desired behavior of a system under test (SUT). The model can be used to create test cases to verify that the implementation of the SUT is working correctly. In MBT, the creation of the model can be time-consuming and costly if done manually, and prone to errors, especially for complex real systems.

Active automata learning is a technique for deriving a behavioral model that represents the actual behavior by running test queries on the SUT (also called System Under Learning (SUL) in this context). A popular framework for setting up active automata learning is LearnLib [17]. However, this technique has two major problems, which are even more emphasized in the context of real systems.

The first major problem is that active automata learning is a rather expensive technique, since obtaining sufficient knowledge to derive a behavioral model of the target system involves a significant number of test queries sent to the SUL. Depending on the complexity of the system, answering such a test query can cause high latency and take a lot of time. Consequently, learning behavioral models of such a system can take hours, days, or even months. [55]

The second problem is that active automata learning requires the independence of test queries in order to learn correctly. This means that no test query should influence another test query. Therefore, the requirement for the target system is to provide a reset function or some other abstraction that guarantees that no query affects another. However, in the context of real-world systems, this requirement is often not met, since the system may, for example, consist of multiple services and persistent storage solutions to maintain the system's state. [45]

Despite the progress in reducing the total learning time and enabling the learning of real systems, learning with active automata is still a heavyweight process. Therefore, this thesis proposes and evaluates an architecture that enables scalable active automata learning with adaptations to real systems.

1.1 Related Work

Performing active automata learning is a heavy weighted process, especially in the context of real-life systems. A learning process may take a long time to learn a behavioral model of the target system. Therefore, often not feasible to utilize in MBT in, e.g., agile or time-constrained environments. Thus, existing studies examined various methods to reduce the total execution time of the active automata learning process, such as: optimizing equivalence queries, using learn algorithms [56, 55], reusing states(removing duplicated work) [29, 36], and abstraction layers (e.g., using equivalence classes) [24].

In [36], active automata learning is evaluated in a context combining multi-threading and reusing states and is compared to the usually single-threaded approach taken in the research mentioned above. This approach runs multiple SULs simultaneously as threads on a single machine. It divides the Membership Query (MQ)s, which are sent in the learning stage, and Equivalence Query (EQ)s before being sent into numerous batches and distributed to the running SULs to balance the load.

Additionally, [45] examined active automata learning with a primary focus on real-life applications, regarding the challenges of learning them and which steps can be taken to make learning such systems more practicable. This work follows the same approach as [36] by parallelizing and distributing queries to multiple SULs on a distributed system and compares different learning algorithms regarding the query batch size affecting the total execution time in this context.

However, no in-depth evaluation was made using active automata learning on real-life applications performed on a distributed system. Thus, this work tries to close this gap by tackling the challenges of active automata learning regarding real-life applications on distributed systems and evaluating such a setup's performance and practicability.

1.2 Goals

The approach in this research complements the scalable aspect of the method from [36] and shifts the context, similar to the cloud computing approach [45], from the active automata learning process running on a single physical machine to a clustered system. A clustered system consists of a group of at least two physical and virtual machines interconnected in such a way that they can behave as a single system. Running active automata learning on a clustered system by parallelizing and distributing the processing of queries across multiple SUL instances can significantly improve the execution time to learn a behavioral model and is not limited by the resources of a single machine.

However, a clustered system requires some management overhead to achieve this. Through the use of virtualization technologies and orchestration tools, it can provision and manage resources on demand and thus adapt to ever-changing workloads by scaling resources up and down on the fly and being resilient. Therefore, this work has the following main objectives:

1. Build a scalable architecture by extending LearnLib and utilizing a clustered system.
2. Evaluation of methods for automatic scaling.

Consequently, this research proposes an architecture to perform scalable active automata learning. The utilization of a clustered system and technologies enables the concurrent processing of test requests distributed across multiple SULs. In addition, strategies for automatic scaling at runtime are used to efficiently utilize available cluster resources and compared to an approach without scaling. Target SULs are matched to real systems by delaying the processing of queries and restarting them to ensure the independence of test queries. The result is an architecture that can learn all types of SULs without providing additional features such as reset mechanisms. Furthermore, this approach is evaluated in terms of practicality, scalability, and constraints, with a focus on real systems. This addresses the research gap mentioned above.

1.3 Structure

This work is structured as follows. In Chap. 2 the basics of active automata learning and the process of learning a behavioral model of a SUL are discussed. In addition, Mealy Machines and the LearnLib tool are described in this context. Finally, to complete the fundamentals, the technologies used to set up active automata learning in a clustered system are explained.

In Chap. 3, the proposed architecture of the clustered automata learning system is described. In addition, in Chap. 4 explains how the experiments are set up, run, and measured for later evaluation.

In Chap. 5.1 the results of the performed learning experiments are presented, analyzed, and evaluated. This part provides information about the practicality, limitations, and performance of the proposed scalable architecture.

Finally, in Chap. 6 concludes this research by determining whether the goals stated in Sec. 1.2 were achieved. It also presents ideas for further research that focuses on possible improvements or extensions to the proposed architecture, leading to further reductions in overall learning time or enabling the architecture to learn real-world systems.

Chapter 2

Preliminaries

Before explaining how proposed architecture can be implemented to utilize the underlying clustered system to perform scaleable active automata learning, it is essential to introduce the fundamentals of the technologies used to make that happen. In the same breath, the tool LearnLib is examined regarding how its structure and features can help to move the active automata learning process to a clustered system.

2.1 Mealy Machines

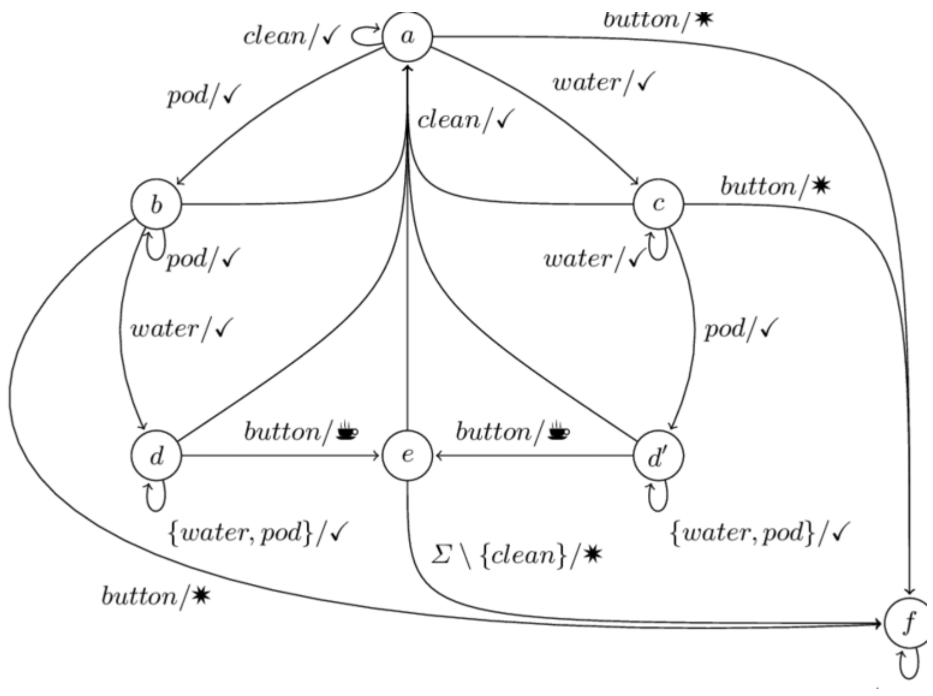

Figure 2.1: A coffee machine as a Mealy machine [59]

LearnLib offers various ways to implement and learn finite deterministic automatons, e.g., Mealy machines. [40] In this work, it is crucial to understand how Mealy machines work because the behavior of the SUL used to evaluate the architecture is represented as Mealy machines.

Mealy machines are a variant of a finite-state machine, where an output alphabet accompanies the input alphabet. The output is determined by the current input and the current state of the Mealy machine. Mealy machines differentiate themselves from other finite-state machines by having a transition function defined for all input symbols resulting in the output to a sequence of inputs being deterministic. A state diagram can represent Mealy machines.

2.1.1 Definition. The formal definition of a Mealy machine is a six-tuple $M = (S, s_0, \Sigma, \Omega, \delta, \lambda)$ where

- S is a finite nonempty set of states
- $s_0 \in S$ is the initial state,
- Σ is a finite input alphabet,
- Ω is a finite output alphabet,
- $\delta : S \times \Sigma \rightarrow S$ is the transition function, and
- $\lambda : S \times \Sigma \rightarrow \Omega$ is the output function.

The example Mealy machine state diagram shown in Fig. 2.1 represents a coffee machine. This state diagram defines a coffee machine with four different interactions: fill in water (water), put in a coffee pod (pod), start coffee making process (button), and clean the machine (clean). These interactions can occur in arbitrary order and based on the sequence of interactions.

The following responses are possible: valid interaction ($\checkmark$), invalid interaction ($\star$), and making coffee (☕). Nevertheless, it is required to have enough water and a coffee pod in the coffee machine before pressing the button to make a cup of coffee successfully (e.g., $\{clean, pod, water, coffee\} =$ ☕). All other sequences in-between either lead to an invalid action ($\star$) or are missing further interactions ($\checkmark$) to make a cup of coffee.

The behavior of the coffee machine explained above is learned with various configurations to evaluate the proposed architecture of this work. More details on the specifications of the coffee machine are covered in Sec. 4.1.

2.2 Active Automata Learning

Active Automata Learning is a process that automatically infers formal behavioral models of, e.g., black box systems through testing [60]. Furthermore, inferred behavioral models enrich the development of systems by using, e.g., quality assurance.

The indicated process adheres to the Minimally Adequate Teacher (MAT) model and is comparable to the analogy of a relationship between a learner and a teacher (further called SUL). In the context of active automata learning, the learner has to figure out an unknown formal language L known by the SUL by requesting answers from the SUL on specific questions (further called queries) [49, 45]:

1. A Membership query (MQ) is a sequence $w \in \Sigma^*$, whereas Σ is given out by the teacher. Based on w, that gets processed by the SUL, leading to an answer. The learner uses this answer to construct a hypothesis model of L

2. An Equivalence query is used to compare the current constructed hypothesis model of the learner with the actual behavioral model of L known by the SUL. This comparison may result in inequality, and a counterexample is provided, proofing the dissimilarity

Using these types of queries, the learner follows an alternating two-stage strategy to determine the L. At first, the MQs are used to construct the hypothesis model of the unknown L, followed by testing the current hypothesis model through EQs may result in a counterexample. A refined hypothesis model can be constructed using the counterexample, which may result in new MQs and EQs. As stated, these two stages take turns until EQs no longer produce counterexamples. At this point, the hypothesis model is equal to the target L.

However, this work focuses on real-life systems, usually black-box systems that introduce some challenges. Firstly, the hypothesis model can not be directly compared to the target model of the SUL; thus, the EQs have to be approximated. Consequently, this may result in a higher MQ and EQ count than learning a non-black-box system. [61]

Secondly, MQs need to be independent of each other - meaning the SUL in question has to be resettable to its initial state before each MQ. This requirement can be problematic because real-life systems often rely on an internal state stored in, e.g., databases and may contain multiple services. Such systems may not offer functionality to reset the whole system to its initial state without restarting it. [39]

At last, simple input alphabets often do not suffice to communicate with real-life systems; thus, a so-called ‘mapper‘ is used, representing an intermediate layer between the learner and the SUL, which translates MQs, e.g., to method invocations. [58, 38] The first two challenges are tackled in this work.

There is a choice to be made when it comes to libraries enabling automata learning adhering to the explained MAT model: LearnLib [17] or libalf [19]. This work uses LearnLib as the library of choice. The reason is the faster performance compared to the other available choices and its flexible modular structure. [40, 18]

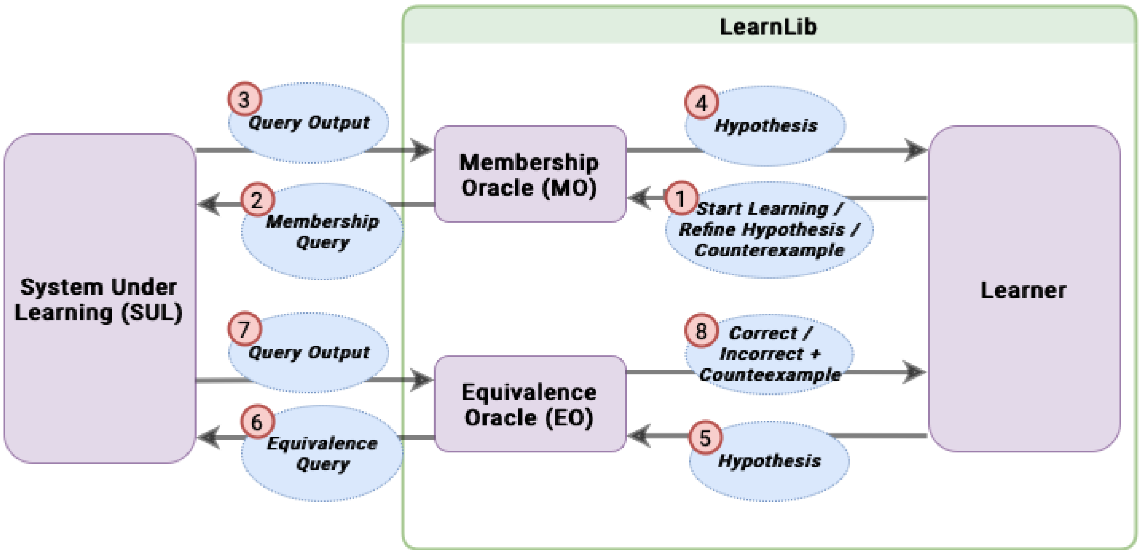

Figure 2.2: Learn process with LearnLib based on [36]

2.3 LearnLib

LearnLib is an open-source framework built in Java [13] specializing in active automata learning. It provides a modular design enabling flexible and extensible learning configurations and allows for capitalization of specific properties of the target SUL. [49] Additionally, the design structure accommodates using different infrastructures, e.g., distributed systems while maintaining a good performance [40, 45, 36].

