technische universität
dortmund

Bachelor 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

Technische Universität Dortmund
Fakultät für Informatik
Programmiersysteme (LS-5)
http://ls5-www.cs.tu-dortmund.de

Contents

Chapter 1

Introduction

These days, computer software is used in many objects in the environment. It begins with the apparent computer helping us in our daily life, to crucial medical equipment for preserving or saving lives, and ends with self-driving cars for transporting goods or people. Regrettably, even the simplest software is prone to bugs expressing themselves differently. A bug in software can express itself as a simple button on a website not working. However, it can also lead to catastrophic financial damage or, even worse, to loss of life.[68] Thus, it is important to prevent software failure or bugs that can influence lives in a bad way or even endanger them. Consequently, between 30 and 60 percent of the software development process consists of testing.[66]

One effective approach to prevent or reduce software failures and raise software quality is Model-Based Testing (MBT). In this approach, a model, for example, a Finite-State Machine (FSM), is used to represent the desired behavior of a System Under Test (SUT). With the help of the model, test cases can be generated to test whether the implementation of the SUT is working correctly. In MBT, creating the model can be time-consuming and cost-inefficient when done manually and vulnerable to errors, especially regarding complex real-life systems.

Active automata learning is a technique to infer a behavioral model representing the actual behavior of the target system (namely System Under Learning (SUL)) by executing test queries on a system. A framework popular for setting up active automata learning is LearnLib[18]. However, this technique has two major concerns, which are emphasized in complex real-life systems.

The first primary concern is that active automata learning is a rather costly technique because gaining sufficient knowledge for inferring a behavioral model of the target system is glued to a significant number of test queries sent to the SUL. Depending on the system's complexity, responding to such a test query may induce high latency operations, taking a long time to process. Consequently, learning behavioral models of such a system may take hours, days, or even months[56].

The second concern is that active automata learning requires test query independence to learn correctly. Meaning no test query affects another test query. Therefore, the requirement on the target system is to offer reset functionality or any other abstractions guaranteeing that no query impacts the other. However, in the context of complex real-life systems, this requirement is often not fulfilled because the system may consist of, e.g., of multiple services and persistent storage to maintain any state.

These two concerns were why inferring behavioral models of real-life systems was not practicable. However, in 2011 Next Generation LearnLib (NGLL)(further referred to as LearnLib for simplicity), with the focus on offering various features making the learning of real-life systems more feasible, such as reset mechanisms and abstraction/refinement techniques, was released. Furthermore, it is implemented in an extensible modular fashion, allowing active automata learning on different system architectures, e.g., a distributed system, thus opening up the possibility of more complex learning setups. [47]

This change led to more research focusing on real-life applications, such as making active automata learning less costly and introducing so-called mappers as a communication interface to real-life systems. Consequently, active automata learning was successfully used to learn various behavioral models and the tool Automata Learning Experience (ALEX)[2] for inferring models of web applications and JSON-based web services developed since 2015.

Despite progress, active automata learning is still a heavily weighted process. This work proposes an architecture to perform scale-able active automata learning. Utilizing a clustered system and technologies enables the simultaneous processing of test queries distributed to multiple SULs at once. Furthermore, run-time auto-scaling strategies are used to efficiently use the available cluster resources and compared to an approach without scaling. The target SULs are adjusted to accommodate real-life systems using processing delays and restarting them to ensure test query independence. Consequently, all kinds of SULs can be learned without offering additional functionality.

1.1 Related Work

In the context of real-life systems - the time taken to create behavioral models for MBT or other purposes, using active automata learning can take, based on the complexity of the target system, days, months, or even years. Therefore, often not feasible in, e.g., agile or time-constrained environments. Thus, existing studies examined various methods to reduce the total execution time of an active automata learning process by optimizing equivalence queries and used learn algorithms[57, 56], reusing states(removing duplicated work)[30, 37] and abstraction layers (e.g., using equivalence classes)[25].

In [37], 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, [46] examined active automata learning regarding challenges and which steps can be taken to make it more practicable regarding real-life systems. This work follows the same approach 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 complements the scale-able aspect of the method taken in [37] and shifts the context, similar to the cloud computing approach [46], from the active automata learning process running on a single physical machine into a clustered system. A clustered system contains a group of at least two physical and virtual machines, interconnected with each other in a way that they can act like a single system. Operating active automata learning on a clustered system through parallelizing and distributing the processing of queries to several SUL instances can improve the execution time to learn a behavioral model notably and is not limited by the resources of a single machine. A clustered system introduces management efforts to the general setup to accomplish that. Still, utilizing virtualization technology and orchestration tools, it can provide and manage resources on demand and adapt to ever-changing workloads by scaling resources up and down while running and being resilient. Therefore this work has the following primary and sub-goals:

Construct an architecture utilizing a clustered system to perform scale-able active automata learning

- Construct an architecture allowing for automatically scale of the number of SULs based on demand

- Examine how the demand for SULs can be calculated

- Evaluate the practicability and limitations of this approach

1.3 Structure

On that account, this work is structured in the following way. In the first part (chapter 2), the fundamentals of active automata learning and the process of learning a behavioral model of a SUL are covered. Furthermore, it is essential to cover mealy machines and the tool LearnLib in this context. Finally, the fundamental technologies used to set up active automata learning in a clustered system are explained to complete the fundamentals.

In the second part (chapter 3 and chapter 4), the constructed architecture of the clustered automata learn setup is explained, accompanied by an explanation of how experiments are set up, run, and measured for later evaluation.

Before last, the results are presented, analyzed and evaluated (**??** and chapter 5). Consequently, insights about practicability, limitations, and performance concerning the constructed architecture are disclosed in this part.

The last part of this work (section 7.1 and chapter 7) contains recommendations about possible improvements or extensions to the architecture used to perform the experiments for future research. Finally, this work concludes by determining whether the goals stated in section 1.2 were accomplished.

Chapter 2

Preliminaries

Before explaining an architecture can be constructed 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 Active Automata Learning

Active Automata Learning is a process that automatically infers formal behavioral models of, e.g., black box systems through testing. [61] 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)[50, 46]:

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 the unknown target L

2. A **Equivalence query (EQ)** is defined like a MQ, but the difference is that this 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

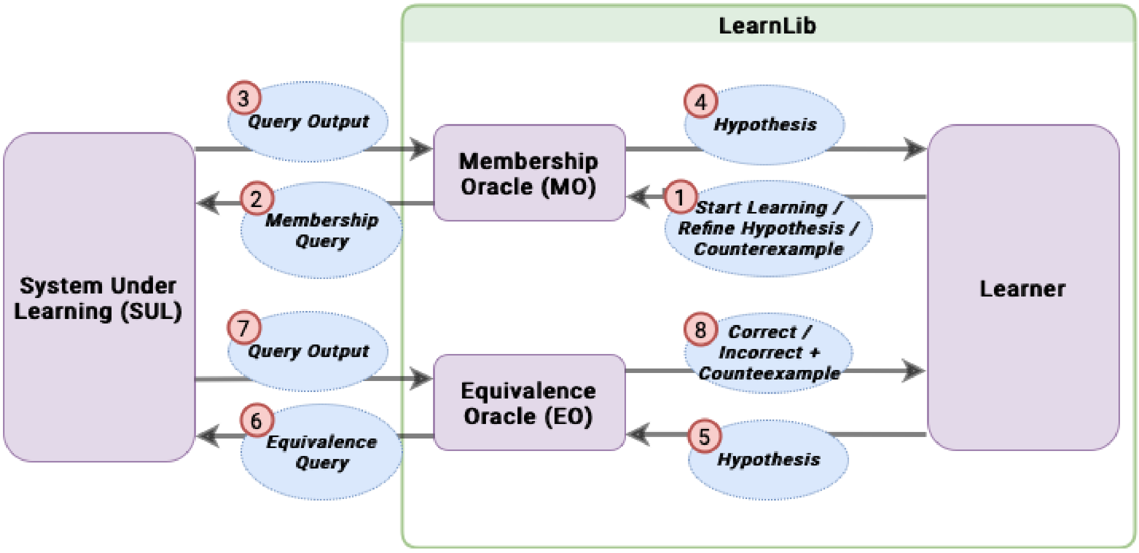

Figure 2.1: Learn Process With LearnLib Based On [37]

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-world 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. [62]

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-world 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. [40]

At last, simple input alphabets often do not suffice to communicate with real-world 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.[59, 39]

There is a choice to be made when it comes to libraries enabling automata learning adhering to the explained MAT model: LearnLib[18] or libalf[20]. 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.[41, 19]

2.1.1 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 [50]. Additionally, the design structure accommodates using different infrastructures, e.g., distributed systems, while maintaining a good performance [41, 46, 37].

