Bachelor Thesis

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

Julien Saevecke

October 4, 2022

Reviewer:
Prof. Dr. Bernhard Steffen
Julius Alexander Bainczyk

Technische Universität Dortmund
Fakultät für Informatik
Lehrstuhl V - Programmiersysteme
Fachgruppe Softwaretechnik sicherer Systeme
https://sse.cs.tu-dortmund.de

Contents

Index of Abbreviations

Index of Abbreviations

1 Introduction

These days, computer software is used in many objects in the manufactured 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. Even simple things like doorbells or light bulbs are not spared. Regrettably, even the simplest of 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.

An accident including the loss of life is the death of 18 patients by miscalculating the required radiation dosage in Panama's largest radiation therapy institution: the National Oncology Institute. Another accident was at the Hartsfield-Jackson Atlanta International Airport, where the security screeners did not realize a test was underway at the security gates. That led to evacuating the security area for two hours and delaying 120 flights.[77]

When generations grow up surrounded by technology, impacting their very lives, it is important to prevent software failure or bugs that can influence lives in a bad way or even endanger them. It is no wonder that between 30 and 60 percent of all software development process consists of testing.[74]

There are different ways to test software. 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 will be generated to test whether the implementation of the SUT is working correctly.

In MBT, generating the model can be time-consuming and cost-inefficient when done manually and vulnerable to errors. Automation of the generation of the model would negate the drawbacks of manually generating the model. The process of automatically learning a model representing the properties and behavior of the SUT can be used on various black box systems.

Automata Learning uses a learner to interact with the System Under Learning (SUL) before called SUT to generate a hypothesis model about the system by sending queries to the SUL and by processing the outputs of the SUL new inputs may be generated. The cycle repeats until no inputs are generated, and until then, the hypothesis model will be refined over and over.

A popular open source tool written in Java specialized in automata learning is called LearnLib.[17][56] Learnlib was used to learn models of various implementations. Some of these implementations are: TCP[47], SSH[71], bank cards[24] and printer software[64]. In the case study, the printer software took 11 days.[64] Depending on how various factors, such as how fast the learning setup can process outputs and the SUL processes inputs, it can take even longer.

1.1 Related Work

Active automata learning is enthusiastically researched and improved. LearnLib is in active development and often used as a tool to execute active automata learning experiments, like in the related work mentioned below. One major topic in this field is to reduce the execution time to learn a complete behavioral model of a SUL, therefore, several different methods have been considered to achieve this goal.

It is essential to know that the typical learning process of active automata learning[27] takes turns between two stages. The first stage is the learning stage, where a hypothesis model is created. The second stage is the test stage, in which a counter-example to the hypothesis model is searched for to refine the hypothesis model further. This process is implemented in LearnLib.[68]

The study [65] and [64] examined the method to reduce the overall queries sent to the SUL by modifying or replacing LearnLib components used in the learning process. In these particular studies, the focus lies in reducing the test queries, also called Equivalence Queries (EQ), sent in the test stage. The study [64] uses different test selection strategies while [65] modifies the underlying L* algorithm to tackle that goal.

Another approach uses an abstraction layer on top of the symbols sent to the SUL. This approach is taken in [25] by obtaining knowledge about the SUL beforehand and assigning each symbol to an equivalence class. This assignment can be used to determine a new reduced set of symbols. This approach results in a smaller learned model.

Reusing the inner system state of the SUL is another approach taken in [32] and [41]. This method tries to store system states of the SUL after processing input to reload this state later in the learning process when appropriate to reduce the time it takes the SUL to process a later sent query. This approach eliminates possible duplicate work that the SUL has to do.

Modifying the overall setup of the learning process is another technique used in [41] to reduce the overall execution time. The study [41] evaluates active automata learning in a multi-threading context against the usually single-threaded approach. This approach runs multiple SULs simultaneously as threads. It divides the Membership Query (MQ), these are sent in the learning stage mentioned above, and EQ before being sent into numerous batches and distributed to the running SULs to balance the load.

In the dissertation [51], the active automata learning process was moved to cloud computing in the context of real-life applications, but no in-depth evaluation was made. This thesis tries to close this gap by tackling the challenges that real-life applications implicate in the context of active automata learning and distributed systems and evaluating such a setup's performance.

1.2 Approach

The approach taken complements the scale-able aspect of the method taken in [41] and shifts the context, similar to the cloud computing approach [51], from the active automata learning process running on a single physical machine into a clustered system containing 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 thesis tries to answer the following research question:

How can a clustered system be utilized to perform scale-able active automata learning with modern technologies?

The following questions are of relevance and have to be answered to answer the primary research question:

- How can a clustered system be constructed to scale up and down replicas of a SUL based on demand?
- How can the demand for SUL replicas be calculated and used for scaling?
- Is this approach practicable, and what limitations are implied?

On that account, this work is structured in the following way. In the first part, 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. Because LearnLib and its features are used to set up the learning process and the SULs that are used - implemented as mealy machines - to evaluate the constructed architecture. 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, the architecture of the clustered automata learn setup is explained, accompanied by an introduction on how the evaluation is performed and how the experiments are set up. In conclusion, in this part, the results of the performed experiments are analyzed and evaluated. The last part of this work concludes by answering the research questions stated above. In the case of future research, recommendations are made about possible improvements, extensions, or even alternatives to the architecture or technologies used to perform the experiments.