Further, it provides various learning algorithms, equivalence approximation strategies, and filters such as caches to reduce the total number of queries. [45] Thus, LearnLib was used to learn models of various implementations. Some of these implementations are: TCP [41], SSH [62], bank cards [23] and printer software [55].

A more in-depth active automata learning process, implemented with LearnLib and adhering to the MAT model, is illustrated in Fig. 2.2. It can be separated into two parts. The first part is the target system that should be learned, namely the SUL. The second is the learning process implemented in LearnLib, which consists of the following components:

- The Membership Oracle (MO) is a simple interface to a SUL, acting as an in-between layer - handling the communication between the SUL and the learner. It sends the MQs generated by the learning algorithm of the learner to the SUL. It sends back the responses to the learner, therefore helping in creating/refining the hypothesis model.

- The Equivalence Oracle produces the already mentioned EQs. The structure is the same as the MQs. The difference is that these queries are sent to the target SUL by the Equivalence Oracle to check the validity of the constructed hypothesis model. A chosen algorithm provided by LearnLib might approximate these queries in the case of black-box systems

To conclude, moving the active automata learning process into the context of a clustered system is possible due to the extensible framework of LearnLib. Thus, LearnLib are utilized in this work to implement, e.g., a custom MO to adhere to the unique requirements of a clustered system (explained in Sec. 3.1).

2.4 Virtualization Technologies

Virtualization generally is a technology that partitions physical system resources to create an abstraction of one or many isolated environments. [51, 33] Virtualization is used to make resource utilization more efficient and provide more flexibility. [30] This technology can be differentiated into multiple types of virtualization. The most used techniques are hypervisor-based and Operating System (OS)-level virtualization. [26, 27]

Virtualization on the hardware level is accomplished through a hypervisor implemented directly on top of the host machine's hardware. The hypervisor is the intermediate software layer that can launch multiple completely isolated guest OS. One such system is referred to as a Virtual Machine (VM) or as full virtualization. Each VM gets its resources through the hypervisor, acting as they run directly on the hardware. [51, 33] This technique brings an overhead because each VM runs its OS and file system. This overhead is why a VM is slow to launch [27, 50] and should be used in scenarios where complete isolation and security, ease of management, and customization are the primary features in need. [52, 33]

The OS-level virtualization technique is also known as Container-based operating system virtualization [57] or containerization [64]. Instead of virtualizing the underlying hardware of the host machine with an intermediate software layer like a hypervisor, the container-based virtualization approach virtualizes the OS kernel without a need for an additional software layer by creating multiple isolated user-space environments by dividing the physical resources of the host machine usually through [66] the use of namespaces [31]. Generally, the resources that each environment has access to are managed by either limiting/prioritizing through control groups [44]. Resulting in an environment in which running processes, the required binaries, and their dependencies are isolated from all other running processes on the host machine and its file system. This isolated environment is referred to as a container. [34, 64] By sharing the same host OS kernel and containers holding only the required binaries and libraries to run, they are considered lightweight compared to VMs because they are likely to utilize less memory and disk space. Allowing running more containers than VMs on the same physical host machine. [25, 30]

The following existing container-based virtualization tools can be used to turn applications into containers but are not limited to LXC [5], Singularity [6], and Docker [4]. For this work, the tool Docker is used to bundle the SUL into a container because it is the industry standard to date and is accompanied by a large community. [20] Docker uses so-called images (created through Dockerfiles) containing step-by-step instructions to create an image instance - namely, a container.

Thus, container-based virtualization technology is the foundation to enable scale-able SULs on-demand based on the workload at any time. Accordingly, moving the active automata learning process to a clustered system is feasible. Still, it is required that the varying simultaneously running number of SUL containers must be managed.

2.5 Container Orchestration

Container-based virtualization tools provide the ability to bundle and run applications as self-contained, lightweight containers. Still, they do not offer capabilities to manage a high number of containers on a clustered system. Manual management of containers on a clustered system is challenging as the container number grows, especially across clusters with one or more physical- or virtual machines - further called nodes.

Thus, container orchestration tools provide a framework to manage containers at scale by utilizing the resources the underlying clustered system provides. Therefore, providing features to simplify the operational effort of deploying, managing the life-cycle, scaling, and resources of containers. Consequently, the range of primary features regarding containers of such orchestration tools are, e.g., resource management, scheduling, fault tolerance, and auto-scaling.[32, 43, 48]

In the context of this work, the auto-scaling feature is essential to perform efficient active automata learning at scale. Auto-scaling regarding containers enables the creation and termination of containers automatically without restarting the whole system or the learning process in the context of automata learning. It is implemented via policies based on configurable thresholds. These thresholds are based on metrics such as the current utilization of Central Processing Unit (CPU) and memory of the containers.

Some orchestration tools allow for configuring the auto-scaling behavior or extension of the available metrics through third-party auto-scalers or custom auto-scaling policies. The benefits of auto-scaling are but are not limited to efficient resource utilization leading to reduced cost of the cluster and more available resources and meeting the demand at any time when facing a varying workload. Besides offering auto-scaling for containers, it is in some cases possible to use this feature to scale nodes to extend the resources of the cluster, as well.

Many popular container orchestration tools can be utilized to manage containers, such as Kubernetes (K8S) [9], Docker Swarm [8], and Apache Mesos [7]. Therefore, to manage the architecture of the clustered system, the orchestration tool K8S is used due to its maturity and stability and provides the most features. [63] Due to being open-source and enabling customization, it is actively developed by a large community. [20] K8S manages the clustered system (further called K8S cluster) based on the concept of maintaining the desired state, which is described through applying, e.g., YAML files, defining K8S resources such as deployments.

Chapter 3

Automata Learning of Clustered Systems

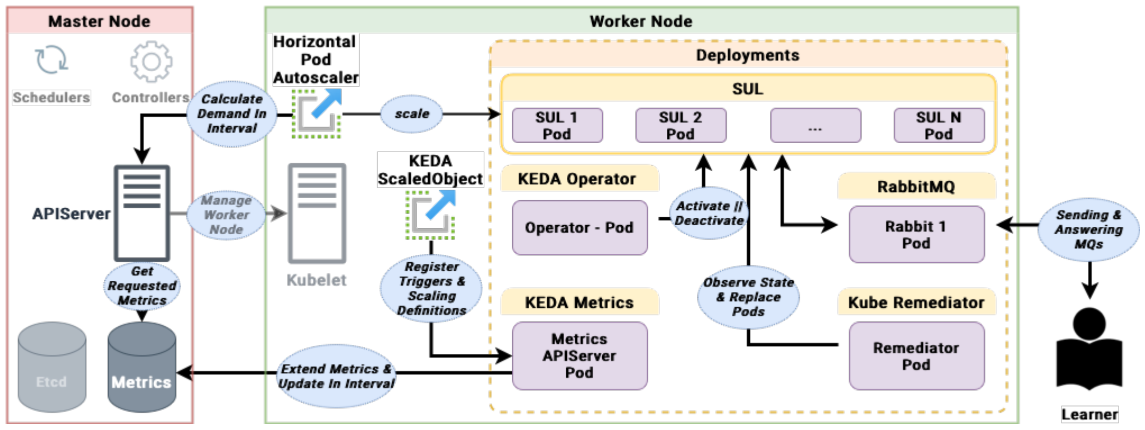

Figure 3.1: Simplified overview of the architecture

The architecture proposed in this work, on which scalable active automata learning processes are executed, is shown in Fig. 3.1. As mentioned in the previous chapter, the underlying cluster is managed by K8S and uses Docker as the container engine to manage and execute its containers. Therefore, this architecture requires the SUL to be containerized (as a Docker image) for learning a behavioral model of the SUL.

However, more considerations need to be made before a viable solution for scalable learning of active automata is available. The deployed SUL containers and the learner must communicate with each other and the MQs distributed among the various SULs. For this purpose, a message broker is used as an intermediary - namely RabbitMQ.

Furthermore, auto-scaling strategies are defined and performed through the use of the K8S Horizontal Pod Autoscaler (HPA) and the third-party auto-scaler Kubernetes Event-driven Autoscaling (KEDA) to leverage the message broker metrics. Finally, another third-

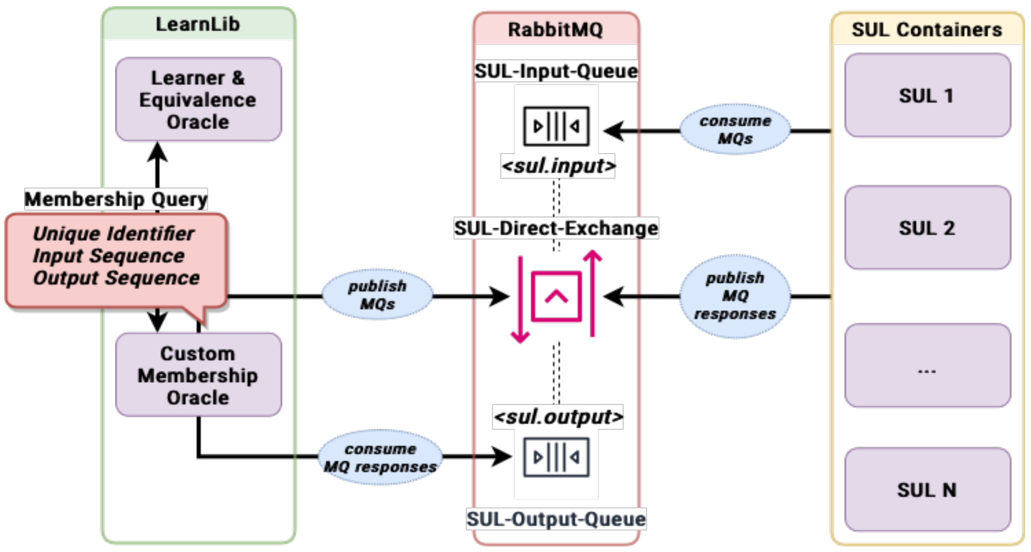

Figure 3.2: Learner and SUL communication with RabbitMQ

party application (called Kube Remediator) is used to enable the use of disposable SULs on K8S, which by default does not support auto-scaling of short-lived containers.

3.1 Membership Query Distribution

Due to automatic scaling, the number of active SUL instances is expected to vary at any time. Therefore, an intermediary that is aware of each SUL is required in order to handle the communication - the exchange of MQs - between the two participants. Moreover, the learning algorithm used in this work poses the queries in batches. Consequently, the communication must be handled in a way that ensures that each SUL has a turn - i.e., load balancing. Also, MQ distribution should be asynchronous to reduce additional overhead, e.g., waiting for a SUL to receive a request. Therefore, a message broker is used to bridge the communication gap.

A message broker is a means of connecting and scaling a modular architecture in a distributed system by enabling communication between the parts of the architecture in question - in this case, the SULs and the learner - via so-called 'message queues'. In addition, a message broker provides a common infrastructure for systems that produce and consume messages. Typically, it is based on the publish/subscribe paradigm. [53]

In the context of automata learning, it is crucial that such a message broker is reliable, stable, and does not incur noticeable overhead when exchanging MQs from the learner to the SULs and back. For this reason, the RabbitMQ [21] message broker is a suitable choice to meet the above requirements. [42]

RabbitMQ is an efficient and scalable open-source message broker that serves as a communication platform between independent applications and uses an Erlang-based Advanced Message Queuing Protocol (AMQP) [2] that enables asynchronous processing of data. The RabbitMQ infrastructure consists of messages, message queues, exchanges, and route bindings. [35]

At the core of RabbitMQ is the concept of a message that can contain arbitrary data. An application that sends a message through RabbitMQ is called a Publisher, while the recipient of the message is called a Consumer. An application can be both a Publisher and a Consumer. In the case of this work, both the learner and the SUL act simultaneously as publishers and consumers of messages. In practice, this may not be the case.

A message always passes through an exchange, which acts as a router and accepts the message, and forwards it to the correct message queue. The exchange knows the correct message queue based on the routing key contained in the message and the route binding that connects the message queue to the exchange. Thus, messages are always accompanied by a routing key. Applications subscribed to a message queue containing messages can consume and respond to them.

Since communication during the learning process between the learner and the SULs goes through RabbitMQ, it is important to capture the existing RabbitMQ communication infrastructure (shown in Fig. 3.2). Despite the ability to create complex topologies with RabbitMQ, the infrastructure used is as simple as RabbitMQ can get. At its core, it is a direct exchange ('*SUL-Direct-Exchange*') through which MQs are forwarded to message queues. On the one hand, the learner publishes unprocessed MQs along with the routing key '*sul.input*', which are distributed to the message queue '*SUL-Input-Queue*' from which SULs retrieve MQs for processing. On the other hand, SULs publish processed MQs with the routing key '*sul.output*' to the message queue '*SUL-Output-Queue*' from which the learner consumes and constructs its hypothesis of SUL.

```
Input: queries ← list of unprocessed queries, sentQueries ← empty hashmap
Output: queries
 1: for all query ∈ queries do
 2:     uid ← generate random unique identifier
 3:     mq ← new MembershipQuery(uid, query)
 4:     add pair (uid, query) to sentQueries
 5:     publish mq to 'SUL-Direct-Exchange' and routing key 'sul.input'
 6: end for
 7: while sentQueries is not empty do
 8:     response ← consume message from 'SUL-Output-Queue'
 9:     query ← get and remove query with key response.uid from sentQueries
10:     query.output ← response.output
11: end while
```

Algorithm 3.1: Simplified RabbitMQ Membership Oracle

Even though the infrastructure is simple, further customization is required. A custom MO must be implemented and used by the learner to generate and consume MQs RabbitMQ messages, as in Alg. 3.1. The algorithm consists of two parts: publishing unprocessed queries to RabbitMQ (lines 1-6) and consuming SUL responses to the published queries (lines 7-11).