Further, it provides various learning algorithms, equivalence approximation strategies, and filters such as caches to reduce the total number of queries.[46] Thus, LearnLib was used to learn models of various implementations. Some of these implementations are: TCP[42], SSH[63], bank cards[24] and printer software[56].

A more in-depth active automata learning process, implemented with LearnLib and adhering to the MAT model, is illustrated in Figure 2.1. 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:

- **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.

- **Equivalence Oracle (EO)** 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 EO 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 is utilized in this work, and, e.g., its components, such as a custom MO is, implemented to adhere to the unique requirements of a clustered system (explained in section 3.1).

2.2 Mealy Machine

LearnLib offers various ways to implement and learn finite deterministic automatons, e.g., Mealy machines. [41] In this work, it is crucial to understand how Mealy machines work because the behavior of the systems used to evaluate the architecture are represented as Mealy machines and used as SULs.

Mealy machines are a variant of a FSM, where an output alphabet accompanies the input alphabet. The output is determined by the current input and the current state of

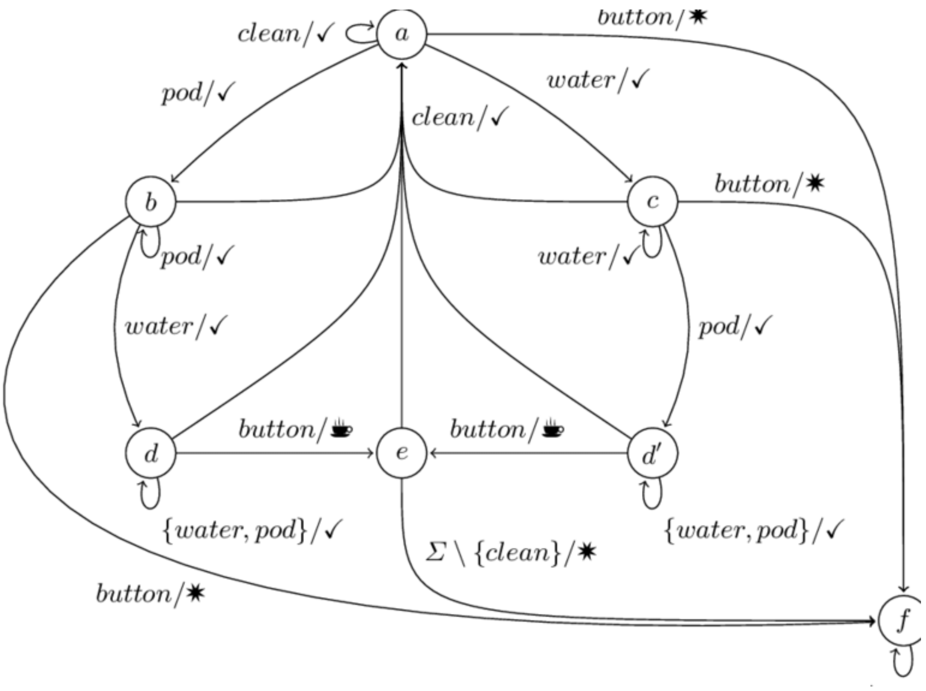

Figure 2.2: A simple coffee machine as a Mealy machine [60]

the Mealy machine. Mealy machines differentiate themselves from other FSMs by having a transition function that is defined for all input symbols resulting in the output to a sequence of inputs being deterministic. A state diagram can represent Mealy machines.

2.2.1 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 Figure 2.2 represents a simple 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 (✓), invalid interaction ($\star$) and making coffee (☕). But 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 (✓) to make a cup of coffee.

2.3 Virtualization Technologies

To perform scaleable automata learning on a clustered system, it is essential to utilize the cluster resources efficiently by having SULs that are easy to distribute, replicate, and fast to deploy and run. The SULs should utilize as little as possible Central Processing Unit (CPU) and memory to run a high quantity of SULs simultaneously to process a high count of queries in parallel. Another requirement is to deploy a new SUL or terminate an existing SUL replica while a learning process is still running to enable scaling on-demand. For this purpose, the use of virtualization technology is vital.

Virtualization generally is a technology that partitions physical system resources to create an abstraction of one or many isolated environments. [52, 34] Virtualization is used to make resource utilization more efficient and provide more flexibility. [31] This technology can be differentiated into multiple types of virtualization. The most used techniques are hypervisor-based and Operating System (OS)-level virtualization. [27, 28]

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. [52, 34] This technique brings an overhead because each VM runs its OS and file system. This overhead is why a VM is slow to launch[28, 51] and should be used in scenarios where complete isolation and security, ease of management, and customization are the primary features in need.[53, 34]

The OS-level virtualization technique is also known as Container-based operating system (COS) virtualization[58] or containerization[65]. 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[69] the use of namespaces[32]. Generally, the resources that each environment has access to are managed by either limiting/prioritizing through Control Groups (cgroups)[45]. This results in an environment in which running processes and the required binaries to run these processes, along with 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. [35, 65] By sharing the same host OS kernel and that 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. This allows running more containers than VMs on the same physical host machine. [26, 31]

The following existing container-based virtualization tools can be used to turn applications into containers but are not limited to LXC[6], Singularity[7], and Docker[5]. 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.[21] Docker uses so-called images (created through Dockerfiles) containing step-by-step instructions to create an instance of the image - 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 essential to note that the varying simultaneously running number of SUL containers must be managed.

2.4 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 no easy task 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. The management of hundreds or even more containers is achievable with the help of orchestration tools.[?] Consequently, the range of primary features regarding containers of such orchestration tools are, e.g., resource management, scheduling, fault tolerance, and auto-scaling.[33, 44, 49]

In the context of this work, the auto-scaling feature is essential to perform 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 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)[10], Docker Swarm[9], and Apache Mesos[8]. 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.[64] Due to being open-source and enabling customization, it is actively developed by a large community.[21] 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.

To conclude, orchestrating lightweight SUL containers using K8S enables to move of active automata learning to an environment of a clustered system. Furthermore, it enables the efficient use of resources and matching the number of active SULs to the varying workload of a running learning process through auto-scaling.

Chapter 3

Automata Learning of Clustered Systems

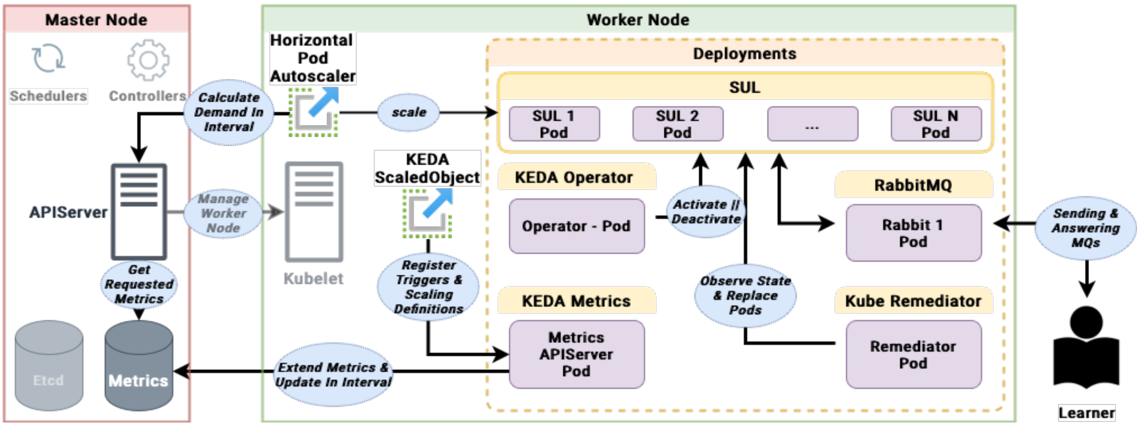

Figure 3.1: Simplified Overview Of the Architecture

The constructed architecture of this work, on which scale-able active automata learning processes are run, is illustrated in Figure 3.1. As stated in the previous chapter, the underlying cluster is managed by K8S and uses Docker as the container engine to manage and run its containers. Therefore, this architecture requires the SUL to be containerized (as a Docker image) for learning a behavioral model of the SUL. However, further consideration must be made before having a feasible solution to perform scale-able active automata learning, especially regarding real-life applications.