2 Preliminaries and an Introduction to Automata Learning

Before explaining an architecture can be constructed to utilize the underlying clustered system to perform scaleable active automata learning, it is essential to concisely introduce the fundamentals of the modern 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 Machine

LearnLib offers various ways to implement and learn finite deterministic automatons, e.g., Mealy machines. [46] In this work, it is crucial to understand how Mealy machines work because the SULs used to evaluate the constructed architecture are implemented as Mealy machines. One example is shown in Figure 2.1. 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 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. An example of such a state diagram is shown in Figure 2.1.

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 (be $n = |S|$ the size of the Mealy machine),
- $s_0 \in S$ is the initial state,
- Σ is a finite input alphabet,
- Ω is a finite output alphabet,
- $\delta : S \times \Sigma \longrightarrow S$ is the transition function, and
- $\lambda : S \times \Sigma \longrightarrow \Omega$ is the output function.

The example mealy machine state diagram shown in Figure 2.1 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, 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. All other sequences do not lead to a cup of coffee.

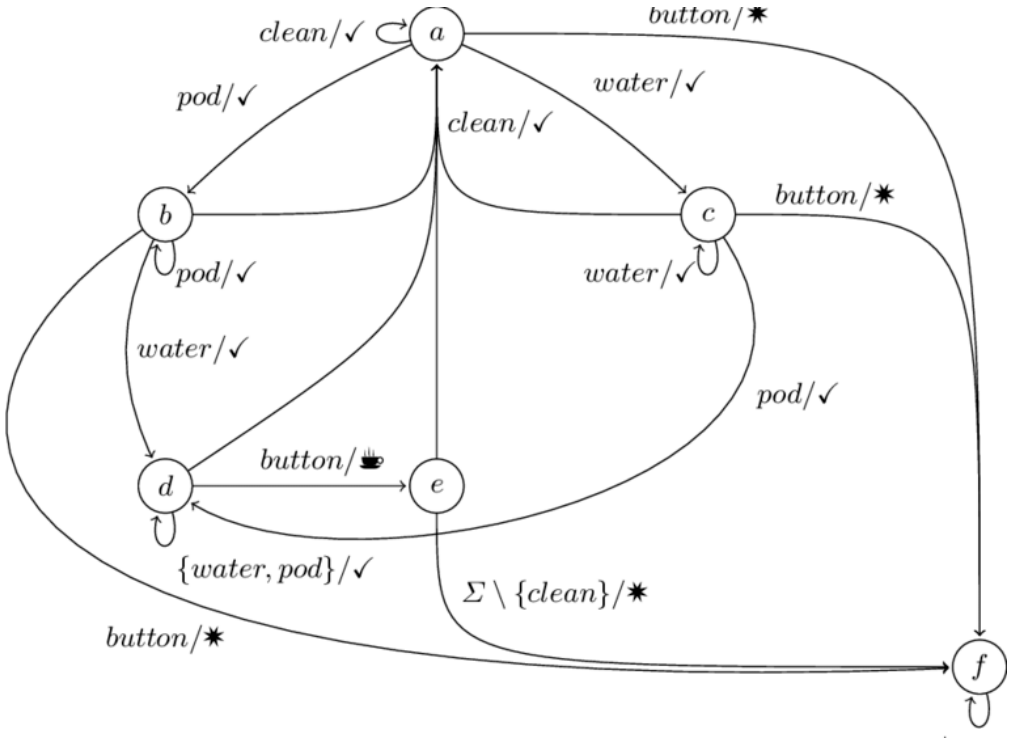

Figure 2.1: A simple coffee machine as a Mealy machine [68]

The formal definition for the coffee machine example would be:

Definition 1 *The coffee machine Mealy machine is defined as a tuple* $M = (S, s_0, \Sigma, \Omega, \delta, \lambda)$

- $S = \{a, b, c, d, e, f\}$
- $s_0 = a$
- $\Sigma = \{water, pod, button, clean\}$
- $\Omega = \{check, coffee, failure\}$
- $\delta : S \times \Sigma \longrightarrow S$ *is defined by the state diagram in Figure 2.1*
- $\lambda : S \times \Sigma \longrightarrow \Omega$ *is defined by the state diagram in Figure 2.1*

2.2 Active Automata Learning

Active Automata Learning, also called regular extrapolation, is a process that automatically infers formal behavioral models of, e.g., black box systems through testing. [69] 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. In the context of active automata learning, the learner has to figure out an unknown formal language L known by the teacher by requesting answers from the teacher on specific questions (further called queries)[56][51]:

1. **MQ** is a sequence of inputs from a predefined input alphabet given by the teacher. Based on the sequence of inputs that get processed by the teacher, leading to an answer. The learner uses this answer to construct a hypothesis model of the unknown target formal language L.

2. **EQ** is defined like an 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 the formal language L known by the teacher. This comparison may result in inequality, and a Counter-Example (CE) is provided, proofing the dissimilarity.