The custom MO accepts a list of unprocessed queries. Each unprocessed query is extended with a generated Unique Identifier (UID) (lines 1-3); therefore, an additional UID field is required in the MQ data structure. Due to the asynchronous nature of RabbitMQ, it is not guaranteed that the same order in which the learner publishes the MQs is maintained when the learner consumes the responses to the published MQs. Therefore, a hash map is used to store each query by its UID before it is published (lines 4-5). This makes each published query unique, allowing each response to be matched to the correct query (lines 9-10). The MO processes batch by batch. Consequently, no new queries are published to RabbitMQ until each query in the current batch has been answered.

3.2 Challenges of Real-Life Applications

In general, performing active automata learning to learn a target system, especially in real-world applications, is fraught with challenges. In section 2.2, some of them were mentioned. One challenge, in particular, is the requirement to guarantee the independence of each query, which is addressed in a unique way in this research.

The first way is to execute a reset procedure on the target system. This functionality is usually not available in real systems. Second, the only requirement is that no query affects the other, which is possible through an abstraction, e.g., using independent sessions in web applications. Thus, no direct reset function is required. [45]

However, sometimes direct reset or reset by abstraction is not feasible. Therefore, in this work, all experiments are performed with some kind of SUL container that terminates after a query is processed. As K8S attempts to maintain the desired cluster state, the terminated SUL container is restarted. Therefore, for each published MQ, a SUL container is created to process it. This type of SUL is further called ,disposable' SUL.

Nevertheless, this approach introduces further challenges when a learning process is performed in terms of auto-scaling and how K8S attempts to maintain the desired cluster state. These challenges are discussed in detail in the 3.3 and 3.4 sections.

3.3 Orchestration of Containerized SULs

Containers are not run directly on the cluster itself because K8S encapsulates containers in its basic unit, a pod. A pod can contain one or more containers. In this work, a pod is limited to one container; therefore, a pod is the same as an SUL container. In addition, a pod contains memory resources, a network IP, and other configurations for running the container(s). The requested and limited compute resources (CPU and memory) can be specified at creation.

As mentioned earlier, scheduling is one of the main tasks of a master node. The default K8S scheduler uses the information (e.g., memory and CPU limit) about each pod to make pod placement decisions and ensure that no single node in the K8S cluster exceeds its resource capacity. Consequently, pods that exceed their memory limit are terminated, but in contrast, briefly exceeding the CPU limit does not cause the pod to terminate immediately.

The scheduler is challenged by the unique circumstances created by the use of disposable SULs. It is expected that many SULs will need to be replaced at the same time. This places a heavy load on the scheduler and may result in additional overhead or unexpected behavior.

Looking at the scheduler, it is also important to know how K8S manages the container lifecycle. This will help to understand the challenges of using disposable SULs (in 3.2) and autoscaling (in 3.4). Each container in a pod can be in one of the following states:

- Waiting: Still executing instructions to complete the start of the container.
- Running: The container is running without any problems.
- Finished: The container has run to the end, failed, or is forced to exit.

In addition, it is possible to execute instructions before a certain state is reached by lifecycle hooks. For example, the 'preStop' hook can be configured and executed before a container enters the terminated state.

To conclude container lifecycle management, it is important to note that each container is subject to a restart policy with limited options. The restart policy determines when the containers in the pod should be restarted. It takes effect when the pod fails, succeeds, or is in an undefined state and therefore terminates. The Pod 'restartPolicy' field is configurable and can take one of the following values: 'Always', ,OnFailure', or 'Never' (default: 'Always'). When the 'restartPolicy' takes effect, the containers enter a restart loop with an exponential back-off delay with an upper limit of five minutes. This delay is reset once a container has been running for 10 minutes without issue. [14]

For managing pod instances, K8S provides various resources such as deployments and jobs. The proposed architecture uses deployments to manage SUL pods while learning. Deployments define long-running tasks that are supported by the K8S auto-scale feature. It also creates a replica set, which in turn creates the desired initial number of replicated pods (namely replicas). An image can be specified that will be used to create the container in each replica. The deployment maintains the desired number of pods, which may be set or adjusted by auto-scaling. In addition, CPU and memory requirements and limits can be set for each pod in a deployment.

It is vital to know that a pod in a deployment only supports the restart policy with the value set to 'Always' - this value is not configurable. This default policy causes pods to constantly restart and enter a restart loop when the pod fails or runs to completion. The restart loop ensures that a pod is available at all times. However, the exponential back-off delay, which is also not configurable, results in significant downtime between responding to queries with disposable SULs that run to completion after processing a query. Consequently, each SUL continuously enters the restart loop after processing a query. However, this mechanism does not replace the failed pod. Instead, it restarts the container(s) within the pod, so no heavily weighted processes are triggered by this process. Nevertheless, the exponential back-off delay in restarting is a challenge to overcome.

K8S does not provide an alternative to a deployment that allows automatic scaling for the particular circumstances of this work. It is still possible to use disposable SULs while taking advantage of a K8S deployment. The following options can achieve this:

1. Implement a custom K8S control plane that allows, for example, configuring the restart policy in deployments and replaces the default K8S control plane.

2. Instead of using K8S resources such as deployments to orchestrate containers, implement custom pod management logic that is deployed as a pod and allows creation, termination, and scaling by using the K8S API

3. Observe the state of each pod as it is deployed and replace it when it reaches a certain state.

For this work, the decision falls on the third option, since such an application already exists. To this end, the Kube Remediator application is used to enable the use of disposable SULs, therefore, significantly reducing downtime by replacing each pod when it reaches its restart loop. While this approach enables the use of disposable SULs, the process of replacing pods involves terminating an old pod and creating a new one. Therefore, this process may result in extra work.

In summary, K8S provides many features and options for orchestrating containerized applications designed for long and short-running tasks. This work presents special requirements that do not fit into the K8S design. However, using third-party tools, it is possible to work around the challenges presented and effectively perform active automata learning on a K8S cluster utilitzing disposable SULs.

3.4 Strategies for Scaling SULs

In a scenario where auto-scaling is required, K8S provides two resources for auto-scaling pods: HPA and Vertical Pod Autoscaler (VPA). Although both auto-scaler types do their

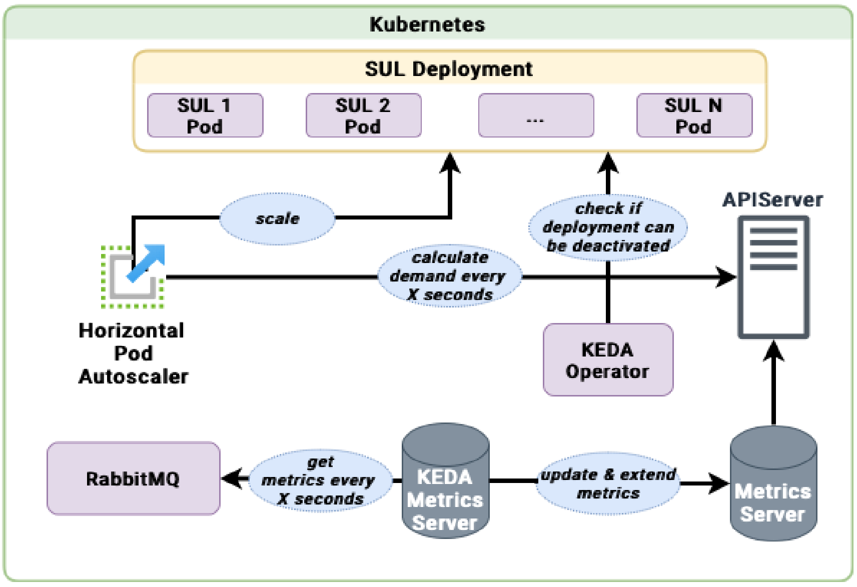

Figure 3.3: Simplified overview of the KEDA auto-scaling process

job without manual intervention, they differ in the entity on which the scaling is performed. The VPA scales the computational resources that each pod can access based on resource analysis over time. Unlike the VPA, the HPA adjusts the desired number of replicas based on the utilization of the compute resources allocated to them. [28]

In the context of this work, the VPA does not lead to any significant performance gain due to the low resource requirements that the SULs used have by simply representing a Mealy machine. Since the goal is to process as many MQs as possible simultaneously, it is necessary to scale the number of SUL pods up and down - i.e., to use a HPA. HPA subsection Horizontal Pod Autoscaler The HPA scales a K8S target resource, such as a deployment, by adjusting the desired number of replicas in the target resource; thus, the HPA does not work directly with the pod replicas. In the deployment example, replica count maintenance is done by the deployment's replica controller, as shown in Fig. 3.3. By default, only two basic computational resources are available as metrics for automatic replica count scaling: CPU and memory. [47] HPA scales based on the average utilization of the selected metric across all replicas and triggers a replica count update using the following equation [10]:

$$desiredReplicas = \lceil currentReplicas * \frac{currentMetricValue}{desiredMetricValue} \rceil$$

. In addition, when using default HPA, it is not possible to scale the number of replicas to zero and disable the deployment to save resources. A simplified overview of the autoscaling process used in this work, including where the aggregated metrics are sourced, is shown in Fig. 3.3.

HPA updates its metrics values every 15 seconds (default value). These metrics are aggregated via the metrics server, which retrieves its information from each node in the K8S cluster. By default, aggregation occurs every 60 seconds. Consequently, given the default settings, in the worst case there may be a 45-second time lag between the determination that the target should be scaled up and the scaling itself. As a result, HPA misses work peaks in the range of seconds, which can occur every now and then. In addition, the calculation is based on the average workload of the target metric aggregated across all pods, which misses workload spikes that occur every second. [54]

However, as mentioned earlier, HPA only provides two basic pod metrics for calculating the current demand for pod replicas. Neither option is suitable for guaranteeing the availability of a SUL for each MQ sent by the learner. Consequently, each SUL replica processes only one MQ at a time, resulting in uniform resource consumption for each replica. A more appropriate auto-scaling strategy is to create one MQ replica for each MQ in a given RabbitMQ queue to process that MQ, at any time. Consequently, RabbitMQ-specific metrics must be added to the available metrics for the HPA. As a result, a third-party solution, namely KEDA [15], is used to provide automatic scaling for the chosen event-driven architecture: RabbitMQ.

3.4.1 Kubernetes Event Driven Autoscaling

KEDA is an extensible, lightweight component. It can be attached to an K8S cluster without interfering with or replacing K8S's own autoscaling resources such as HPA. Therefore, it is a safe option that can be used on any K8S cluster by coexisting with other auto-scaling options. It behaves more like an extension of the standard HPA and works with it under the hood.

KEDA provides a variety of auto-scalers for event-driven infrastructures that use message brokers (e.g. RabbitMQ) or similar tools. This enables auto-scaling of any K8S resource based on the exposed metrics of the particular event-driven infrastructure. In the case of RabbitMQ, the following metrics (called triggers in KEDA) are available for auto-scaling:

- Message Rate: Triggers auto-scaling based on the published messages to a specific queue in a second (publish rate)
- Queue Length: Triggers auto-scaling on a set number of messages in the specified queue

A KEDA configuration file is shown in Fig. 3.4 (more fields and details in [15]). This configuration file is used in the learning configurations (Sec. 4.5) and illustrates how to configure a KEDA ScaledObject and its available fields. In this case, the initial number of SULs in the setup is zero (line 11). The RabbitMQ metric is updated every second. The trigger (lines 13-25) determines the metric - 'QueueLength' (line 20) - and the threshold (line 23) when to auto-scale. In this case, the desired SUL count is compared to the message

Figure 3.4: KEDA ScaledObject configuration file

```
 1  spec:
 2    scaleTargetRef:
 3      apiVersion: apps/v1
 4      # K8S resource type of the target to scale
 5      kind: Deployment
 6      # The name refers to the target K8S resource
 7      name: sul
 8    # Interval to update metrics in seconds
 9    pollingInterval:  1
10    # Min. replica count to be maintained
11    minReplicaCount:  0
12    # List of auto-scale triggers
13    triggers:
14    # Type of auto-scaler
15    - type: rabbitmq
16      metadata:
17        # Protocol used to scrape the metrics
18        protocol: amqp
19        # Possible metrics: QueueLength or MessageRate
20        mode: QueueLength
21        # Threshold on which to trigger
22        # Message count or Publish/sec. rate
23        value: "1"
24        # Target RabbitMQ message queue
25        queueName: sul_input_q
```

count of the RabbitMQ queue '*sul_input_q*' (line 25) at the time the HPA calculates the new demand. Lines 1-7 specify the name of the target (line 7) and the K8S resource type (line 5) to be scaled automatically.