3.1 Membership Query Distribution

The usage of self-contained SUL containers makes direct communication between the learner and the SULs impossible. Furthermore, due to auto-scaling, the number of active SULs varies at any time. Thus, it requires a mediator who knows about each SUL to

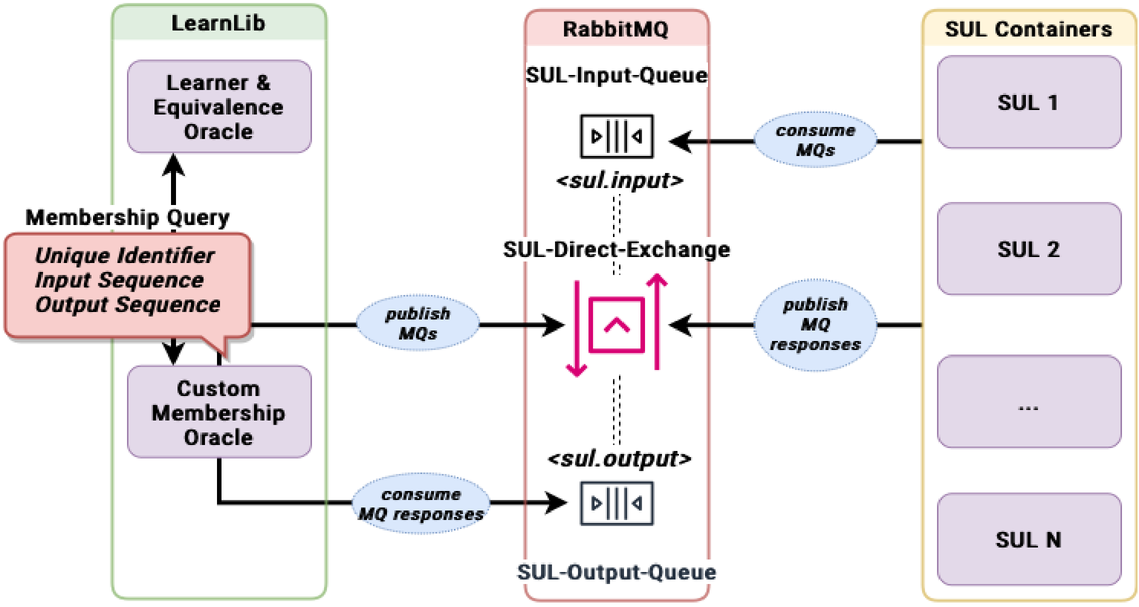

Figure 3.2: Learner and SUL communication with RabbitMQ

handle the communication - exchanging MQs - between the two partakers. Moreover, it is required to handle communication in such a way that it is ensured that every SUL has a turn - namely, load balancing. In addition, the MQ distribution should be performed asynchronously to reduce additional overhead, e.g., wait time until a SUL receives a query. Hence a message broker is used to close the communication gap.

A message broker is a means to connect and scale a modular architecture in a distributed system by allowing communication between the parts of the architecture in question - in this case, the SULs and the learner - via so-called "message queues." Moreover, a message broker offers a common infrastructure for systems to produce and consume messages. Usually, it is based on the publish/subscribe paradigm.[54] In the context of automata learning, it is critical that such a message broker is reliable, stable, and does not introduce a noticeable overhead while exchanging MQs from the learner to the SULs and back. For this reason, the message broker RabbitMQ[22] is a suitable choice for fulfilling the stated requirements.[43]

RabbitMQ is an efficient and scale-able open-source message broker acting as a communication platform between independent applications using an Erlang-based Advanced Message Queuing Protocol (AMQP)[3], enabling asynchronous processing of data. The RabbitMQ infrastructure is built around messages, message queues, exchanges, and route bindings.[36] The core of RabbitMQ is the concept of a message, which may contain any data. An application that sends a message through RabbitMQ is called a publisher, whereas the message receiver is called a consumer. An application may be a publisher as well as a consumer. A message always goes through an exchange, acting as a router, accepting and sending the message 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 connecting 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 act upon them.

Since communication during the learning process between the learner and the SULs goes through RabbitMQ, it is essential to cover the existing RabbitMQ communication infrastructure used (illustrated in Figure 3.2). Despite the ability to create complex topologies with RabbitMQ, the infrastructure utilized is as simple as RabbitMQ can get. At its core is a direct exchange ('SUL-Direct-Exchange') through which MQs are passed to message queues. On the one hand, the learner publishes unprocessed MQs, accompanied by the routing key 'sul.input', which are distributed to the message queue 'SUL-Input-Queue,' from which SULs consume MQs for processing. On the other hand, SULs publish processed MQs, holding the routing key 'sul.output', to the message queue 'SUL-Output-Queue,' from which the learner consumes and constructs its hypothesis of the SUL. Consequently, the learner and SUL act as publishers and consumers of messages at the same time.

```
Input: $queries \leftarrow list\ of\ unprocessed\ queries,\ sentQueries \leftarrow empty\ hashmap$
Output: $queries$
for all $query \in queries$ do
    $uid \leftarrow generate\ random\ unique\ identifier$
    $mq \leftarrow new\ MembershipQuery(uid, query)$
    $add\ pair\ (uuid, query)\ to\ sentQueries$
    $publish\ mq\ to\ 'SUL-Direct-Exchange'\ and\ routing\ key\ 'sul.input'$
end for
while $sentQueries\ is\ not\ empty$ do
    $response \leftarrow consume\ message\ from\ 'SUL-Output-Queue'$
    $query \leftarrow get\ and\ remove\ query\ with\ key\ response.uid\ from\ sentQueries$
    $query.output \leftarrow response.output$
end while
```

Algorithm 3.1: Simplified RabbitMQ Membership Oracle

Even though the infrastructure is simple, further adjustments must be made. A custom MO must be implemented and utilized by the learner that can produce and consume MQs RabbitMQ messages, as shown in 3.1. Further considerations must be made regarding the custom MO due to the asynchronous nature of RabbitMQ. It is not guaranteed that the same order in which the learner publishes the MQs is adhered to when the learner consumes responses to the published MQs. As a consequence, the structure of the MQ is extended by a Unique Identifier (UID) which is used to match a consumed response from any SUL to the appropriate MQ. As shown in 3.1, each query in a batch about to be published is extended by a generated UID, thus making each query sent unique and, therefore, enabling the assignment of each response to the correct query.

3.2 Challenges Of Real-Life Applications

Generally, performing active automata learning to learn a target system, especially regarding real-life applications, implicate challenges. In section 2.1, few of these were mentioned. One challenge, in particular, the requirement to guarantee each query's independence, is of interest in this work. The first option is to execute a reset procedure on the target system. This functionality is commonly unavailable in real-life systems. Secondly, 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 functionality is required. [46]

However, a direct reset or one through abstraction may sometimes be impossible. Thus, this work performs all experiments with a type of SUL that reboots itself after processing one query. Therefore, for each published MQ, a SUL container is created to process it. This type of SUL is further called 'disposable'-SUL. This approach induces other challenges while performing a learning process, especially regarding auto-scaling.

3.3 Orchestrating Containerized SULs

Containers are not run directly on the cluster itself because K8S encapsulates containers in its basic unit called a pod. A pod can contain one or multiple containers. In this work, a pod is limited to one container; therefore, a pod is the same as one SUL container. In addition, a pod includes storage resources, a network IP, and other configurations on how the container(s) are run. The requested and limited computation resources (CPU and memory) can be set upon creation.

As mentioned above, scheduling is one of the main tasks a master node has to do. The default scheduler of K8S uses the information (e.g., storage and CPU limit) about each pod to make decisions about pod placement the to ensure that not a single node in the K8S cluster exceeds its capacity of resources. As a result, pods surpassing their storage limit are terminated, but in contrast, surpassing the CPU for a short time does not lead to immediate termination of the pod.