Using these two types of queries, the learner follows an alternating two-stage strategy to determine the formal language L (Figure 2.2). At first, the MQs are used to construct the hypothesis model of the unknown formal language, followed by testing the current hypothesis model through EQs may result in a CE. A refined hypothesis model can be constructed using the CE, which may result in new MQs and EQs. As stated, these two stages take turns until EQs no longer produce CEs. At this point, the hypothesis model is equal to the target formal language. However, this work focuses on real-world systems, often

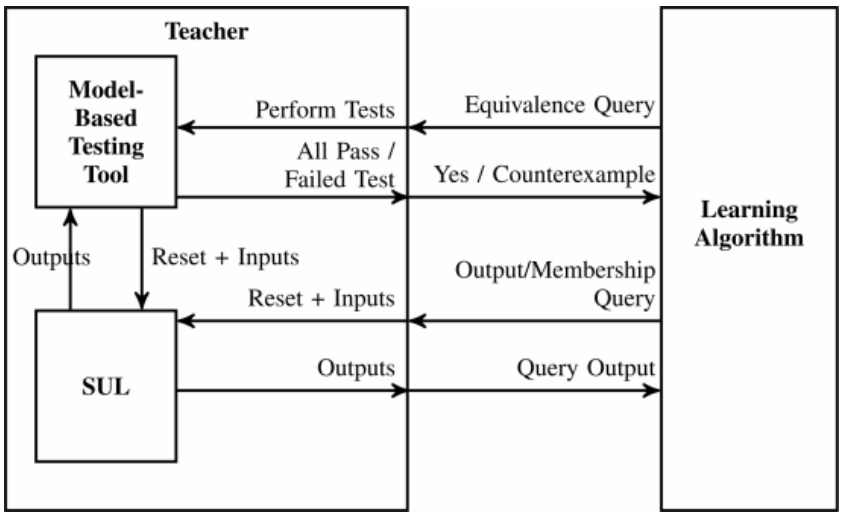

Figure 2.2: The interaction between SUL, teacher and learning algorithm based on [70] and [64]

black-box systems that introduce some challenges. The MBT shown in Figure 2.2 is not present while working with black-box systems resulting in a challenge regarding EQs. With this change, the EQs have

to be approximated and sent as MQs because the teacher does not know about the actual model resulting in no direct comparison between the hypothesis model of the learner and the actual behavioral model looked out for. The teacher only knows the answers to queries, so no directed learning process occurs. This may result in a higher MQ and EQ count than learning a non-black-box system. Other problems emerge while using active automata learning on SUL akin to real-world systems. [70]

Additionally, 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 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. [44]

Furthermore, simple input alphabets often do not suffice to communicate with real-world systems. [43] In the example of web services, the problem starts when a Representational State Transfer (REST) Application Programming Interface (API) is required to be used to communicate with them. A simple authentication request is challenging to handle with an input alphabet because such a request is sent with additional values/parameters that need to match credentials stored in the SUL. This can be solved by using the so-called mapper that represents an intermediate layer between the learner and the SUL, which translates MQs, e.g., to method invocations, and calls them instead of sending a sequence of inputs from an alphabet. [67]

There is a choice to be made when it comes to libraries enabling automata learning adhering to the explained MAT model: LearnLib[17], libalf[19], or AIDE. This thesis uses LearnLib as the library of choice. The reason is the faster performance compared to the other available choices and its flexible modular structure. [46][18]

2.2.1 Automata Learning Framework: LearnLib

LearnLib is an open-source library built in Java with the focal point being automata learning. It provides a modular design enabling flexible and extensible learning configurations and allows for capitalization of specific properties of the target SUL [56]. Additionally, the design structure accommodates making use of different infrastructures, e.g., distributed systems, while maintaining a good performance [46][51][41]. Not only does LearnLib provide interfaces to setup and modify learning scenarios for special needs, but it is also accompanied by the automata library AutomataLib[21], containing the feature to model various graph-based structures (including FSM, e.g., Mealy machines) enabled by a generic transition system.

The the primary modules of LearnLib are [56]:

1. **Learning algorithms**: The way MQ are generated and used to construct/refine the hypothesis model
2. **Equivalence approximation strategies**: Determines how the approximation of EQs works - implemented as conformance testing
3. **Filters**: Strategies to reduce the number of queries

A more in-depth active automata learning process, implemented with LearnLib and adhering to the MAT

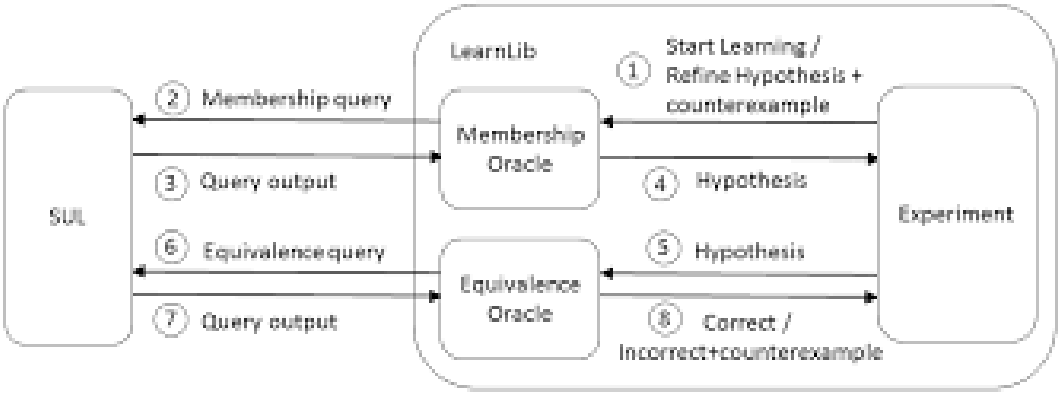

Figure 2.3: Overview Of LearnLib Algorithm [41]

model, is illustrated in Figure 2.3. It can be separated into two parts. The first 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 an abstraction to a SUL. It allows for the production of queries, namely MQs. As mentioned above, a MQ is a sequence of input symbols. However, the sequence is further divided into a prefix and suffix part in this case. Sending MQs to the target SUL, done by the oracle in question, produces an observable behavior helping to create or refine a 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 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.