In addition, a ScaledObject provides the ability to change the HPA behavior by using extended fields in the configuration file. The up and down scaling behavior can be set independently. There is also a difference in the way scaling up and scaling down work. When scaling down, a stabilization window (default: 300 seconds) is taken into account, where the desired state is derived by selecting the highest value in a list of previously calculated desired states in the specified interval, e.g. in the last 5 minutes. This window prevents the number of replicas from fluctuating greatly and, in particular, removing pods only to create the same number of pods the next moment. In contrast, when scaling up, there is no stabilization window by default. [12] Finally, the limit on the number of pods terminated or created by scaling is also configurable.

In this work, automatic scaling, especially scaling down, only works by ensuring that an SUL that has consumed an MQ is terminated only after it has finished its query. This prevents the loss of queries. The following options are available:

1. Re-queueing the MQ before terminating the SUL

2. Removing the MQ from the queue only if it got processed by a SUL

The second approach is taken in this work. While this approach prevents the loss of MQs, it does not solve the problem of the SUL terminating before the MQ is processed. As a result, scenarios can occur where MQs are passed from SUL to SUL, hindering the learning process.

K8S provides functions that are configurable in the deployment file that solve this problem. As mentioned earlier, it is possible to configure a 'preStop' lifecycle hook. It will be executed before the container is terminated. Through this hook, it is possible to delay termination until the SUL in question has finished processing the MQ. However, this approach assumes that the container provides a shell environment and possibly read/write access (based on the hook's implementation) to the K8S cluster on which it is running. The native image created with GraalVM does not provide a shell by default. Therefore, this approach is not suitable.

In this work, the following approach is taken. Pods that are about to terminate go through a termination grace period that allows for a soft termination. This means that K8S allows the pod to, for example, terminate the RabbitMQ connection or let it finish processing and run to completion before being forcibly terminated by K8S. The grace period is 30 seconds by default. However, it must be adjusted if applications are used that perform heavy processing tasks exceeding this limit.

In summary, automatic scaling of active learning automata can be achieved with the design decisions made in this work using the dedicated event-driven auto-scaler KEDA. Therefore, providing one SUL replica for each MQ in a given RabbitMQ queue may result in more efficient use of computational resources than the approach of using a constant number of SULs. In contrast, the auto-scaling strategy could be slower in terms of overall execution time. Moreover, auto-scaling involves additional management overhead in terms of the container lifecycle, depending on the learning configuration.

Chapter 4

Evaluation Method

The architecture described in Chap. 3 and shown in Fig. 3.1, has been set up and is used to perform the active automata learning setups presented in this chapter. The architecture is built as a single-node cluster using MicroK8s [16] and is equipped with the following hardware - 126 GB RAM and an AMD EPYC™ 7443P CPU with 24 cores @ 2.85 GHZ with 48 threads.

4.1 Experiment: Coffee Machine

The evaluation of the proposed architecture in terms of practicality, performance, automatic scaling and constraints is done by executing active automata learning processes. In Sec. 2.1, the Mealy machine, which is a coffee machine, was introduced. This Mealy machine is containerized and used as a disposable SUL in the different learning configurations to evaluate the architecture.

It is important to note that the SUL, and thus the coffee machine application itself, consumes the queries from the RabbitMQ queue. In practice, using the proposed architecture with any type of SUL would typically require an additional mapper between RabbitMQ to handle the communication.

As mentioned in Sec. 2.3, LearnLib provides various algorithms and filters for the learning process. The learner used in the learning setups performs the learning process using the following LearnLib components:

- Query cache that reduces the total sent queries by filtering out queries that were already sent
- The Equivalence Oracle in use simulates the behavioral model of the SUL; thus, no approximation of EQs takes place. Therefore, reducing the total amount of sent queries significantly by directly comparing the hypothesis to the target model to generate counterexamples.

Table 4.1: Statistics of learning a coffee machine

Data	**Value**
Number Of States	5
Number Of Inputs	4
Number Of Outputs	3
Number Of Queries Sent	118
Min. Batch Size	1
Max. Batch Size	4
Avg. Batch Size	2
Min. Sequence Length	1
Max. Sequence Length	6
Avg. Sequence Length	3

- LearnLib offers many learning algorithms. In this work, the Direct Hypothesis Construction (DHC) learning algorithm is used because it supports query batches, enabling query distribution for simultaneous processing. Moreover, the primary reason is that the DHC algorithm favors big batch sizes compared to the other learning algorithms available in LearnLib, such as $L*$. [45]

As a result, Tab. 4.1 illustrated characteristics of the coffee machine as well as statistics of the process learning it.

4.2 SUL Containerization

In order to orchestrate SUL instances with K8S, the SUL application must be turned into a container on which to base experiments. Although creating a container with a tool like Docker is not complicated, the overhead involved can vary significantly depending on the binary and dependencies required to run SUL. Reducing overhead is critical in this work, as many containers are likely to be terminated and new ones created over time when using auto-scaling and disposable SULs. Consequently, the underlying basis of the SUL container determines the impact on:

- Processing time of a MQ
- Scale-ability of the SUL, determined by:
  - Creation time
  - Start-up time
  - Termination time
- Deployment time of the SUL

Thus, it can have a significant impact on the overall execution time and feasibility of a learning setup. The coffee machine container is based on an image containing the Java runtime library [13]. In addition to using LearnLib to implement the coffee machine behavior

Table 4.2: Image size variance: coffee machine

Base Image	Compressed Size
openjdk:11	334.91 MB
openjdk:11-jdk-slim	241.63 MB
adoptopenjdk/openjdk11:alpine-jre	67.74 MB
native-image graalvm	26.02 MB

Table 4.3: Start-up time based on image: coffee machine

Base Image	Average Start-Up Time
adoptopenjdk/openjdk11:alpine-jre	3.18s
native-image graalvm	0.1s

of SUL, the application uses Spring [22] as an intermediary between the coffee machine and RabbitMQ. These details can have a significant impact on image size, application start-up time, and thus the overall execution time of a learning process.

The Docker container layer that wraps the application itself adds an imperceptible overhead of about $\sim 0.036s$ to the container creation time, according to the author of [37]. Docker image size is an issue when using Docker image registries, from which the image is pulled to create containers. A smaller image size results in faster loading speed from the relevant registries and thus faster container creation. Tab. 4.2 illustrates the variance that a coffee machine SUL image-size can have when using different base images required to run the coffee machine and its dependencies.

Finally, the start-up time of the SUL itself determines the time it takes for the SUL to be ready to accept MQs. As shown in Tab. 4.3, the time it takes to start the SUL coffee machine itself can fluctuate and vary greatly depending on how it is implemented, how the executable is created, and how the Docker image is created. In this work, a significant reduction in start-up time and image size was achieved by creating a native image with GraalVM [11] using Spring Native [3]. This reduces the overall overhead caused by containerization and the SUL application itself. The native image is thus used to create Coffee Machine SUL containers in a learning process.

4.3 Simulating Real-Life Systems

The processing time of an MQ with respect to the Mealy machine implementation is almost instantaneous, unlike real-world applications that can take significantly longer, even seconds to minutes. Therefore, adjustments are made in the learning processes to match real-world systems and, more importantly, to produce measurable effects.

Adaptation to real systems is an artificial delay to the SUL processing time. The artificial delay is calculated based on the sequence length of the MQ to be processed. For each symbol in the sequence, a random time interval is accumulated to give calculate the processing delay before the MQ is processed. The time interval used in the calculation is configurable, and various values are used in this work. See Sec. 4.5 for more details on the values used. As introduced in Sec. 3.2, the use of disposable SULs accommodates real systems that do not provide functionality or take a long time to reset, thus satisfying the MQ independence requirement.

Together, the two approaches simulate complex systems that may include many services and persistent storage solutions that take longer to process queries and do not provide the functionality to reset the entire system. Therefore, the scaling approach in this work may be used in practice when learning systems utilizing, for example, databases to store system states.

4.4 Data Acquisition

In this work, for each execution of a learn setup (explained in Sec. 4.5), data about the SUL and the learning process is collected. Besides the already mentioned details about the coffee machine mentioned in Tab. 4.1, additional data is acquired for evaluation:

- Learn setup used
- Processing time (min, max, avg) of each SUL
- Start-up time (min, max, avg) of each SUL
- Interval between each response the learner gets (min, max, avg)
- Artificial delay accumulated (min, max, avg) over each input in a MQ sequence of each SUL before the query is processed
- The total execution time it took to learn the model

In addition, the learner calculates and accumulates the data and receives SUL-specific data as an appendix to the MQ response. At the end of each learning process, the data is averaged. It also ensures that the executed learning process sends the same queries in the same order for each learning configuration.

4.5 Learning Setups

The proposed architecture allows the configuration of various parameters of a learning setup, such as:

1. Configuration of the SUL deployment including:
    - The SUL image that should be used for auto-scaling

    - Minimum and maximum SUL instances, which may be overwritten by KEDA when using auto-scaling
    - Deployment-related configuration, which may be unique to the SUL requirements, e.g., termination grace period or a pre-stop hook

2. If the deployment is to scale automatically, a KEDA ScaledObject must be configured including:

    - Minimum and maximum SUL instances
    - The event-driven infrastructure used. In this case, RabbitMQ is used, and the auto-scale triggers:
        - Metrics: 'QueueLength' or 'MessageRate'
        - Target queue
        - Trigger threshold value determining when to auto-scale
        - Polling interval of the metrics and cool-down period for disabling the deployment when no new message arrived in the target queue
        - For more fine-grained control over the auto-scaling process, the HPA can be configured based on the requirements of the learning setup

In addition, the termination grace period for the completion of each learning setup is set to 10 seconds and 60 seconds for more extensive delays. These values are sufficient for the learning configurations in this work and guarantee that each MQ is processed by the SUL before it is forced to terminate by K8S. Furthermore, before executing the learning process of a learning configuration, it is ensured that the configured initial number of SULs is deployed and ready to receive queries.

In conclusion, learning setups can be divided into two categories. The first category of learning setups uses a constant number of disposable SULs. The purpose of this category is to highlight the scalability and practicality of the chosen approach. The second category uses automatic scaling with KEDA based on the length of the SUL input queue. With the goal of having one SUL instance ready for each request in the input queue.

Chapter 5

Evaluation Results

5.1 Architecture Overhead

Table 5.1: Architecture overhead

Overhead In Seconds	**1 SUL**	**8 SULs**
Minimum	0.007	-0.08
Average	9.95	1.74
Maximum	24.18	12.62

In this chapter, the term ‘performance‘ is used to describe the total learning time of a learning process; consequently, the term ‘performance gain‘ refers to a reduction in learning time compared to the baseline in question. However, before we address the performance gains of the different strategies, it is important to know the total overhead incurred by the proposed architecture. Tab. 5.1 shows the minimum, average, and maximum overhead in seconds added to the time it takes for a SUL to respond to a request. Observing a single SUL for the entire learning process yields the following results:

1. The minimum overhead is in the nanosecond range, similar to the start-up time of the coffee machine application itself, as shown in Tab. 4.1. The reason for this is the behavior of the restart loop enforced by K8S; the first restart is instantaneous. The restart loop does not terminate the pod in question, so there is no need to create a new pod. Instead, the container in the pod is restarted and so is the coffee machine application itself. This results in a fast start-up and no overhead at all.

2. There is overhead because containers need to be terminated and restarted. As mentioned in Sec. 3.3, the Kube Remediator is used to terminate pods that reach the first stage of exponential back-off in the restart loop. As a result, K8S maintains the desired SUL count by restarting a new pod instead in place of the terminated

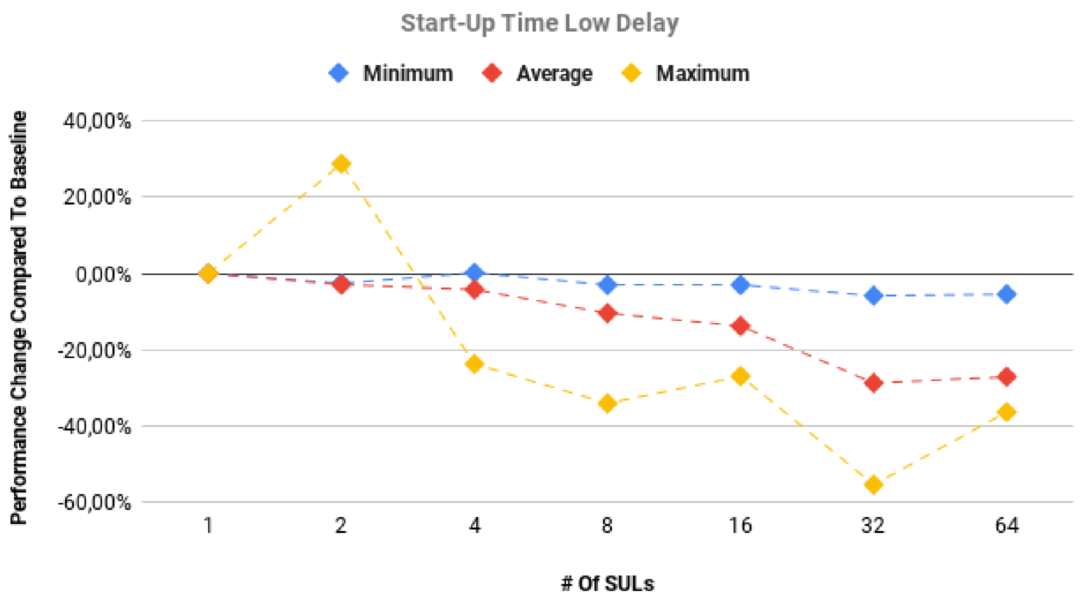

Figure 5.1: Start-up time based on SUL count with low processing delay

pod. Consequently, terminating and creating a pod introduces an average overhead of about 10 seconds until a new SUL is ready to receive a query.