It is essential to know how K8S manages the container life cycle to understand the challenges introduced by using disposable SULs (established in section 3.2) and auto-scaling (discussed in section 3.4). Each container in a pod can be in one of the following states:

- **Waiting**: Still executes instructions to complete the container start-up

- **Running**: The container runs without problems.
- **Terminated**: The container ran to completion, failed, or is forced to terminate

Furthermore, it is possible to run instructions before entering a specific state through life cycle hooks. For example, the 'preStop' hook can be configured and run before a container enters the terminated state.

To conclude the life cycle management of containers, it is essential to note that each container adheres to a restart policy with limited options. The restart policy defines when to restart the containers in the pod. It becomes effective when the pod fails, succeeds, or is in an undefined state and therefore exits. The pod 'restartPolicy' field is configurable and can accept one of the following values: 'Always,' 'OnFailure,' or 'Never' (default: 'Always'). If the 'restartPolicy' becomes effective, containers enter a restart loop with an exponential back-off delay which is capped at five minutes. This delay is reset once a container runs for 10 minutes without issues. [14]

For managing instances of pods, K8S offers different resources such as deployments and jobs. This work uses deployments to manage the SUL pods while learning. Deployments define long-running tasks supported by the K8S auto-scale feature. Furthermore, it creates a replica set that, in turn, creates the desired initial number of replicated pods (namely replicas). An image can be set, which is used to create the container inside each replica. The deployment maintains the desired number of pods specified or adjusted through auto-scaling. In addition, CPU and memory requests and limits for each pod can be set in a deployment.

It is important to note that a pod in a deployment supports only the restart policy with the set value 'Always' - this value is not configurable. This default policy results in pods continuously restarting and entering a restart loop when the pod fails or runs to completion. By the design of a deployment, the restart loop ensures that a pod is available at any time. However, the exponential back-off delay, which is also not configurable - leads to a significant downtime between answering queries using disposable SULs that terminate themselves after processing one query. Consequently, each SUL constantly enters the restart loop.

K8S does not offer a specific pod management resource for the unique circumstances of this work. However, deployments are still practicable, even using disposable SULs. The following options may achieve this:

1. Implementing a **custom K8S control plane** enabling, e.g., the configuration of the restart policy in deployments and replacing the default K8S control plane
2. Instead of using K8S resources like deployments to orchestrate containers, implementing a **custom pod management logic** deployed as a pod that handles creation, termination, and scaling through the use of the K8S API

3. Observing each pod's state in deployment and **replacing the pods upon reaching a specific state**

For this work, the decision lies on the third option because such an application already exists. For this purpose, the application Kube Remediator[15] is used to enable the use of disposable SULs by reducing the downtime significantly by replacing each container upon reaching its restart loop.

In conclusion, K8S offers many features and options to orchestrate containerized applications designed for long and short-running tasks. This work holds special requirements which do not fit the K8S design. Through third-party tools, it is practicable to perform active automata learning on a K8S cluster even with disposable SULs.

3.4 Strategies for Scaling SULs

In a scenario in which auto-scaling is required, K8S offers two resources to auto-scale pods: Horizontal Pod Autoscaler (HPA) and Vertical Pod Autoscaler (VPA). Even though both auto-scaler types do their task without any manual intervention, therefore, doing the task automatically, they differentiate in the way on which entity the scaling is done. The VPA scales the proper computation resources each pod can access based on a resource analysis over time. In contrast to the VPA, the HPA adjusts the desired quantity of replicas based on the utilization of their assigned computation resources. [29] In the context of this work, the VPA does not result in any significant performance gain due to the low resource requirements the used SULs have as a result of simply representing a Mealy machine. Because the actual goal is to process as many MQs as possible simultaneously, therefore, scaling up and down the number of SUL pods is required - thus using a HPA.

3.4.1 Horizontal Pod Autoscaler

The HPA scales a target K8S resource, e.g., deployment, by adjusting the desired replica count in the target resource; therefore, the HPA is not working directly with the pod replicas. In the example of a deployment, the maintenance of the replica count is done by the deployments replica controller, as shown in Figure 3.3. By default, only two basic computation resources are available as a metric to auto-scale the replica count: CPU and memory. [48] HPA scales based on the average utilization, across all replicas, of the chosen metric and triggers a replica count update through the following equation[11]:

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

Furthermore, using standard HPA, it is impossible to scale the replica count to zero, disabling the deployment to save resources. A simplified overview of the default K8S auto-scaling process, including where the aggregated metrics are pulled from, is illustrated in Figure 3.3.

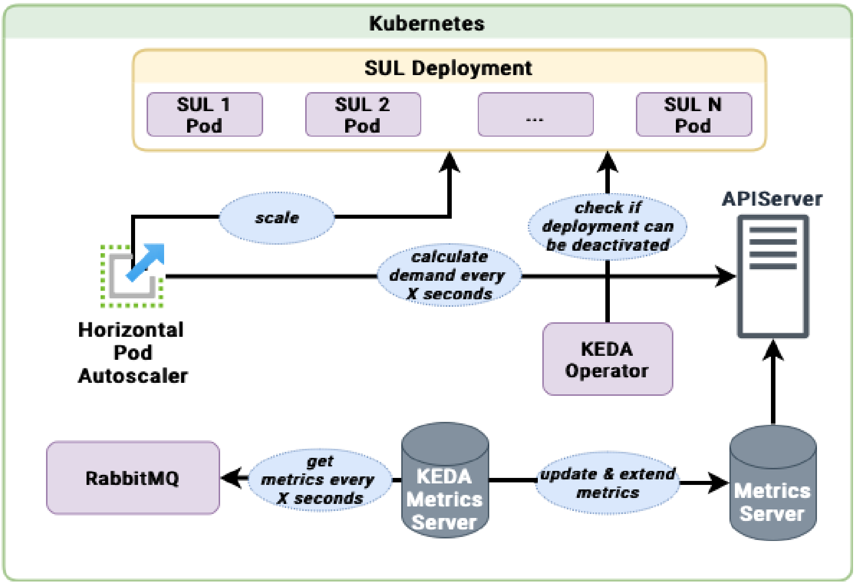

Figure 3.3: Simplified overview of the KEDA auto-scaling process.

HPA updates its metric values every 15 seconds (default value). These metrics are aggregated through the metrics server, which retrieves its information from every node in the K8S cluster. The aggregation is done every 60 seconds by default. Consequently, with the default settings in mind, there may be a time gap of 45 seconds in the worst case between identifying that the target should be scaled up and the scaling itself. As a result, HPA can miss workload peaks in a time range of seconds that might occur now and then. Furthermore, the calculation is based on the average utilization of the target metric aggregated over all pods resulting in missing out on workload peaks that occur every second. [55]

However, as mentioned above, HPA only offers, by default, two basic pod metrics to calculate the current demand for pod replicas. Both options are not suitable for guaranteeing the availability of a SUL for each MQ sent by the learner. Consequently, each SUL replica only processes one MQ at a time resulting in steady resource consumption for each replica. A more appropriate auto-scaling strategy is for each MQ in a specific RabbitMQ queue; one SUL replica is spawned to process that MQ. Consequently, the available metrics to the HPA must be extended by RabbitMQ-specific metrics.

As a result, a third-party solution, namely Kubernetes Event-driven Autoscaling (KEDA)[16], is utilized to enable auto-scaling for the chosen event-driven architecture: RabbitMQ.

3.4.2 Kubernetes Event-driven Autoscaling

KEDA is an extensible, lightweight component. It can be attached to a K8S cluster without disturbing or replacing K8S own auto-scaling resources like the HPA, therefore, being a secure option to use on any K8S cluster by coexisting with other auto-scaling options. It acts more like an extension to the standard HPA and works with it under the hood.