- **The experiment** defines all possible interactions with the target SUL (input alphabet). Furthermore, it interacts with the MO and EO to create, refine and test the hypothesis model. At last, it contains the chosen learning algorithm, provided by LearnLib, used to create queries and determines how the hypothesis model is constructed.

In sum, this work utilizes LearnLib to move the active automata learning process into the context of a clustered system. Using the modifiable LearnLib and its provided features and the third-party AutomataLib allows the implementation of the SULs as Mealy machines and deploy a modified version of the learning process illustrated in Figure 2.3 onto the clustered system. It is a requirement to modify, e.g., the MO, to adhere to the unique requirements that a clustered system induces.

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 should be possible 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. [58][37] Virtualization is used to make resource utilization more efficient and provide more flexibility. [33] This technology can be differentiated into multiple types of virtualization. The most used techniques are hypervisor-based and Operating System (OS)-level virtualization. [29][30]

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. [58][37] This technique brings a big overhead because each VM runs its OS and file system. This overhead is why a VM is slow to launch[30][57] and should be used in scenarios where complete isolation and security, ease of management, and customization are the primary features in need.[59][37]

The OS-level virtualization technique is also known as Container-based operating system (COS) virtualization[66] or containerization[73]. 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[78] the use of namespaces[34]. Generally, the resources that each environment has access to are managed by either limiting/prioritizing through Control Groups (cgroups)[50]. The implementation may differ between the virtualization tools used. Each environment has its process ids, user identifiers, file system namespaces, and so on.[28] 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. [38][73]

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][33]

The lightweight containers found their way into High-Performance Computing (HPC), cloud computing, and other areas because they are fast to launch and can be used to allocate resources or as portable, interoperable applications that can be distributed as packaged applications.[30][54][55] The overhead of a container can be so small that the performance can be almost as fast as bare-metal systems.[33]

The hypervisor-based virtualization and OS-level virtualization solve different problems. While hypervisor-

based virtualization is often more used as the core at the infrastructure layer of a system because of its big overhead[61], container-based virtualization has more focus on constructing self-contained application packages that are fast and easy to deploy, run, distribute, replicate and have the ability to interconnect with other containers. [54][53] The following existing container-based virtualization tools can be used to turn applications into containers but are not limited to LXC[4], Singularity[5], and Docker[3]. 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]

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, before concluding whether the approach is practicable, it is essential to note that the varying simultaneously running number of SUL containers must be managed.

2.4 Orchestration Tools for Containers

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. Managing application containers on a clustered system is no easy task as the number of containers grows. Container orchestration tools provide a framework for managing containers at scale. Furthermore, container orchestration tools can help automate container deployment, management, scaling, and networking. In addition, it is achievable to run hundreds and more with the help of orchestration tools. The range of primary features for orchestration tools is as follows [36][49][55]:

- **Cluster state management**
- Ensuring **security**
- **Resource limit control**: Permitting and restricting the computing power and memory of the container
- **Scheduling**: Placing the desired quantity of containers on the available nodes while bearing in mind any constraints or requirements, e.g., the resource limit that the container underlies
- **Load balancing**: Balancing the load by distributing it among multiple containers
- **Simplifying networking**
- **Health check**: Checks whether a container is inoperative to decide whether it has to be terminated and a new one launched
- **Fault Tolerance**: Through the help of the health check - the replica controller that implements the fault tolerance can maintain the desired quantity of containers by replacing inoperative or faulty containers
- **Auto-scaling**: Dynamically adjusting the number of containers based on the demand to handle the ever-changing workload and enable high availability

- **Service Discovery**: In a service registry network, locations of service instances are stored to work with those services - this is needed because of the dynamic nature of a cluster
- Enabling **continuous delivery and deployment**
- **Providing monitoring and governance** for the infrastructure and container activity

Three popular container orchestration tools that can be utilized to manage containers are but are not limited to Kubernetes (K8S)[8], Docker Swarm[7], and Apache Mesos[6]. 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.[72] Due to being open-source, enabling customization it is actively developed by a large community.[20]

To conclude, utilizing Docker to wrap the SUL application into a lightweight container which in turn is orchestrated by K8S, enabling but not limited to auto-scaling, scheduling, and resource consumption control of said containers, is practicable to move active automata learning to an environment of a clustered system. Nevertheless, it is essential to remember that using these tools increases the overhead of the overall learning setup, which is evaluated in chapter 6.

3 Applying Automata Learning to a Clustered System

The underlying architecture on which the active automata learning runs is based on the container-based virtualization technology by utilizing the tool Docker to create lightweight self-contained SUL containers. In addition to an orchestration tool - K8S - to manage and deploy said containers while offering the feature to scale them up and down based on a metric while a learning process is running. Before bringing Docker and K8S together, further considerations have to be made.