3. The maximum overhead of almost 25 seconds is due to K8S sometimes failing to create a pod for various reasons. Therefore, the pod is terminated and recreated, resulting in significant overhead.

However, the introduced overhead can be compensated by using more, e.g., eight SULs, as shown in Tab. 5.1. It is worth noting that more SULs do not reduce the architecture overhead that affects each SUL. Instead, the increasing number shortens the time interval until a SUL is ready to respond to a request, thereby mitigating the overhead.

Another finding is that the overhead of K8S itself increases as more SULs are used in a learning process. This overhead is more pronounced when using a learning configuration with little to no processing delay (Fig. 5.1) than when using a configuration with a high processing delay (Fig. 5.2). This is because the administrative overhead of scheduling, reserving resources, creating, and terminating pods is much higher in low-delay learning configurations due to the frequency with which pods are replaced. This results in an average slower start-up time of about 45% for 64 SULs. In this case, the K8S cluster begins to throttle the resources that the SULs have access to. As a result, more resources are available to maintain the desired cluster state. The large outliers in the use of 32 and 64 SULs in both graphs may be due to K8S using many resources for maintenance, scheduling, and termination, and creating many SULs at once.

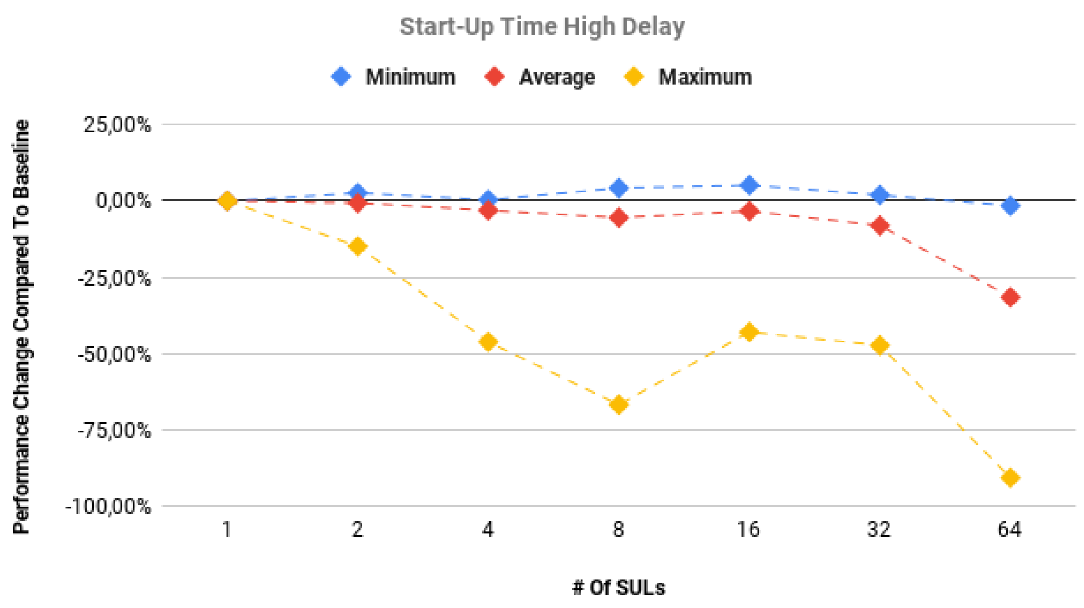

Figure 5.2: Start-up time based on SUL count with high processing delay

5.2 Strategy 1a: Constant SUL Count Without Delay

The strategy of using a constant number of SULs is executed with the following SUL numbers: 1, 2, 4, 8, 16, 32 and 64. The baseline of this strategy is the use of a single permanent SUL. The total time to learn the coffee machine using the baseline is on average $\sim 1,186.5$ seconds (~ 20 minutes). Apart from the overhead introduced by the proposed architecture, parallelization is still very effective in improving the overall learning time when using disposable SULs, as shown in Fig. 5.3. In this case study, using the coffee machine and 64 SUL instances significantly improves performance by 3,450.25% compared to the baseline, resulting in a total learning time of ~ 33.4 seconds on average.

Even using only two SULs leads to a faster performance of 95.90%. Thus, the results suggest that learning target systems that respond quickly to queries benefit significantly by increasing the number of SULs. However, the performance gain decreases significantly with each additional SUL. At the point where eight SULs are used, doubling the SUL count results in a performance increase of about 5% on average, as Fig. 5.4 shows. In the case of the coffee machine, the graph suggests that the optimal cost-benefit ratio is about six to eight simultaneously active SULs, resulting in an average performance gain of about $\sim 425.2\%$ in learning systems that start quickly and respond quickly.

Furthermore, the data suggest that an optimal cost-benefit ratio can be achieved by using a number of SULs equal to the maximum batch size of the experiment. In the case of the coffee machine, the maximum batch size is four. However, more SULs are needed to compensate for the overhead caused by the architecture. The number of SULs that

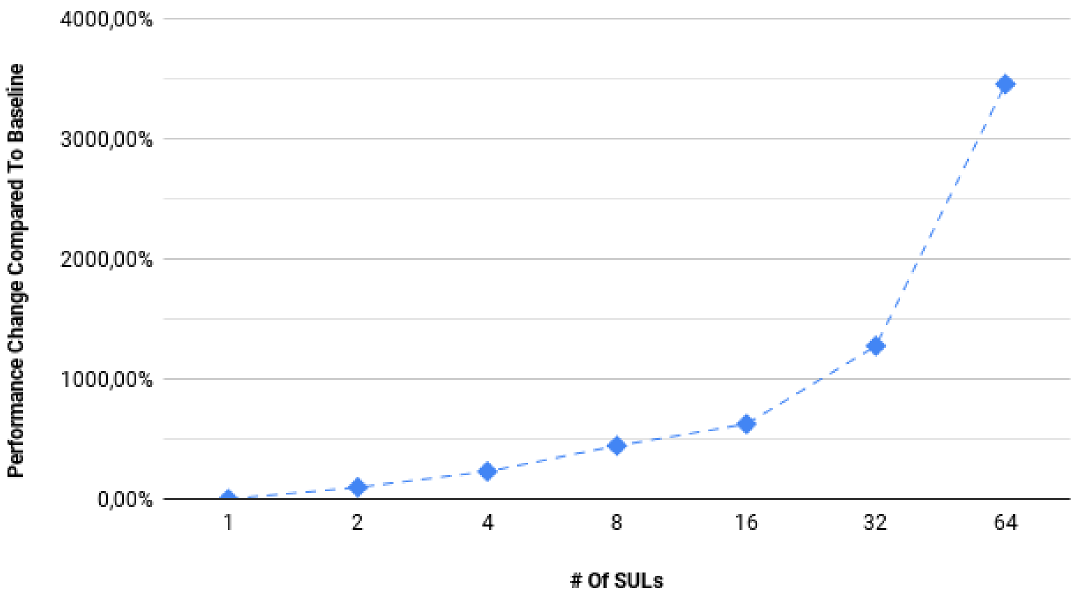

Figure 5.3: Performance gain without processing delay

must be used to achieve an optimal cost-benefit ratio can therefore be approximated by the following equation, where N is the number of SULs that mitigate the architecture overhead:

$$\#SULs = MaximumBatchSize + N \tag{5.1}$$

5.3 Strategy 1b: Constant SUL Count With Delay

As mentioned in Sec. 4.3, an artificial delay is added to the processing time of each SUL. For each input in a sequence of a MQ, a random number is calculated in an interval. Also, the calculated numbers are accumulated, which results in the processing delay. For example, using the coffee machine in this study, which has a maximum sequence length of six (Tab. 4.1) and a processing delay interval of $500ms - 1000ms$, yields a processing delay between three to six seconds. The baseline with a single permanent SUL is run with the following delay intervals in milliseconds, resulting in new baselines for each delay interval: $100-150$, $500-1000$, $3000-5000$, and $5000-10000$. This results in the following average total execution times for the baselines: ~ 20.1 minutes ($100-150$), ~ 22.6 minutes ($500-1000$), ~ 57 minutes ($3000-5000$), ~ 111 minutes ($5000-10000$).

The reduction in cost-benefit for each SUL is even more pronounced when using systems that are slow to respond to queries - represented in this work by adding the mentioned processing delays. As shown in Fig. 5.6, the performance gain for each additional SUL when using sixteen instead of eight SULs is less than 1% when a delay interval between $5000ms - 10000ms$ is used. The higher the processing delay, the more negligible the

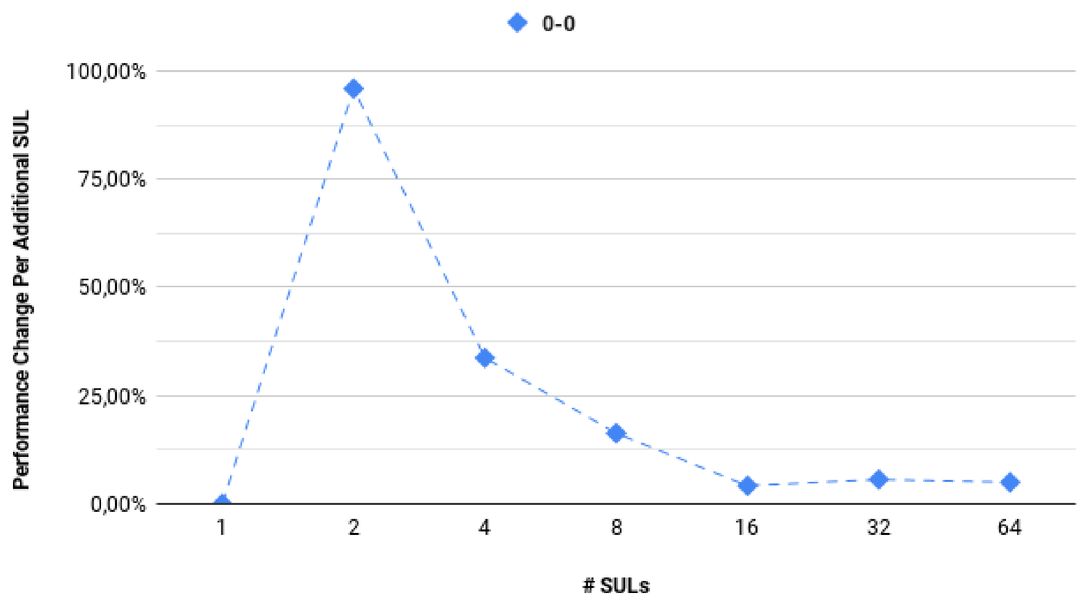

Figure 5.4: Performance gain per SUL instance without processing delay

performance gain from adding more SULs. Fig. 5.5 suggests that the coffee machine's performance gain is limited at about $\sim 347\%$.

This is because SULs take so long to answer a query that fewer SULs are needed to fill the gap until a SUL is ready. Since it takes longer to process a query than it does to finish and start a new SUL, using more than eight SULs is a waste of resources in the high delay area; there is no performance gain. It comes to the point that the performance gain of four SULs is close to that of 64, 32, 16, and 8 SULs.

Here we can assume that the performance gain is capped around the maximum query batch size of the experiment. In the case of the coffee machine used, the batch size is four, as shown in Tab. 4.1. However, the coffee machine has an average query batch size of two. Therefore, using a constant number of SULs always results in time intervals when most SULs are idle, using the available resources inefficiently.

In contrast, in the scenario where all SULs process their queries almost simultaneously (at an interval of about 10 seconds, as in Tab. 5.1), using a SUL count that exceeds the maximum stack size can be beneficial. New queries are published and all available SULs are restarted; therefore, the process rests for a few seconds. New queries can be processed with more SULs while the others restart. However, this can lead to time intervals with idle SULs. The results indicate that the optimal cost-benefit ratio of using the coffee machine is about six to eight SUL instances; thus, this case study confirms the findings of strategy 1a.

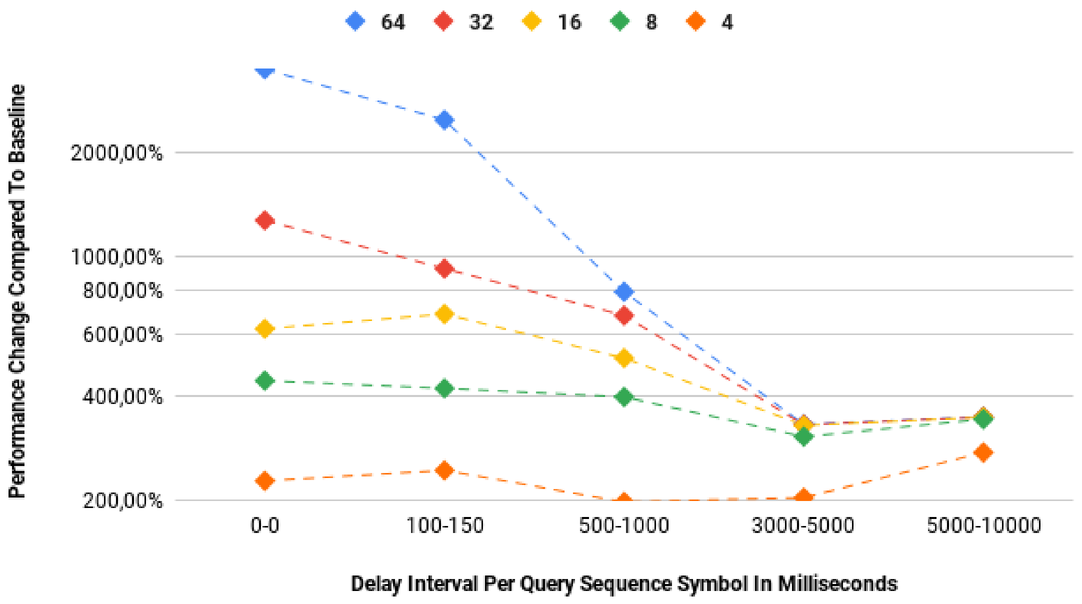

Figure 5.5: Performance gain with processing delay

5.4 Strategy 2: Auto-Scaling

In this study case, KEDA is configured to update the RabbitMQ metrics used every second. In addition, no upper limit is set for the scaling of the SULs. The following HPA autoscaling intervals in seconds are used: 5 ('$hpa5$'), 15 ('$hpa15$'), and 30 ('$hpa30$'). In addition, an approach to calculating demand is taken where demand is calculated every five seconds for upscaling and every 30 seconds for downscaling ('$hpa530$'). Furthermore, each configuration is run with no initial SUL ('$p0$') and one initial SUL ('$p1$'). The baselines of strategy 1a and 1b mentioned earlier are used for comparison - for each processing delay interval respectively.