KEDA offers a multitude of auto-scalers regarding event-driven infrastructures using message brokers (e.g., RabbitMQ) or similar tools. Therefore allowing auto-scaling of any K8S resource based on the exposed metrics of each event-driven infrastructure. In the case of RabbitMQ, the following metrics (are 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

Figure 3.4: Extract of a KEDA ScaledObject

```
spec:
    scaleTargetRef:
        apiVersion: apps/v1
        # K8S resource type of the target to scale
        kind: Deployment
        # The name refers to the target K8S resource
        name: sul
    # Interval to update metrics
    pollingInterval: 1
    # Min. replica count to be maintained
    minReplicaCount: 0
    # List of auto-scale triggers
    triggers:
    # Type of auto-scaler
    - type: rabbitmq
        metadata:
            # Protocol used to scrape the metrics
            protocol: amqp
            # Possible metrics: QueueLength or MessageRate
            mode: QueueLength
            # Threshold on which to trigger
            # Message count or Publish/sec. rate
            value: "1"
            # Target RabbitMQ message queue
            queueName: sul_input_q
```

An used KEDA configuration file in the learning setups (section 4.5), which illustrates how a KEDA ScaledObject and its available fields can be configured, is shown in figure 3.4 (more options and details in [16]). Finally, a ScaledObject offers the modification of the HPA behavior through the use of advanced fields in the configuration file. By default, scaling down and up can occur every 15 seconds. However, it is possible to independently change the interval for scaling up and down. Moreover, there is a difference in the way scaling up and down works. Scaling down considers a stabilization window (default: 300

seconds) in which the desired state is inferred by choosing the highest value in a list of previously calculated desired states in the set interval, e.g., in the last 5 minutes. This window prevents a heavy alternating of the replica count and especially removing pods to create the same number of pods in the next moment. In contrast to that, scaling up has, by default, no stabilization window.[12] At last, limiting the number of pods terminated or created through scaling is also configurable.

Auto-scaling may not work for every learning setup, especially regarding scaling down replicas. It has to be ensured that a created SUL which consumed a MQ, is terminated after it finished processing. 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 that the SUL is terminated before processing the MQ. As a result, scenarios can occur in which MQs are passed from SUL to SUL, hindering the learning process.

K8S offers features configurable in the deployment file, solving this issue. As mentioned above, it is possible to configure a 'preStop' life cycle hook. It runs before the container is terminated. Through this hook, it is possible to delay the termination until the SUL in question completes processing the MQ. However, this approach requires that the container offers a shell environment and eventually read/write access (based on the implementation of the hook) on the K8S cluster it runs on. The native image created with GraalVM offers, by default, no shell. Therefore, this approach is not suitable.

In this work, the following approach is taken. Pods about to be terminated are going through a grace period, allowing for a graceful termination. Meaning K8S allows the pod to shut down, e.g., RabbitMQ connection, or let it finish processing and terminate itself before being forcefully killed by K8S. The grace period is, by default, 30 seconds. However, the grace period has to be adjusted while using applications that run heavy processing tasks that exceed this boundary.

To summarize, auto-scaling active automata learning with the design choices made in this work is achievable through the dedicated event-driven auto scaler KEDA. Therefore, having one SUL replica ready for each MQ in a specific RabbitMQ queue results in more efficient use of computational resources than the fixed number of SUL replicas. In contrast, the auto-scaling strategy might be slower regarding the total execution time. Furthermore, auto-scaling implicates additional management effort regarding the container life cycle depending on the learn setup.

Chapter 4

Evaluation Method

The architecture explained in chapter 3 and shown in Figure 3.1 has been set up and is used to perform the active automata learning setups introduced in this chapter. The architecture is constructed as a single node cluster with the help of MicroK8s[17] and is equipped with the following hardware - 126 GB of RAM and an AMD EPYC™ 7443P CPU with 24 cores @ 2.85 GHZ with 48 threads (more details on [1]).

4.1 Experiment: Simple Coffee Machine

Evaluation of the constructed architecture regarding practicability, performance, autoscaling, and limitations is done by executing active automata learning processes. In section 2.2, the Mealy machine representing a simple coffee machine, was introduced. This Mealy machine is containerized and used as a disposable SUL in the different learning setups to evaluate the architecture.

Moreover, as mentioned in subsection 2.1.1, LearnLib offers various algorithms and filters regarding the learning process. The learner used in the learning setups runs the learning process with the following LearnLib components:

- **Query cache** that reduces the total sent queries by filtering out queries that were already sent

- The **Equivalence Oracle** simulates the behavioral model of the SUL; thus, no approximation of EQs takes place, therefore reducing the total amount of sent queries significantly.

- LearnLib offers many learning algorithms. In this work, the **Direct Hypothesis Construction (DHC) learning algorithm** is used because it supports query batches enabling the distribution of queries for simultaneous processing. It favors big batch sizes compared to the other learning algorithms available in LearnLib, such as $L*$. [46]

As a result, the following metadata of the Mealy coffee machine in use is illustrated in Table 4.1.

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

Table 4.1: Coffee Mealy Machine Statistics

4.2 SUL Containerization

To orchestrate SUL instances with K8S, turning the SUL application into a container on which the experiments are based is required. Although creating a container is not complicated when done with a tool like Docker, the generated overhead in the process can vary significantly based on the binary and dependencies required to run the SUL. Reducing the overhead is vital in this work because many containers are expected to be terminated and new ones created over time when using auto-scaling and disposable SULs. Consequently, the underlying foundation of the SUL container determines the impact on:

- **Processing time** of a MQ
- **Scale-ability** of the SUL, including the time it takes to create, start-up, and terminate a container
- **Deployment time** can vary by the size of the container

Thus it can have quite an impact on the total execution time and practicability of a learn setup. Therefore, crucial details about the implementation of the SUL representing the simple coffee machine are as follows:

- **LearnLib** is used to implement the coffee machine's behavioral model represented as a Mealy machine
- The language that is used is **Java**[13]. As a result, the SUL runs by default on a **Java Virtual Machine (JVM)**[67]

- Through the use of the framework **Spring Boot**[23] or **Spring Native**[4] (explained below) to trivialize the creation process of an application that is runnable, and the use of third party libraries

- The **RabbitMQ** third-party library for the Spring Boot framework

These details may notably impact image size, application start-up time, and, therefore, the total execution time of a learning process.

The Docker container layer wrapped around the application itself adds a not noticeable overhead of approximately 0.036s on the container creation time, as stated by the author of [38].

The size of the Docker image is a concern when using Docker image registries, where the image is pulled from to create containers. A smaller image size results in a faster load speed from the registries in question and thus leads to faster container creation. Table 4.2 illustrates the variance a coffee machine SUL image size can have used different base images required to run the coffee machine and its dependencies. At last, the start-up time of the SUL itself determines the time it takes until the SUL is ready to accept MQs. As shown in Table 4.3, the time it takes to start up the coffee machine SUL itself can fluctuate and vary a lot based on the implementation, how the executable is built and how the Docker image is created. In this work, a significant reduction of the start-up time and image size could be achieved by building a native image with GraalVM. Therefore, reducing the overall overhead induced by containerization and of the SUL application itself. Thus, the native image is used to create coffee machine SULs in a learning process.

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 4.2: Image Size Variance: Example Coffee Machine

Base Image	**Average Start Up Time**
adoptopenjdk /openjdk11:alpine-jre	**3.1796s**
native-image (GraalVM)	**0.0982s**

Table 4.3: Start-Up Time Coffee Machine SUL - JVM vs. GraalVM

4.3 Simulating Real-Life Systems

The processing time of a MQ regarding the Mealy machine implementation is near instant, in contrast to real-life applications, which may take noticeably longer, even seconds. Consequently, adjustments are made in the learning processes to accommodate real-life systems, although it does not replace them.

The accommodation to real-life systems is an artificial delay to each 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, resulting in the wait time before the MQ is processed. The time interval used in the calculation is configurable, and various values are used in this work. See section 4.5 for more details on the values used. This approach simulates real-life applications designed to process heavy tasks that take a long time.

As introduced in section 3.2, using the disposable SULs accommodates real-life systems that do not offer functionality or take a long time to reset, fulfilling the requirement of MQ independence.

Both approaches merged cover complex and extensive systems that may include many services and persistent storage solutions that take longer to process queries and often do not offer functionality to reset the whole system. These systems are commonly built so that a reboot is needed to set the system to its initial state because they are not primarily built for active automata learning.

4.4 Data Acquisition

In this work, for each execution of a learn setup (explained in section 4.5), data about the SUL and the learning process is collected. Besides the already mentioned details about the coffee machine mentioned in Table 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)
- **Time** it takes **to process a batch** of MQs (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

Furthermore, 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. Additionally, it is ensured that the executed learning process sends the same queries in the same order for each learning setup.

4.5 Learning Setups

The constructed architecture allows configuring of various parameters of a learning setup, such as:

1. Configuration of the **SUL deployment** including:

    - The **SUL image** that should be used. Determines the behavior of the SUL, e.g., coffee machine
    - **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:

    - **SUL instance count range limit** that the auto-scaling does not exceed or undercut
    - 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 this work, the learning setups can be differentiated into two categories. The first category of learning setups uses a constant number of disposable SULs. The following values are used: 1, 2, 4, 8, 16, and 32. The purpose of this category is to highlight the scalability and practicability of the approach taken.