3.1 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 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 gain fluctuates.

Furthermore, it is conditional 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 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 2.1, 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.[51]

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

3.1.1 The Need for a Broker

Due to dynamic scaling and the usage of self-contained containers for the SUL, which are orchestrated on a clustered system, direct communication between the learner and the SULs is not feasible due to the dynamic number of active SULs at any point in time. Thus, a system is required to handle the communication and ensure load balancing while the sent MQs are distributed to the available SULs, waiting for a query. In addition, the query distribution should be performed asynchronously to reduce additional overhead, e.g., wait time until a SUL receives a query. Therefore 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. The primary characteristics of a message broker are [40]:

- **Time decoupling:** Interaction between entities does not need to happen simultaneously.
- **Entity decoupling:** Entities that interact through a message broker are not required to be aware of each other.
- **Synchronization decoupling:** The execution process of entities producing or consuming messages is not blocked.

Moreover, a message broker offers a common infrastructure for systems to produce and consume messages. Usually, it is based around the publish/subscribe paradigm.[60] 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 transferring messages from the learner to the SULs. For this reason, the message broker RabbitMQ[22] is suitable because it fulfills these requirements.[48]

Enable Asynchronous Query Processing Using RabbitMQ

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)[1], enabling asynchronous processing of data. The RabbitMQ infrastructure is built around messages, queues, exchanges, and route bindings, as shown in Figure 3.1.[39]

At the core of RabbitMQ are messages containing any data. A message that is sent by an application goes through an exchange that acts similar to a router accepting messages and sending them on their way to a message queue based on route bindings. Thus bindings connect exchanges and queues. An application subscribed to a message queue can consume and act on these messages—the distribution of the messages in a queue to the subscribed applications is done with the Round-Robin algorithm. [45] Consequently,

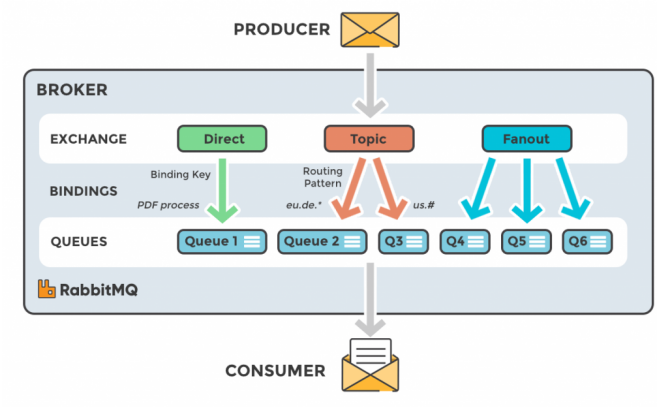

Figure 3.1: The RabbitMQ Flow[1]

enabling load balancing by default through the nature of the round robin algorithm to distribute the messages evenly among the consuming applications.

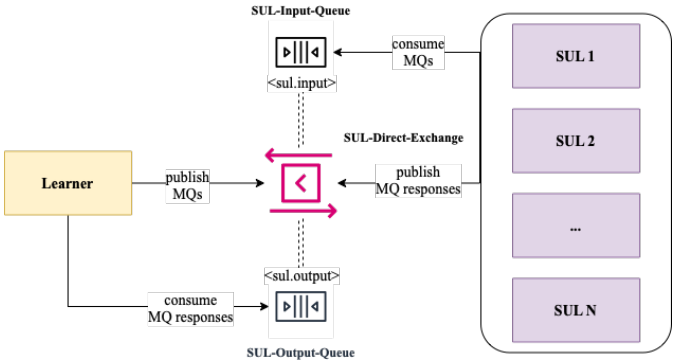

Figure 3.2: Learner and SUL communication with RabbitMQ

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, unprocessed MQs with the routing key 'sul.input' are distributed to the message queue 'SUL-Input-Queue,' from which SULs retrieve MQs for processing. On the other hand, processed MQs are sent to the message queue 'SUL-Output-Queue' with the routing key 'sul.output', from which the learner builds its hypothesis of SUL. As mentioned above, the built-in Round-Robin algorithm provides load balancing between SULs.

3.1.2 Necessary Adjustments to System Under Learning and Learner

Through the use of a message broker to handle communication, there is a need to adapt the implementation of the SUL and learner regarding the start-up of said applications and the queries sent as messages. First, both learner and SUL need to establish a connection to the RabbitMQ infrastructure by executing the following instructions on start-up:

- Creation of a connection factory instance
- Configuring the host
- Create a connection
- Creating a channel for the connection
- Creating a queue for the channel (if not present)

Both learner and SUL are consumers and producers of messages, and as a result of the asynchronous communication, the structure of a MQ has to be modified as well. The reason is that the learner that produces a message that is received, processed, and sent back by one of many SULs is not guaranteed to be received in the same order in which the learner sent them. Therefore, assigning the received output from any SUL to the matching MQ is difficult. As a consequence, the structure of the MQ is extended by a Unique Identifier (UID) which is used to match a received answer from any SUL to the appropriate MQ.