Fig. 5.7 shows a similar performance gain across all HPA autoscaling intervals compared to the baselines in the case of the coffee machine. The results suggest that the scaling strategy does not have a noticeable impact on the performance gain for the coffee machine as the target system. The outlier may be caused by several external factors, such as non-optimal auto-scaling timings and other factors such as scheduling.

On average, using no initial SUL leads to better $\sim 20\%$ performance compared to an initial SUL when low delay strategies are used. This results from the fact that an initial SUL directly removes a MQ from the queue before the first auto-scaling occurs. At this point, one or more messages are already being processed, and the number of SULs is scaled less than without initial SULs. The initial batch of four queries may result in the number of SULs being one to three instead of four after the initial auto-scaling. The result of this scenario depends, for example, on the scaling interval used and the processing time of the

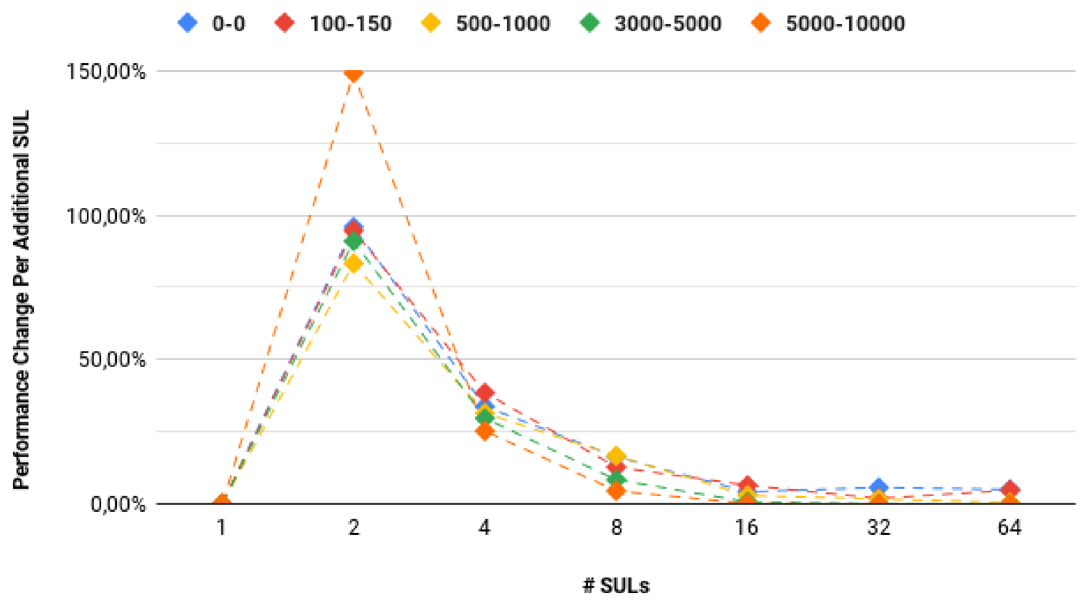

Figure 5.6: Performance gain per SUL instance with processing delay

SULs, as Fig. 5.8 illustrates. In contrast, results suggest that using an initial SUL at a delay interval equal to or greater than $500 - 1000$ performs better in the case of this study ($\sim$ 15% on average). In the remainder of this paper, the best-performing configurations for automatic scaling are compared to the results of strategy 1.

As shown in Fig. 5.8, the auto-scaling strategy that uses one SUL for each MQ in the RabbitMQ queue at demand computation time results in lower performance gains ($\sim$ 200% on average) compared to strategy 1 that uses four ($\sim$ 330% on average) to eight ($\sim$ 381% on average) SUL across all delay intervals. This becomes even more apparent in the context of high delay intervals during processing (Fig. 5.8).

Furthermore, the data shown underscores the results of strategy 1 that further SULs are needed to reduce the architectural overhead of this study and thus further reduce the overall learning time. This can be seen in the performance difference between strategy 1 with four SULs and the auto-scaling approach, where SUL instances are scaled to match the number of queries in the queue. The use of four permanent SULs performs better on average $\sim$ 20% than the auto-scaling approach because, in the scenario where the published query batch size is smaller than the maximum batch size, strategy 1 has additional SULs to reduce the overhead and therefore responds faster to queries of batch sizes 1, 2, and 3 throughout the complete learning process. This further confirms the results of strategy 1.

In summary, in this study case, the auto-scaling strategy provides efficient use of available resources. However, due to architectural overhead, it is beneficial to use more SULs than the current batch size in the queue. Therefore, adjusting the auto-scaling strategy to allow further scaling beyond the current queue length would mitigate the overhead and

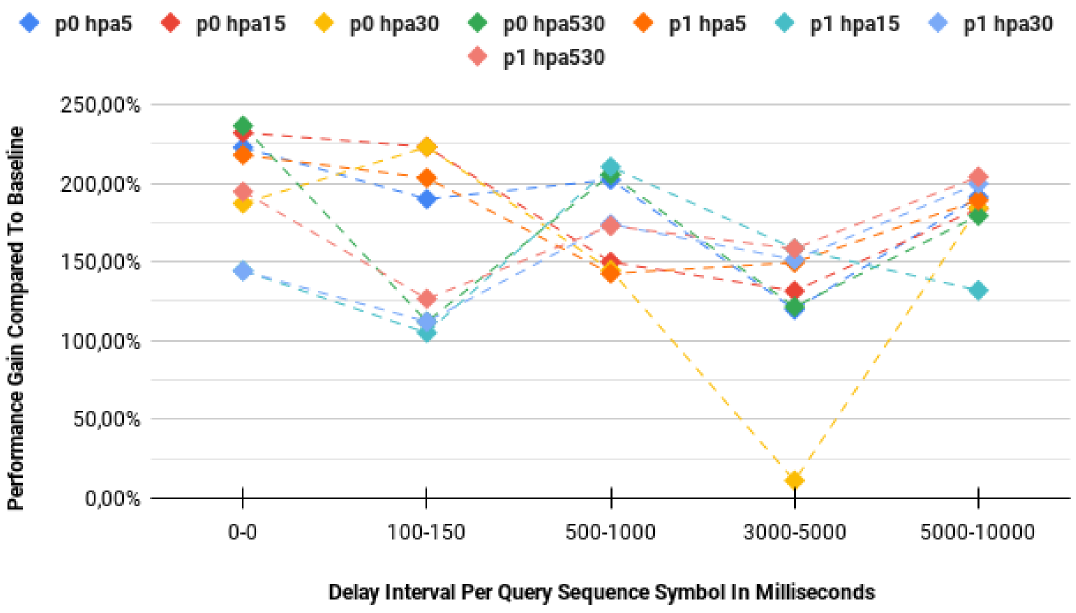

Figure 5.7: Comparison of HPA auto-scaling intervals

result in performance gains closer to the optimal cost-benefit ratio. Nevertheless, strategy 1 requires manual configuration, and statistics such as batch size may not be known in advance. Therefore, automatic scaling is advantageous in this respect and therefore a more reliable choice compared to strategy 1.

5.5 Discussion

Performing active automata learning on a clustered system by parallelizing and distributing the MQs across numerous running SUL instances where the MQs are processed concurrently can lead to a considerable reduction in overall execution time. Depending on the internal/external factors of the actual learning setup, e.g., complexity, time to reset, processing time, and startup time of the SUL, performance is expected to vary.

Using strategy 1 often results in intervals where SULs are idle, resulting in wasted resources and avoidable overhead caused by maintaining the idle SULs. These time intervals and the number of SULs idle are significantly reduced by using auto-scaling strategies, thus using resources more efficiently and reducing costs.

It also depends on the constructed architecture specification and the clustered system used for the learning process. In the case of this work, changing the architecture to restart only the container itself would significantly reduce the aforementioned overhead, leaving only the start-up time of the application itself to matter. Replacing pods leads to, but is not limited to, heavily weighted tasks such as scheduling, reserving resources, and creating the container itself. Thus, this leads to significant overhead.

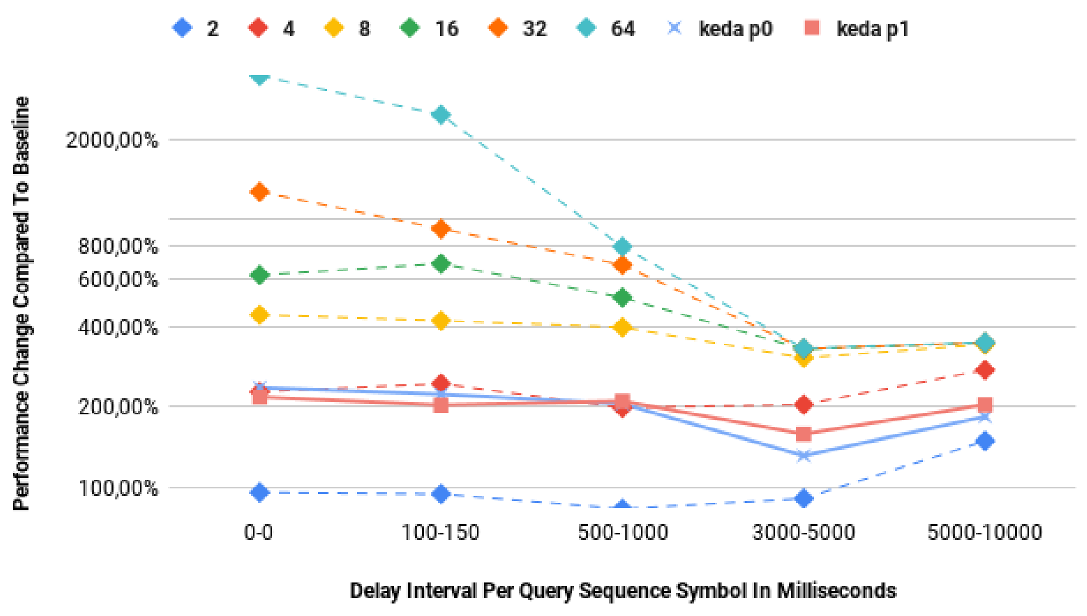

Figure 5.8: Auto-scaling compared to strategy 1a and 1b

Furthermore, increasing the number of SUL instances does not always lead to an increase in performance. It may even result in no performance gain at all. There are two reasons for this:

- First, the overhead of the technologies used grows as the complexity of the container management task increases, or multiple SUL are fast enough that each addition can result in an idle SUL and no performance gain.
- Second, MQs are sent in batches, and each batch must be fully processed until the next batch of MQs is sent. Thus, there is a performance constraint that depends on the size of the query batches that the learner sends. It is important to note that more extensive batch-size experiments can lead to better performance gain than the coffee machine experiment.

Finally, the number of SUL instances is limited by the resource consumption of each instance and the available resources of each cluster system. Consequently, all of these constraints must be taken into account when constructing a learning setup and may affect the overall execution time of a learning process.

In summary, the results of the coffee machine experiment show that performance gains are limited by the learner's published (maximum) query batch size. Further SULs are needed to mitigate the overhead caused by the architecture. Therefore, the equation 5.1 approximates the number of SULs to use for a near-optimal cost-benefit ratio with respect to the coffee machine.

Consequently, an automatic scaling strategy that allows more SULs than the current published batch size to compensate for the overhead would bring the performance closer to the optimal cost-benefit ratio. In the case of this work, a strategy that scales up two SULs for each MQ is expected to perform better and more efficiently. As the results suggest, a SUL count of six to eight pods would be optimum with respect to the coffee machine in both high and low delay areas.

Chapter 6

Conclusion

This thesis investigated how a clustered system can be utilized to perform scalable active automata learning with LearnLib. An architecture was proposed that takes advantage of the properties of a clustered system and its resources to parallelize query processing. Therefore, technologies K8S, Docker, and an event-driven auto-scaler called KEDA were used to manage and automatically scale SUL containers as needed. RabbitMQ was also deployed to provide communication and load balancing between SULs and learners. As a result, LearnLib was extended to include an asynchronous Membership Oracle that communicates with RabbitMQ.

In this research, a simulated system (coffee machine) was studied and used as SUL in all learning configurations. In addition, adaptations were made to the system, such as delays in processing queries and restarting SUL instances, ensuring independence of queries, and simulating more complex real systems. The system and its adaptations were evaluated against two strategies for reducing the overall learning time of a behavioral model: a constant number of SULs and automatic scaling. Both strategies were compared in terms of efficient use of resources and performance gains.

The results show that both strategies are beneficial in reducing the total learning time. However, using a constant number of SULs is inefficient in terms of resource utilization and cannot be adjusted without restarting the learning process to accommodate a varying workload. Depending on the target system, using a high number of SULs may be beneficial, but conversely may also result in no benefit, only more cost. Thus, without knowing the specifications of the target system's learning process, such as the batch size and processing time, it is difficult to determine the right number of SULs to achieve a near optimal cost-benefit ratio. In addition, the batch size of the query can vary greatly, leading to many SULs idle in the worst case.