The second category uses auto-scaling with KEDA based on the queue length of the SUL input queue. For each MQ in the queue at the time, the new demand is calculated by the HPA. In this case, KEDA is configured, so the metrics are updated each second. The following HPA demand calculation intervals are used: 5s, 15s, and 30s. The different HPA values should highlight that careful configuration may have an impact. Furthermore, this category is run with zero and one initial SULs.

Before the learning process of a learning setup is executed by sending the first batch of queries, it is ensured that the mentioned initial number of SULs are deployed and are ready to receive queries.

The termination grace period for each learning setup is set to 10 seconds, and for more extensive delays, 60 seconds. These values are sufficient for the learning setups in this work, guaranteeing that each MQ is processed by the SUL before it is forced to terminate through K8S.

As mentioned in section 4.3, an artificial delay is added to the processing time of each SUL. The following intervals are used and accumulated over each input in the MQ: 100-150ms, 500-1000ms, 3000-5000ms, 5000-10000ms.

Finally, the evaluation baseline used in chapter 5 for comparison is the learning setup with one disposable SUL. Furthermore, both categories mentioned above are compared to each other regarding total execution time and resources spent.

Chapter 5

Evaluation Results

In this section only the summary of the results are shown. The complete overview of results, including all details, can be found in —.

5.0.1 Constant Number Of SULs

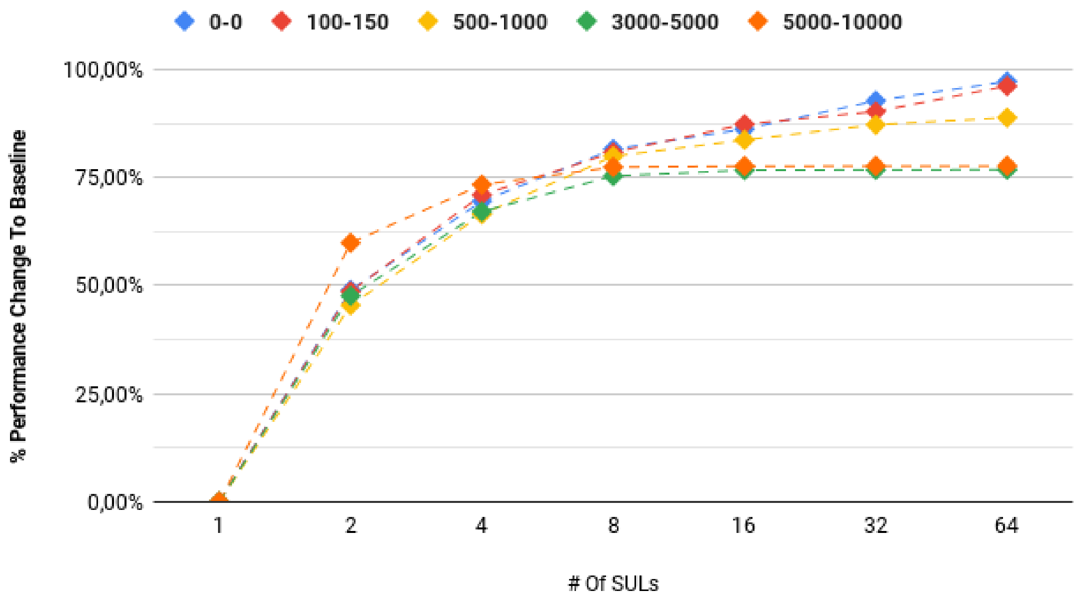

Figure 5.1: –

Figure 5.1 illustrates, that applying parallelization even with disposable SULs is very effective to improve the total learning time. It is possible to gain nearly double the performance when using systems that respond fast (low delay). However, the more SULs are active at the same time the performance gain reduces significantly. While the increase from one SUL to two SULs increased the performance by 50%, getting from eight to sixteen SULs only improves the performance by a additional 5-6% compared to the base line. ?? This effect is even more prevelant when using high delay. ?? highlights that the performance gain gets smaller when adding more delay. Using sixteen suls instead of 8 improves the performance by under 1% when using a delay 5000-10000ms. The bigger the delay the smaller the performance gain of adding more suls.

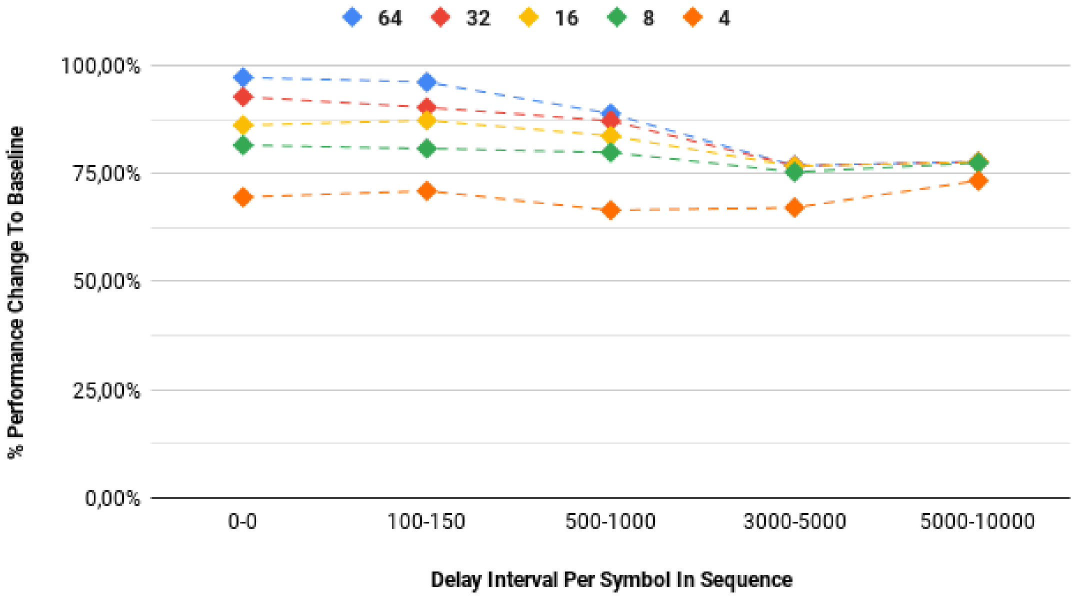

Figure 5.2: —

5.0.2 Auto-Scaling

good example: A performance comparison of cloud-based container orchestration tools

While running the experiments the SULs had no CPU or memory limit. They ran on average with 300-400m CPU and 50-100mbit memory. creation time termination time

rabbitmq performance and resource consumption

ab wann flacht es ab?

Coffee Machine 118 Queries 61 Batches Min Batch Size 1 Max Batch Size 4

JVM vs GraalVM

Nevertheless, it is essential to remember that using these tools increases the overhead of the overall learning setup, which is evaluated in chapter 5.

———————-

5.1 Preliminary Considerations

Performing active automata learning on a clustered system by parallelizing and distributing the MQs to numerous running SUL instances where the MQs get simultaneously processed