3.2 Containerization Of The System Under Learning

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. As a result, the underlying foundation of the SUL determines the impact on such as:

- **Processing time** of a MQ
- **Scale-ability** of the SUL, including the time it takes to create 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 are as follows:

- **LearnLib** is used to implement the SULs behavioral model represented as a Mealy machine
- The language that is used is **Java**[12]. As a result, the SUL runs by default on a **Java Virtual Machine (JVM)**[75]
- Through the use of the framework **Spring Boot**[23] or **Spring Native**[2] (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

3.2.1 From Application To Container

The tool Docker runs on the underlying Docker Engine (), which is essentially the server that provides the container virtualization technology and the interaction interface to manage Docker objects such as:

- **Images**: A Docker image is a step-by-step instruction on creating a Docker container out of an application. It is a shareable, read-only creation template to package the application in question into a container. Such an image can be created based on a Docker file, a simple file that defines the steps to create an image and how the application is run through a special syntax. Docker files can be based on another image already created, allowing for the extension of the said base image. For example, the SUL Docker file can use the image base of an OS, e.g., Ubuntu, and install the SUL application with all its dependencies and configurations needed to run the application. As a result, this Docker file can be used to create a new image enabling the creation of Docker containers based on this new image. Another benefit of these templates is that they are shareable through registries by uploading and downloading them. [35]
- **Containers**: An image can be used to create a container; after creation, it is runnable. Using the Docker Engine, such a container can be interacted with by but not limited to starting, stopping, deleting, and even creating new images of the current state of the container. However, by removing a container, any over-time changes to its state that are not saved in any persistent storage vanish.[35]

Before discussing concerns regarding the practicability and performance of the active automata learning running on a clustered system, there is to note that the following two SUL designs are used in the experiments (discussed in chapter 4) in this work:

- **Permanent SUL**: This type of SUL is already used in many experiments, as mentioned in section 1.1 because the chosen implementations offer reset functionality. This type constantly runs while it accepts and answers all MQs distributed to it. This SUL container is created once and stays until it is terminated through auto-scaling or a run-time error.
- **One-Use SUL**: This SUL accepts only one MQ and answers it. After answering the query, it terminates itself. As a result, for each MQ, a SUL container has to be created.

The one-use type of SUL exists to simulate complex real-life applications that do not offer any functionality to reset itself to its initial state to preserve the requirement that each MQ is independent of the other. Consequently, rebooting such applications is mandatory in the context of automata learning.

3.2.2 Factors That Influence Performance And Practicability

Besides the overhead of the orchestration tool (explained in section 3.3 and evaluated in chapter 6), the container itself may be a bottleneck regarding the total execution time of a learning process. The following factors, but not limited to, may have a notable impact on the total execution time: Docker image size, the start-up time of the application itself, and the overhead of Docker. To start, the added overhead of Docker

containers is of no concern. Without including the application, its dependencies, and the base image used, Docker adds a not noticeable overhead - approximately 0.036s stated by the author of [42].

Despite that, when using Docker image registries to create containers in the clustered system, the size of the Docker image is essential. Since the image has to be pulled before it can be used as a base to create containers and depending on the network speed of the clustered system in use, the time taken till the container is ready is impacted by the size of the Docker image. As shown in **??**, the size of an image can vary a lot.

Base Image	Compressed Size
openjdk:11	***334,91 MB***
openjdk:11-jdk-slim	***241,63 MB***
openjdk:11-jre-slim	***95,57 MB***
openjdk:11-jre-slim-buster	***90,16 MB***
adoptopenjdk /openjdk11:alpine-jre	***67,74 MB***
native-image graalvm	***26,02 MB***

Table 3.1: Image Size Variation

However, this impact is significantly reduced when using the K8S feature to use an image cache - each node has a separate image cache. As a result, the image has to be pulled only at the start of a learning process if it is not present on the node's image cache. In contrast, it is still a concern when using more than one node in a cluster due to only operating on a single node. This behavior is configurable through the pull image policy in the specification of containers in K8S and can take on the following values: *'Always'*, *'IfNotPresent'*, and *Never*. For this work, the value *'IfNotPresent'* is used, which only pulls the image at the beginning of a learning process and is re-used for each newly created SUL container. As a result, container creation is reduced, especially in a learning setup using one-use SULs.

Another concern regarding auto-scaling (explained in section 3.4) is that many containers are expected to be terminated over time and new ones are created. This is primarily a concern when using the one-use SUL type. On the one hand, the Docker image size and the orchestration overhead of K8S influence how fast K8S creates a container. Still, on the other hand, 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 3.2, the time it takes to start up the SUL application itself can fluctuate and vary a lot based on the implementation, how the executable is built and how the Docker image is created. As a result, this factor significantly impacts the total execution time of a learning process when paired with auto-scaling or one-use SULs. An in-depth evaluation of the significance of a lower start-up time can be found in chapter 6. As illustrated in Table 3.2, optimizing and reducing the overhead of the dependencies of the SUL can significantly increase the total execution time compared to an optimized approach using Spring Native and GraalVM[10] and considering a scenario of a learner sending in a total of 18046 queries (based on an experiment Linux TCP Server protocol model explained in chapter 4). In addition, on the condition that a learning process is using the one-use SUL and only considering the start-up times (approximated to the next higher value) mentioned above, a learning process spends approximately 16 hours (using not optimized SUL) compared to half an hour (using optimized SUL) on starting up SUL containers alone. That is to say, the total queries sent by the

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

Table 3.2: Importance Of The SUL's Internal Factors To Performance

learner in the example above are based on the Learner already knowing the behavioral model of the Linux TCP server protocol. As a result, the learner is not using approximated EQs to learn the behavioral model and knows an optimized way to learn, leading to a notably reduced sent query count. As an example of the magnitude, the coffee machine (explained in section 2.1) requires only 118 queries to be learned, not using approximated EQs compared to over 5000 while using a learning algorithm using approximated EQs. This is an increase of 4137% in total queries sent.