However, when learning the coffee machine without processing delays, this strategy achieved a $3,450.25\%$ performance gain compared to the baseline with a permanent SUL, resulting in a significant reduction in total learning time from $\sim 1,186.5$ seconds (baseline) to $\sim$ 33.4 seconds (using 64 SULs). Furthermore, the performance gain in high delay areas reached an upper limit of $\sim$ 347%.

The use of automatic scaling proved to be the more reliable choice, especially when statistics about the target system and the learning process itself are unknown. The autoscaling approach resulted in an average performance gain of 200% over all delay intervals when learning the coffee machine. Moreover, changing the auto-scaling strategy to allow more SULs to compensate for the architecture overhead would lead to a more optimal learning process in terms of total learning time and resource consumption.

Thus, the findings and results can be considered promising for learning simple and complex real systems with further adaptations to the proposed architecture. Future investigations are necessary to validate the conclusions drawn in this study, using real systems..

6.1 Future Research

The proposed architecture enables efficient active automata learning for various types of target systems. It is a feasible and practical foundation, with limitations, that enables learning of complex real-world applications if improved. Therefore, several ideas, primarily focused on real-world systems, are provided to extend and improve the proposed architecture for future research.

Learning Real-Life Systems Enabled Through Mappers Although this work attempts to approximate real-life systems as closely as possible through the use of disposable SULs and additional processing delays, further adjustments are needed to use the proposed architecture with real-life systems. The paper [58] introduces so-called 'mappers' that serve as an interface for communicating with real systems such as web applications. By being able to configure SUL mappers on the architecture, models can be learned from real systems. The Automata Learning Experience (ALEX) [1] tool uses this concept to perform active automata learning with LearnLib on web applications. Another approach would be to apply the architecture discovered in this work to the ALEX tool. An extension of the proposed architecture with mappers can be achieved by using a mapper service for each SUL instance that accepts each MQ and transforms it into queries understandable by the target system and executes the transformed query directly on the system.

Reduction of Architecture Overhead In addition to using an alternative automatic scaling strategy, the overall learning time can be improved even more significantly by reducing the overhead of the proposed architecture. As discussed in Chap. 5.1, the Kube Remediator replaces pods in the restart loop with exponential back-off, allowing the use of disposable SULs. However, replacing pods that enter the restart loop introduces a significant overhead of 10 seconds on average, greatly slowing down the learning process.

One way to reduce the overhead is to modify the K8S Kubelet code to allow configuration of the restart loop, or to modify the exponential back-off itself.

Asynchronous sending of membership queries and equivalence queries As mentioned earlier, the reduction in learning time is related to or even limited by the size of the query batch. The reason is that the learner must wait until the entire batch is processed and answered before publishing new queries. Thus, further extension to the LearnLib in future work is required. The DHC algorithm used in this study is based on expanding a spanning tree of explored states using a breadth-first method and merging states with the same result. It examines unexplored states in order. [46, 59] A naive approach would therefore be to explore all unexplored states asynchronously and merge similar explored states at the tree level, resulting in larger batch sizes. A more advanced approach might even be to partially explore the next tree level of already explored states of the level that is still being process. Thus, in the case of the proposed architecture, the queries may be published to the RabbitMQ queue in a more stream-like fashion. However, the complexity of merging similar states increases in this approach. In summary, the possibility of a significant reduction in overall learning time through a learning algorithm that leverages the asynchronous nature of the proposed architecture, and thus the extension of LearnLib, warrants further investigation.

Bibliography

[1] Automata learning experience (alex). https://learnlib.de/projects/alex/. Accessed: 2022-11-10.

[2] Cloudamqp official website. https://www.cloudamqp.com/. Accessed: 2022-11-10.

[3] Compiling spring applications to native executables. https://docs.spring.io/spring-native/docs/current/reference/htmlsingle/. Accessed: 2022-11-10.

[4] Container-based operating systems virtualization - docker. https://www.docker.com. Accessed: 2022-11-10.

[5] Container-based operating systems virtualization - lxc. https://linuxcontainers.org. Accessed: 2022-11-10.

[6] Container-based operating systems virtualization - singularity. https://docs.sylabs.io/guides/3.5/user-guide/introduction.html. Accessed: 2022-11-10.

[7] Container orchestration tool - apache mesos. https://mesos.apache.org. Accessed: 2022-11-10.

[8] Container orchestration tool - docker swarm. https://docs.docker.com/engine/swarm/. Accessed: 2022-11-10.

[9] Container orchestration tool - kubernetes. https://kubernetes.io. Accessed: 2022-11-10.

[10] Container orchestration tool - kubernetes - hpa. https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/. Accessed: 2022-11-10.

[11] Graalvm - high-performance jdk. https://www.graalvm.org. Accessed: 2022-11-10.

[12] Horizontal pod autoscaling. https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/#configurable-scaling-behavior. Accessed: 2022-11-10.

[13] Java official website. https://www.java.com/de/. Accessed: 2022-11-10.

[14] K8s pod lifecylce. `https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/`. Accessed: 2022-11-10.

[15] Kubernetes event-driven autoscaling. `https://keda.sh`. Accessed: 2022-11-10.

[16] Kubernetes setup tool - microk8s. `https://microk8s.io`. Accessed: 2022-11-10.

[17] Learnlib official website. `https://learnlib.de/`. Accessed: 2022-11-10.

[18] Learnlib official website - performance. `https://learnlib.de/performance/`. Accessed: 2022-11-10.

[19] libalf official website. `http://libalf.informatik.rwth-aachen.de`. Accessed: 2022-11-10.

[20] Most loved and dreaded technologies. `https://survey.stackoverflow.co/2022/#most-loved-dreaded-and-wanted-tools-tech-love-dread`. Accessed: 2022-11-10.

[21] Rabbitmq official website. `https://www.rabbitmq.com/`. Accessed: 2022-11-10.

[22] Spring boot official website. `https://spring.io/projects/spring-boot`. Accessed: 2022-11-10.

[23] Fides Aarts, Joeri De Ruiter, and Erik Poll. Formal models of bank cards for free. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops*, pages 461–468. IEEE, 2013.

[24] Fides Aarts, Julien Schmaltz, and Frits Vaandrager. Inference and abstraction of the biometric passport. In *International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*, pages 673–686. Springer, 2010.

[25] Theodora Adufu, Jieun Choi, and Yoonhee Kim. Is container-based technology a winner for high performance scientific applications? In *2015 17th Asia-Pacific Network Operations and Management Symposium (APNOMS)*, pages 507–510. IEEE, 2015.

[26] S Anish Babu, MJ Hareesh, John Paul Martin, Sijo Cherian, and Yedhu Sastri. System performance evaluation of para virtualization, container virtualization, and full virtualization using xen, openvz, and xenserver. In *2014 fourth international conference on advances in computing and communications*, pages 247–250. IEEE, 2014.

[27] Naylor G Bachiega, Paulo SL Souza, Sarita M Bruschi, and Simone Do RS De Souza. Container-based performance evaluation: a survey and challenges. In *2018 IEEE International Conference on Cloud Engineering (IC2E)*, pages 398–403. IEEE, 2018.

[28] David Balla, Csaba Simon, and Markosz Maliosz. Adaptive scaling of kubernetes pods. In *NOMS 2020-2020 IEEE/IFIP Network Operations and Management Symposium*, pages 1–5. IEEE, 2020.

[29] Oliver Bauer, Johannes Neubauer, Bernhard Steffen, and Falk Howar. Reusing system states by active learning algorithms. In *International Workshop on Eternal Systems*, pages 61–78. Springer, 2011.

[30] Aditya Bhardwaj and C Rama Krishna. Virtualization in cloud computing: Moving from hypervisor to containerization—a survey. *Arabian Journal for Science and Engineering*, 46(9):8585–8601, 2021.

[31] Eric W Biederman and Linux Networx. Multiple instances of the global linux namespaces. In *Proceedings of the Linux Symposium*, volume 1, pages 101–112. Citeseer, 2006.

[32] Emiliano Casalicchio. Container orchestration: a survey. *Systems Modeling: Methodologies and Tools*, pages 221–235, 2019.

[33] Susanta Nanda Tzi-cker Chiueh and Stony Brook. A survey on virtualization technologies. *Rpe Report*, 142, 2005.

[34] Minh Thanh Chung, Nguyen Quang-Hung, Manh-Thin Nguyen, and Nam Thoai. Using docker in high performance computing applications. In *2016 IEEE Sixth International Conference on Communications and Electronics (ICCE)*, pages 52–57. IEEE, 2016.

[35] Philippe Dobbelaere and Kyumars Sheykh Esmaili. Kafka versus rabbitmq: A comparative study of two industry reference publish/subscribe implementations: Industry paper. In *Proceedings of the 11th ACM international conference on distributed and event-based systems*, pages 227–238, 2017.

[36] Marco Henrix. Performance improvement in automata learning. Master's thesis, Radboud University, Nijmegen, August 2015. `http://www.ru.nl/publish/pages/769526/z_masterthesismhenrix_marco_henrix.pdf`.

[37] Saiful Hoque, Mathias Santos De Brito, Alexander Willner, Oliver Keil, and Thomas Magedanz. Towards container orchestration in fog computing infrastructures. In *2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC)*, volume 2, pages 294–299. IEEE, 2017.

[38] Falk Howar, Malte Isberner, Maik Merten, and Bernhard Steffen. Learnlib tutorial: from finite automata to register interface programs. In *International Symposium On*

Leveraging Applications of Formal Methods, Verification and Validation, pages 587–590. Springer, 2012.

[39] Falk Howar and Bernhard Steffen. Active automata learning in practice. In *Machine Learning for Dynamic Software Analysis: Potentials and Limits*, pages 123–148. Springer, 2018.

[40] Malte Isberner, Falk Howar, and Bernhard Steffen. The open-source learnlib. In *International Conference on Computer Aided Verification*, pages 487–495. Springer, 2015.

[41] Ramon Janssen, Frits W Vaandrager, and Sicco Verwer. Learning a state diagram of tcp using abstraction. *Bachelor thesis, ICIS, Radboud University Nijmegen*, 12, 2013.

[42] Vineet John and Xia Liu. A survey of distributed message broker queues. *arXiv preprint arXiv:1704.00411*, 2017.

[43] Asif Khan. Key characteristics of a container orchestration platform to enable a modern application. *IEEE cloud Computing*, 4(5):42–48, 2017.

[44] Paul Menage. Linux control groups. `https://www.kernel.org/doc/html/latest/admin-guide/cgroup-v1/cgroups.html`, 2004. Accessed: 2022-11-10.

[45] Maik Merten. *Active automata learning for real life applications.* PhD thesis, TU Dortmund University, Dortmund, January 2013. `https://eldorado.tu-dortmund.de/bitstream/2003/29884/1/dissertation-pdfa.pdf`.

[46] Maik Merten, Falk Howar, Bernhard Steffen, and Tiziana Margaria. Automata learning with on-the-fly direct hypothesis construction. In *International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*, pages 248–260. Springer, 2011.

[47] Thanh-Tung Nguyen, Yu-Jin Yeom, Taehong Kim, Dae-Heon Park, and Sehan Kim. Horizontal pod autoscaling in kubernetes for elastic container orchestration. *Sensors*, 20(16):4621, 2020.

[48] Claus Pahl and Brian Lee. Containers and clusters for edge cloud architectures–a technology review. In *2015 3rd international conference on future internet of things and cloud*, pages 379–386. IEEE, 2015.

[49] Harald Raffelt, Bernhard Steffen, and Therese Berg. Learnlib: A library for automata learning and experimentation. In *Proceedings of the 10th international workshop on Formal methods for industrial critical systems*, pages 62–71, 2005.

[50] Nathan Regola and Jean-Christophe Ducom. Recommendations for virtualization technologies in high performance computing. In *2010 IEEE Second International Conference on Cloud Computing Technology and Science*, pages 409–416. IEEE, 2010.

[51] Fernando Rodríguez-Haro, Felix Freitag, Leandro Navarro, Efraín Hernánchez-sánchez, Nicandro Farías-Mendoza, Juan Antonio Guerrero-Ibáñez, and Apolinar González-Potes. A summary of virtualization techniques. *Procedia Technology*, 3:267–272, 2012.

[52] Robert Rose. Survey of system virtualization techniques. March 2004. `https://ir.library.oregonstate.edu/dspace/bitstream/1957/9907/1/rose-virtualization.pdf`.

[53] Maciej Rostanski, Krzysztof Grochla, and Aleksander Seman. Evaluation of highly available and fault-tolerant middleware clustered architectures using rabbitmq. In *2014 federated conference on computer science and information systems*, pages 879–884. IEEE, 2014.

[54] Sasidhar Sekar. Autoscaling in kubernetes: Why doesn't the horizontal pod autoscaler work for me? `https://medium.com/expedia-group-tech/autoscaling-in-kubernetes-why-doesnt-the-horizontal-pod-autoscaler-\-work-for-me-5f0094694054`, 2020. Accessed: 2022-11-10.

[55] Wouter Smeenk. Applying automata learning to complex industrial software. *Master's thesis, Radboud University Nijmegen*, 2012.

[56] Rick Smetsers, Michele Volpato, Frits Vaandrager, and Sicco Verwer. Bigger is not always better: on the quality of hypotheses in active automata learning. In *International Conference on Grammatical Inference*, pages 167–181. PMLR, 2014.