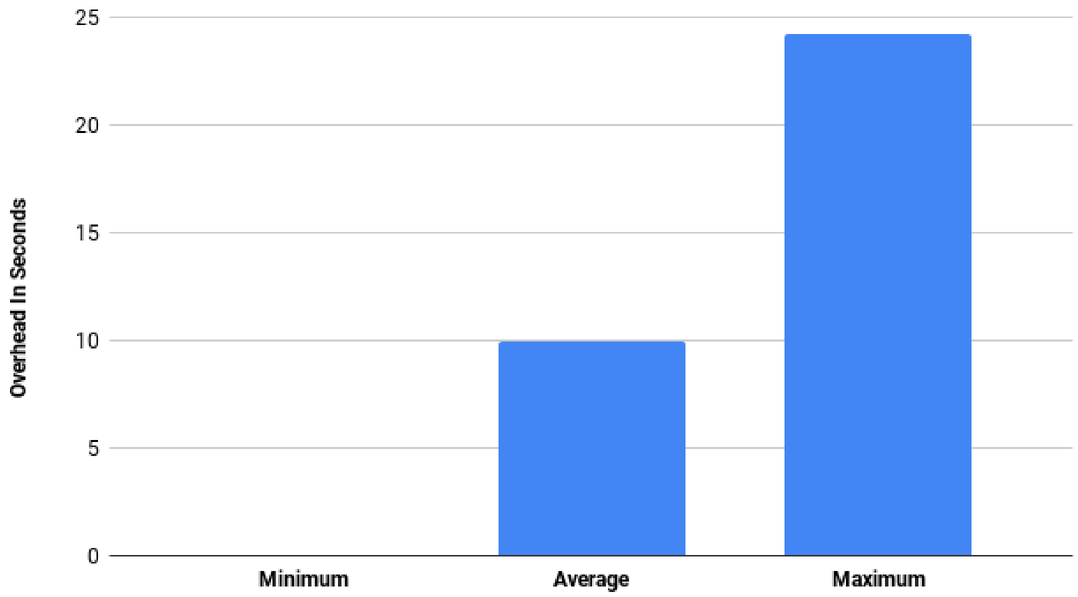

Figure 5.3: —

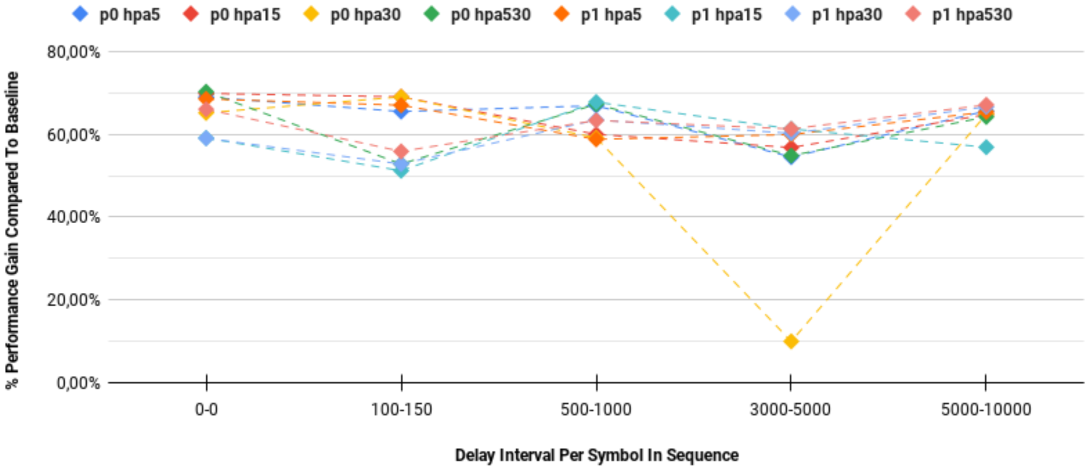

Figure 5.4: —

can lead to a notable reduction in the total execution time. Depending on the internal/external factors of the actual learning setup, e.g., the complexity, time to reset, processing time, and the start-up time of the SUL, the performance is expected fluctuate.

Furthermore, it depends on the constructed architecture specification and the clustered system used for the learning process. Moreover, increasing the SUL instance count does not always result in better performance. The reason is that the overhead of the used technologies grows as the container management task increases in complexity or several

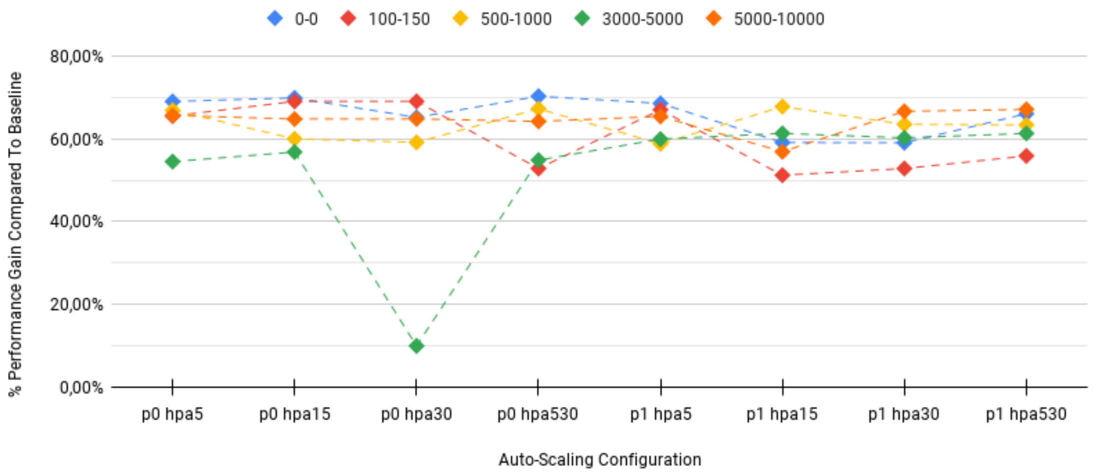

Figure 5.5: —

SULs are fast enough so that each addition might result in an idle SUL and no performance gain.

Another reason is that the MQs are sent in batches, and each batch has to be fully processed until the next batch of MQs is sent. Thus there is a constraint on the performance dependent on the query batch size the learner sends. Constructing a learner that sends queries asynchronously and independently from each other is difficult because each batch of MQs is sent depending on the learner's constructed hypothesis model.

Automata learning itself induces challenges when used in the context of real-life systems. In contrast to experimental systems, e.g., coffee machines explained in section 2.2, real-life systems are often more complex. Such systems often need longer to process queries and take longer to reset to their initial state (ensuring the independence of queries). These systems may not even have the functionality to reset them into their initial state. Furthermore, larger systems may need thousands of queries to be learned, which is solvable by utilizing more resources/computing power when available (is subject to resource constraint). All these challenges and constraints come together and add up to a varying increase in the total execution time, resulting in learning that may take too long.[46]

Finally, the number of SUL instances is limited by the resource consumption of each instance and the available resources of the clustered system in question. Consequently, all those constraints must be considered when constructing a learning setup and also can affect the total execution time of a learning process ———

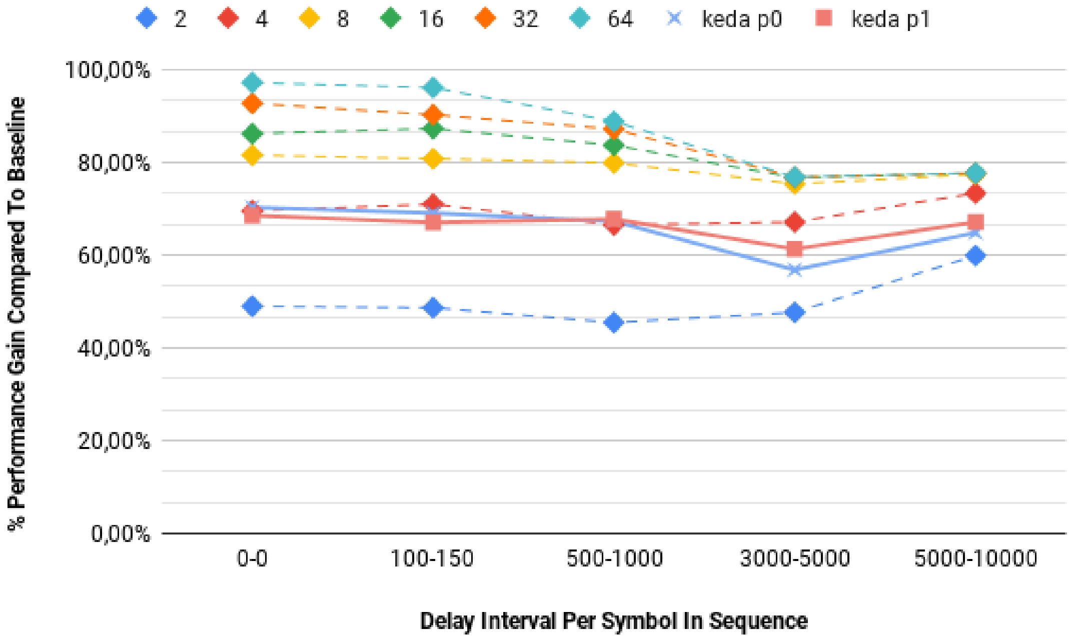
2
4
8
16
32
64
keda p0
keda p1
100,00%
80,00%
60,00%
40,00%
20,00%
0,00%
% Performance Gain Compared To Baseline
0-0
100-150
500-1000
3000-5000
5000-10000
Delay Interval Per Symbol In Sequence

Figure 5.6: —

Chapter 6

Future Research

6.1 Learning Real-Life Systems With Mappers

Although this work tries to get close as possible to real-life systems through using disposable SULs and additional delays, this approach does not replace them. The paper [59] introduces so-called "mappers" that serve as an interface for communication with real-life systems such as web applications. The tool ALEX[2] utilizes this concept to perform active automata learning with LearnLib on web applications.