In conclusion, the process of how the SUL containerization is done can notably impact the total execution time of the learning process. Furthermore, the implementation and how the executable of the SUL is created must be considered, especially regarding real-life applications and auto-scaling usage. To conclude, the SUL image created through GraalVM[10] and Spring Native[2] is used in the experiments discussed in chapter 4 due to its small size and fast start-up time compared to the other base images.

3.3 Orchestrating Containerized SULs

For a start, K8S is an open-source container orchestration tool. It enables creating, deploying, managing, and scaling containerized applications across one or more physical-/virtual- machines - namely nodes. Furthermore, it offers numerous tools to orchestrate containerized applications; these tools are necessary to offer the following features such as:

- Auto-Scaling of containerized applications and their available resources based on metrics
- Managing the life-cycle by automating deployments and the ability to rollback to previous versions or pause and continue a deployment
- Ability to declare the desired state, which is maintained at all times and recovered from any failures
- Resilience and self-healing by restarting, replication, scaling, and placement of containers
- Persistent storage by mounting and adding storage dynamically
- Load balancing

K8S running on one or more nodes is called a K8S cluster. Each node provides the cluster with the necessary resources, e.g., storage and CPU, to run and manage containerized applications. A K8S cluster supports up to <cite and number> nodes through node scaling. Furthermore, K8S cluster nodes follow a master-/slave-architecture by differentiating nodes into at least one master node and one or multiple worker nodes. In contrast, a K8S cluster with only one node that fulfills the role of both master- and

worker nodes is also possible. The worker nodes have the task of running the containerized applications while listening to the commands of the master node.

A K8S cluster is based on the concept of the *desired state*, a programmable collection of configuration files applied to the K8S cluster. Such a configuration file can include, e.g., which images should be used to create containerized applications, the initial quantity of replicated containers and the resource range for each container is limited to. Provided that auto-scaling is enabled, scaling container replicas up and down based on metrics is another task, such as restarting crashed containers or rolling back updates on failure. Hence the master node(s) take(s) care of but is not limited to:

- **Controlling** and **managing** worker nodes
- **Scheduling** and **scaling** containerized applications
- **Maintaining** and achieving the configurable **desired state** of the K8S cluster

As a result, K8S provides the following components to master nodes to achieve the tasks of maintaining the desired state:

- **Kube-API server** offers operations ready to use to configure the objects of the K8S cluster
- **Kube-Controller-Manager** watches the K8S cluster state through the Kube-API server and makes changes to achieve, maintain or restore the desired state
- **Etcd** stores the K8S cluster configuration data, and metadata
- **Kube-Scheduler** is an ongoing process that assigns containers to nodes based on, e.g., its constraints and configured available resource range

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, or in the context of this work, a SUL. 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. K8S uses but is not limited to the Docker Engine as its container engine, enabling the use of Docker images and the interaction with Docker containers.

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 one-use SULs (established in section 2.3) 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, e.g., pulling the image from an image registry, applying configurations or secrets.
- **Running**: The container runs without problems.

- **Terminated**: The container ran to completion, failed, or is forced to terminate through external means, e.g., auto-scaling.

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. [13]

Besides the basic unit pod, K8S offers other configurable objects. In this work, the focus lies on the objects deployments and (batch) jobs:

- **Deployments** define long-running tasks that are supported by the 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. It is important to note that pods in a deployment only support the value 'Always' as the restart policy. The deployment maintains the desired number of pods specified. In addition, CPU and memory requests and limits can be set in a deployment.
- **(Batch) jobs** define tasks that run to completion. Unlike deployments, jobs offer the same configurations but are not supported by the auto-scale feature. As a result, scaling implicates a restart of the whole setup when used in a learning setup. The desired number of replicas are maintained, but in contrast to deployments, all restart policies are supported.

Both options are unsuitable due to this thesis's design choices. Firstly the learning process has to be scale-able on run-time and secondly allows using one-use SULs. As a result, (batch) jobs are not suitable because they are not scale-able on run-time and have to run to completion which K8S enforces by having to set a fixed number of completions for each (batch job). On the other side, deployments unsuitable because of the fixed always restart policy. As a result, it does not matter whether the SULs run to completion after processing a MQ or fail due to an error - the container always restarts. The restart loop or its exponential back-off delay is not configurable and can not be deactivated by design. Consequently, the one-use SULs can not be used efficiently with deployments because this mechanism leads to a significant downtime between MQ answers.