[57] Stephen Soltesz, Herbert Pötzl, Marc E Fiuczynski, Andy Bavier, and Larry Peterson. Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors. In *Proceedings of the 2Nd ACM SIGOPS/EuroSys european conference on computer systems 2007*, pages 275–287, 2007.

[58] Bernhard Steffen, Falk Howar, and Malte Isberner. Active automata learning: from dfas to interface programs and beyond. In *International Conference on Grammatical Inference*, pages 195–209. PMLR, 2012.

[59] Bernhard Steffen, Falk Howar, and Maik Merten. Introduction to active automata learning from a practical perspective. In *International School on Formal Methods for the Design of Computer, Communication and Software Systems*, pages 256–296. Springer, 2011.

[60] Falk Howar Bernhard Steffen, Malte Isberner. Learnlib: An open-source framework for active automata learning. *TU Dortmund University, Dortmund, Germany*, July 2015. `https://www.researchgate.net/profile/Malte-Isberner/publication/281460959_Poster/links/55e98bc108aeb6516264854f/Poster.pdf`.

[61] Martin Tappler, Bernhard K Aichernig, and Roderick Bloem. Model-based testing iot communication via active automata learning. In *2017 IEEE International conference on software testing, verification and validation (ICST)*, pages 276–287. IEEE, 2017.

[62] Max Tijssen, Erik Poll, and Joeri de Ruiter. Automatic modeling of ssh implementations with state machine learning algorithms. *Bachelor thesis, Radboud University, Nijmegen*, 2014.

[63] Eddy Truyen, Dimitri Van Landuyt, Davy Preuveneers, Bert Lagaisse, and Wouter Joosen. A comprehensive feature comparison study of open-source container orchestration frameworks. *Applied Sciences*, 9(5):931, 2019.

[64] James Turnbull. *The Docker Book: Containerization is the new virtualization.* James Turnbull, 2014.

[65] W Eric Wong, Vidroha Debroy, Adithya Surampudi, HyeonJeong Kim, and Michael F Siok. Recent catastrophic accidents: Investigating how software was responsible. In *2010 Fourth International Conference on Secure Software Integration and Reliability Improvement*, pages 14–22. IEEE, 2010.

[66] Miguel G Xavier, Marcelo V Neves, Fabio D Rossi, Tiago C Ferreto, Timoteo Lange, and Cesar AF De Rose. Performance evaluation of container-based virtualization for high performance computing environments. In *2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 233–240. IEEE, 2013.

Index Of Abbreviations

SUL System Under Learning 1

MO Membership Oracle 8

MQ Membership Query 2

MBT Model-Based Testing 1

MAT Minimally Adequate Teacher 7

EQ Equivalence Query 2

VM Virtual Machine 9

OS Operating System 9

HPA Horizontal Pod Autoscaler 13

K8S Kubernetes 10

CPU Central Processing Unit 10

VPA Vertical Pod Autoscaler 18

KEDA Kubernetes Event-driven Autoscaling 13

List of Figures

Appendix A

Evaluation Results

Table A.1: Evaluation Baselines

# SULs	Delay Interval Per Symbol In Sequence In MS	Total Learn Time In Seconds
1	0-0	1187
1	100-150	1204
1	500-1000	1355
1	3000-5000	3420
1	5000-10000	6657

Table A.2: Performance Change To Baseline - Delay Interval 0-0

# SULs	Total Learn Time In Seconds	Performance Change To Baseline
64	33	3450.25%
32	86	1272.36%
16	164	622.33%
8	219	441.79%
4	362	227.96%
2	606	95.90%
1	1187	0.00%

Table A.3: Performance Change To Baseline - Delay Interval 100-150

# SULs	Total Learn Time In Seconds	Performance Change To Baseline
64	47	2468.70%
32	117	925.49%
16	153	685.90%
8	232	419.89%
4	350	244.37%
2	619	94.63%
1	1204	0.00%

Table A.4: Performance Change To Baseline - Delay Interval 500-1000

# SULs	Total Learn Time In Seconds	Performance Change Change To Baseline
64	152	794.10%
32	174	679.26%
16	221	512.87%
80	273	396.81%
4	454	198.45%
2	739	83.29%
1	1355	0.00%

Table A.5: Performance Change To Baseline - Delay Interval 3000-5000

# SULs	Total Learn Time In Seconds	Performance Change To Baseline
16	793	331.03%
32	795	330.17%
16	798	328.76%
8	844	305.41%
4	1125	204.08%
2	1791	90.99%
1	3420	0.00%

Table A.6: Performance Change To Baseline - Delay Interval 5000-10000

# SULs	Total Learn Time In Seconds	Performance Change To Baseline
32	1485	348,26%
64	1487	347,56%
16	1489	347,15%
8	1489	342,40%
4	1775	275,00%
2	2671	149,26%
1	6657	0.00%

Table A.7: Minimum Overhead - Delay Interval 0-0

# SULs	Start-Up Time In MS	Processing Time In MS	Response Time In MS	Overhead In Seconds
8	84.42	0.32	0.88	-0.08
1	81.87	0.34	89.96	0.008

Table A.8: Average Overhead - Delay Interval 0-0

# SULs	Start-Up Time In MS	Processing Time In MS	Response Time In MS	Overhead In Seconds
1	211.18	2.4	24396.98	24.2
8	320.62	3.27	12941.36	12.62

Table A.9: Maximum overhead with processing delay interval 0-0

# SULs	Start-Up Time In MS	Processing Time In MS	Response Time In MS	Overhead In Seconds
1	101.66	0.62	10051.54	9.95
8	113.5	0.67	1851.76	1.74

Table A.10: Min. Start-up time based on SUL count with processing delay interval 0-0

# SULs	Start-Up Time In Milliseconds	Performance Change To Baseline
64	86.94	-5.83%
32	86.59	-3.02%
16	84.41	-3.01%
8	84.42	-3.02%
4	81.72	0.19%
2	84.08	-2.63%
1	81.87	0.00%

Table A.11: Avg. Start-up time based on SUL count with processing delay interval 0-0

# SULs	Start-Up Time In Milliseconds	Performance Change To Baseline
64	139.51	-27.13%
32	142.71	-28.76%
16	117.93	-13.80%
8	113.5	-10.43%
4	106.13	-4.21%
2	104.72	-2.92%
1	101.66	0.00%

Table A.12: Max. Start-up time based on SUL count with processing delay interval 0-0

# SULs	Start-Up Time In Milliseconds	Performance Change To Baseline
64	332.15	-36.42%
32	474.31	-55.48%
16	289.2	-26.98%
8	320.62	-34.13%
4	277.03	-23.77%
2	163.96	28.80%
1	211.19	0.00%

Table A.13: Min. Start-up time based on SUL count with processing delay interval 5000-10000

# SULs	Start-Up Time In Milliseconds	Performance Change To Baseline
64	89.67	-1.58%
32	86.56	1.96%
16	83.92	5.17%
8	84.67	4.23%
4	87.89	0.42%
2	85.99	2.63%
1	88.25	0.00%

Table A.14: Avg. Start-up time based on SUL count with processing delay interval 5000-10000

# SULs	Start-Up Time In Milliseconds	Performance Change To Baseline
64	154.99	-31.58%
32	115.34	-8.07%
16	109.68	-3.32%
8	112.17	-5.46%
4	109.41	-3.08%
2	106.79	-0.70%
1	106.04	0.00%

Table A.15: Max. Start-up time based on SUL count with processing delay interval 5000-10000

# SULs	Start-Up Time In Milliseconds	Performance Change To Baseline
64	1693.46	-90.79%
32	296.27	-47.38%
16	273.47	-42.99%
8	470.17	-66.84%
4	289.85	-46.21%
2	183.1	-14.85%
1	155.91	0.00%

Table A.16: Performance Change To Baseline - Auto-Scaling - Delay Interval 0-0

# SULs	Scale Up/Down Interval	Total Learn Time In Seconds	Performance Change To Baseline
Initial 0	5/30	353	236.38%
Initial 0	15/15	357	231.93%
Initial 0	5/5	368	222.62%
Initial 1	5/5	373	217.90%
Initial 1	5/30	403	194.55%
Initial 0	30/30	413	187.09%
Initial 1	15/15	485	144.44%
Initial 1	30/30	486	144.17%
1	-/-	1187	0.00%

Table A.17: Performance Change To Baseline - Auto-Scaling - Delay Interval 100-150

# SULs	Scale Up/Down Interval	Total Learn Time In Seconds	Performance Change Change To Baseline
Initial 0	15/15	373	223.00%
Initial 0	30/30	373	222.96%
Initial 1	5/5	397	203.24%
Initial 1	30/30	402	199.95%
Initial 0	5/5	416	189.85%
Initial 1	5/30	532	126.56%
Initial 0	5/30	568	111.95%
Initial 1	15/15	588	104.92%
1	-/-	1204	0.00%

Table A.18: Performance Change To Baseline - Auto-Scaling - Delay Interval 500-1000

# SULs	Scale Up/Down Interval	Total Learn Time In Seconds	Performance Change To Baseline
Initial 1	15/15	437	210.31%
Initial 0	5/30	444	205.40%
Initial 0	5/5	448	202.17%
Initial 1	30/30	495	173.89%
Initial 1	5/30	497	172.81%
Initial 0	15/15	542	149.91%
Initial 0	30/30	554	144.72%
Initial 1	5/5	559	142.57%
1	-/-	1355	0.00%

Table A.19: Performance Change To Baseline - Auto-Scaling - Delay Interval 3000-5000

# SULs	Scale Up/Down Interval	Total Learn Time In Seconds	Performance Change To Baseline
Initial 1	15/15	1323	158.42%
Initial 1	5/30	1324	158.38%
Initial 1	30/30	1360	151.40%
Initial 1	5/5	1370	149.59%
Initial 0	15/15	1478	131.48%
Initial 0	5/30	1544	121.47%
Initial 0	5/5	1557	119.71%
Initial 0	30/30	3079	11.08%
1	-/-	3420	0.00%

Table A.20: Performance Change To Baseline - Auto-Scaling - Delay Interval 5000-10000

# SULs	Scale Up/Down Interval	Total Learn Time In Seconds	Performance Change To Baseline
Initial 1	5/30	2190	204.00%
Initial 1	30/30	2221	199.69%
Initial 0	5/5	2287	191.10%
Initial 1	5/5	2303	189.06%
Initial 0	30/30	2343	184.18%
Initial 0	15/15	2346	183.81%
Initial 0	5/30	2383	179.31%
Initial 1	15/15	2871	131.89%
1	-/-	6657	0.00%

Eidesstattliche Versicherung

(Affidavit)

Saevecke, Julien	190843
Name, Vorname (surname, first name)	Matrikelnummer (student ID number)

[✓ Bachelorarbeit (Bachelor's thesis)] ☐ Masterarbeit (Master's thesis)

Titel (Title)

Automata Learning At Scale: Evaluation of Query Parallelization Strategies In the Context of Clustered Systems

Ich versichere hiermit an Eides statt, dass ich die vorliegende Abschlussarbeit mit dem oben genannten Titel selbstständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie wörtliche und sinngemäße Zitate kenntlich gemacht. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

I declare in lieu of oath that I have completed the present thesis with the above-mentioned title independently and without any unauthorized assistance. I have not used any other sources or aids than the ones listed and have documented quotations and paraphrases as such. The thesis in its current or similar version has not been submitted to an auditing institution before.

Ahnatal, 01.10.2022	[illegible]
Ort, Datum (place, date)	Unterschrift (signature)

Belehrung:
Wer vorsätzlich gegen eine die Täuschung über Prüfungsleistungen betreffende Regelung einer Hochschulprüfungsordnung verstößt, handelt ordnungswidrig. Die Ordnungswidrigkeit kann mit einer Geldbuße von bis zu 50.000,00 € geahndet werden. Zuständige Verwaltungsbehörde für die Verfolgung und Ahndung von Ordnungswidrigkeiten ist der Kanzler/die Kanzlerin der Technischen Universität Dortmund. Im Falle eines mehrfachen oder sonstigen schwerwiegenden Täuschungsversuches kann der Prüfling zudem exmatrikuliert werden. (§ 63 Abs. 5 Hochschulgesetz - HG -).

Die Abgabe einer falschen Versicherung an Eides statt wird mit Freiheitsstrafe bis zu 3 Jahren oder mit Geldstrafe bestraft.

Die Technische Universität Dortmund wird ggf. elektronische Vergleichswerkzeuge (wie z.B. die Software „turnitin") zur Überprüfung von Ordnungswidrigkeiten in Prüfungsverfahren nutzen.

Die oben stehende Belehrung habe ich zur Kenntnis genommen:

Official notification:
Any person who intentionally breaches any regulation of university examination regulations relating to deception in examination performance is acting improperly. This offense can be punished with a fine of up to EUR 50,000.00. The competent administrative authority for the pursuit and prosecution of offenses of this type is the Chancellor of TU Dortmund University. In the case of multiple or other serious attempts at deception, the examinee can also be unenrolled, Section 63 (5) North Rhine-Westphalia Higher Education Act (*Hochschulgesetz, HG*).

The submission of a false affidavit will be punished with a prison sentence of up to three years or a fine.

As may be necessary, TU Dortmund University will make use of electronic plagiarism-prevention tools (e.g. the "turnitin" service) in order to monitor violations during the examination procedures.

I have taken note of the above official notification:*

Ahnatal, 01.10.2022	[illegible]
Ort, Datum (place, date)	Unterschrift (signature)

***Please be aware that solely the German version of the affidavit ("Eidesstattliche Versicherung") for the Bachelor's/ Master's thesis is the official and legally binding version.**