Opening up a way to configure sul mappers on top of the architecture enables the usage of real-life systems as SULs. Another approach would be to apply the architecture discovered in this work to the tool ALEX. There is more to evaluate regarding real-life systems such as termination time of a system, that is not taken into account in this work.

6.2 Custom Autoscaler

The utilized auto-scaling strategy based on the queue length is limited by the query batch size sent by the learner. As illustrated in **??**, using more disposable SULs as the current amount of queries in the queue results in a significant performance improvement. KEDA does not offer, e.g., to scale up two SULs for each additional query in the queue. A 1:1 query to SUL mapping is the lowest it offers. The following approaches can be taken:

1. KEDA offers a way to implement custom autoscaler

2. Custom application using k8s api that for more fine grained control

removing overhead through keda + kube remediator

6.3 Asynchronous Sending of Membership Queries and Equivalence Queries

Right now the learner has to wait till a batch is processed until new queries are sent. Which limits the possbile performance improvement by the max. batch size sent by the learning algorithm based on the available inputs.

Chapter 7

Conclusion

batch size bottleneck - the more the sul takes up time to respond the more it is noticable low respond time -> more suls better high respond time -> suls = max. batch size -> is more efficient overhead impact is less noticable with more suls optimizing containers 1:1 mapping has constant good performance gain while more not always better the more delay

7.1 Future Research

Learning Real-Life Systems With Mappers Although this work tries to get close as possible to real-life systems through using disposable SULs and additional delays, this approach does not replace them. The paper [59] introduces so-called "mappers" that serve as an interface for communication with real-life systems such as web applications. The tool ALEX[2] utilizes this concept to perform active automata learning with LearnLib on web applications.

Opening up a way to configure sul mappers on top of the architecture enables the usage of real-life systems as SULs. Another approach would be to apply the architecture discovered in this work to the tool ALEX. There is more to evaluate regarding real-life systems such as termination time of a system, that is not taken into account in this work.

The utilized auto-scaling strategy based on the queue length is limited by the query batch size sent by the learner. As illustrated in **??**, using more disposable SULs as the current amount of queries in the queue results in a significant performance improvement. KEDA does not offer, e.g., to scale up two SULs for each additional query in the queue. A 1:1 query to SUL mapping is the lowest it offers. The following approaches can be taken:

1. KEDA offers a way to implement custom autoscaler

2. Custom application using k8s api that for more fine grained control

removing overhead through keda + kube remediator

Asynchronous Sending of Membership Queries and Equivalence Queries Right now the learner has to wait till a batch is processed until new queries are sent. Which limits the possbile performance improvement by the max. batch size sent by the learning algorithm based on the available inputs.

Appendix A

Additional Info ...

Appendix B

Index Of Abbreviations

DHC	Direct Hypothesis Construction	23
SUT	System Under Test	1
SUL	System Under Learning	1
FSM	Finite-State Machine	1
MO	Membership Oracle	7
MQ	Membership Query	3
MBT	Model-Based Testing	1
MAT	Minimally Adequate Teacher	5
EO	Equivalence Oracle	7
EQ	Equivalence Query	3
AMQP	Advanced Message Queuing Protocol	14

VM	Virtual Machine	9
OS	Operating System	9
HPA	Horizontal Pod Autoscaler	18
cgroups	Control Groups	9
COS	Container-based operating system	9
K8S	Kubernetes	11
CPU	Central Processing Unit	9
UID	Unique Identifier	15
JVM	Java Virtual Machine	24
VPA	Vertical Pod Autoscaler	18
KEDA	Kubernetes Event-driven Autoscaling	19
ALEX	Automata Learning Experience	2
NGLL	Next Generation LearnLib	2

List of Figures

Index Of Algorithms

Bibliography

[1] Amd epyc™ 7443p. https://www.amd.com/de/products/cpu/amd-epyc-7443p. Accessed: 2022-10-04.

[2] Automata learning experience (alex). https://learnlib.de/projects/alex/, note = Accessed: 2022-09-14.

[3] Cloudamqp official website. https://www.cloudamqp.com/, note = Accessed: 2022-03-31.

[4] Compiling spring applications to native executables. https://docs.spring.io/spring-native/docs/current/reference/htmlsingle/, note = Accessed: 2022-09-14.

[5] Container-based operating systems virtualization - docker. https://www.docker.com, note = Accessed: 2022-09-14.

[6] Container-based operating systems virtualization - lxc. https://linuxcontainers.org, note = Accessed: 2022-09-14.

[7] Container-based operating systems virtualization - singularity. https://docs.sylabs.io/guides/3.5/user-guide/introduction.html, note = Accessed: 2022-09-14.

[8] Container orchestration tool - apache mesos. https://mesos.apache.org, note = Accessed: 2022-09-14.

[9] Container orchestration tool - docker swarm. https://docs.docker.com/engine/swarm/, note = Accessed: 2022-09-14.

[10] Container orchestration tool - kubernetes. https://kubernetes.io, note = Accessed: 2022-09-14.

[11] Container orchestration tool - kubernetes - hpa. https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/, note = Accessed: 2022-09-29.

[12] Horizontal pod autoscaling. https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/#configurable-scaling-behavior. Accessed: 2022-10-04.

[13] Java official website. https://www.java.com/de/. Accessed: 2022-10-04.

[14] K8s pod lifecylce. https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/. Accessed: 2022-10-04.

[15] Kube remediator. https://github.com/ankilosaurus/kube_remediator. Accessed: 2022-10-04.

[16] Kubernetes event-driven autoscaling. https://keda.sh, note = Accessed: 2022-09-14.

[17] Kubernetes setup tool - microk8s. https://microk8s.io, note = Accessed: 2022-09-14.

[18] Learnlib official website. https://learnlib.de/. Accessed: 2022-03-31.

[19] Learnlib official website - performance. https://learnlib.de/performance/. Accessed: 2022-10-04.

[20] libalf official website. http://libalf.informatik.rwth-aachen.de. Accessed: 2022-10-04.

[21] Most loved and dreaded technologies. https://survey.stackoverflow.co/2022/#most-loved-dreaded-and-wanted-tools-tech-love-dread. Accessed: 2022-10-04.

[22] Rabbitmq official website. https://www.rabbitmq.com/. Accessed: 2021-11-21.

[23] Spring boot official website. https://spring.io/projects/spring-boot. Accessed: 2022-10-04.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

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

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

[35] 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.

[36] 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.

[37] Marco Henrix. *Performance improvement in automata learning*. PhD thesis, Master thesis, Radboud University, Nijmegen, 2015.

[38] 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.

[39] 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.

[40] 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.

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

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

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

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

[45] Paul Menage. Control groups definition, implementation details, examples and api, 2004.

[46] Maik Merten. Active automata learning for real life applications. 2013.

[47] Maik Merten, Bernhard Steffen, Falk Howar, and Tiziana Margaria. Next generation learnlib. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 220–223. Springer, 2011.

[48] 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.

[49] 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.

[50] 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.

[51] 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.

[52] 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.

[53] Robert Rose. Survey of system virtualization techniques. 2004.

[54] 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.

[55] 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-5f00 2020. Accessed: 2022-09-29.

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

[57] 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.

[58] 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.

[59] 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.

[60] 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.

[61] Malte Isberner1 Falk Howar2 Bernhard Steffen. Learnlib: An open-source framework for active automata learning.

[62] 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.

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

[64] 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.

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

[66] Mark Utting and Bruno Legeard. *Practical model-based testing: a tools approach.* Elsevier, 2010.

[67] Bill Venners. The java virtual machine. *Java and the Java virtual machine: definition, verification, validation*, 1998.

[68] 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.

[69] 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.

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet sowie Zitate kentlich gemacht habe.

Dortmund, den October 29, 2022

Julien Saevecke