However, even though K8S does not offer a specific object to the unique circumstances of this work, it is still practicable to use deployments, even using one-use SULs. This is achievable through the following options:

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 on the master node of the K8S cluster
2. Instead of using objects like deployments and bash jobs to orchestrate the containers, implementing

a **custom pod management logic** deployed as a pod that handles creation, termination, and scaling through the use of the Kube-API

3. Using deployments but observe each container state in a deployment through another deployed application and **terminate the observed pod upon reaching a specific state and restart a new pod in its place**

For this work, the decision lies on the third option because such an application already exists. For this purpose, the application Kube Remediator[14] is used to enable the use of one-use 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 (with a set limit of completions). This work contains special requirements which do not fit the K8S design. Through the use of third-party tools, it is practicable to perform active automata learning on a K8S cluster even with the use of one-use SULs.

3.4 Active Automata Learning at Scale

The principle focus of this work is to perform active automata learning at scale on a clustered system. In this context, K8S offers, as mentioned in the previous section, the ability to manually or automatically scale up and down the replica count of, e.g., a deployment. Although manually scaling up and down based on the current load is practicable for a K8S cluster with a small and manageable scope, it is not efficient and manageable regarding the unpredictability of the workload and the resilience (high availability) a K8S cluster might need in a production environment.[76] In the context of active automata learning, another concern of using manual scaling or the absence of scaling can lead to the following:

- **Waste of computation resources** through the use of too many replicas resulting in idle pods with no performance gain
- **Longer execution time** due to an increased wait time till a MQ is processed when using an insufficient amount of SULs

Besides the benefits of meeting the current workload by scaling, auto-scaling can have the side effects of managing the available resources of the cluster more efficiently. Consequently, the cluster computation resources are only used as much as needed, making more resources available and reducing the cost of maintaining the cluster. Finally, it leads to more manageable clusters with a enormous scope.[62]

In a scenario in which auto-scaling is required, K8S offers two objects 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. [31] 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, using the HPA is a more suitable choice.

3.4.1 Horizontal Pod Autoscaler

The HPA scales a target K8S object, e.g., deployment, by adjusting the desired replica count in the target object; 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. [52] HPA scales based on the average utilization, across all replicas, of the chosen metric and triggers a replica count update through the following equation[9]:

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

Furthermore, using standard HPA, it is impossible to scale the replica count to zero, therefore 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.

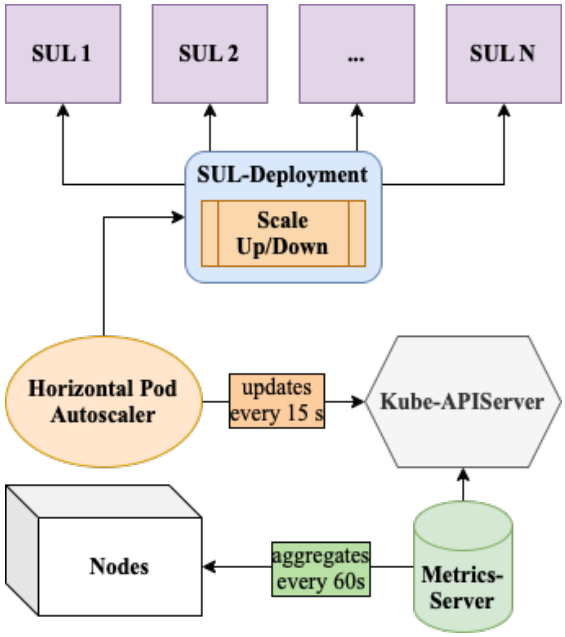

Figure 3.3: Simplified overview of K8S auto-scaling process.

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. This 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. [63]

3.4.2 Kubernetes Event-driven Autoscaling

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.

As a result, a third-party solution, namely Kubernetes Event-driven Autoscaling (KEDA)[15], is utilized to enable auto-scaling for the chosen event-driven architecture: RabbitMQ. KEDA is an extensible, lightweight component and can be attached to a K8S cluster without disturbing or replacing K8S own auto-scaling objects 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 works with the resources such as:

- **ScaledObject**: Defines how a K8S target object should be auto-scaled and on which triggers - if multiple triggers are defined, auto-scaling happens based on the trigger allowing for the highest amount of change in the replica count
- **Keda-Operator**: Is responsible for activate and deactivate the K8S target object by scaling the replica count from zero and to zero respectively
- **Keda-Operator-Metrics-APIServer**: Acts like an extension to the standard K8S metrics server by exposing event data to the HPA for scaling. HPA uses the exposed event data for its scaling task, as illustrated in Figure 3.3. Furthermore, using the KEDA's metrics server allows to avoid the polling interval of the standard K8S metric server (default: 60s). It enables a simplified control over the polling interval (configurable in the ScaledObject through the field 'pollingInterval' shown in ??). In contrast modifying the polling interval of the metrics server affects the whole K8S cluster instead of a specific, e.g., deployment or HPA. It is to note that the polling interval only affects the metrics update and thus does not trigger a new replica demand calculation. Demand calculation and scaling of the Deployment is still the task of the HPA.

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 object based on the exposed metrics of each event-driven infrastructures. 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: Auto-Scaling Configuration Part Of a KEDA ScaledObject (YAML)

```yaml
spec:
    scaleTargetRef:
        apiVersion: apps/v1
        kind: Deployment # K8S object type of the target
        name: sul # The name reference to the target K8S object
    pollingInterval: 5 # Default: 30 seconds - interval to check on metrics
    cooldownPeriod: 6000 # Default: 300 seconds - interval to wait after last
        trigger update till scaling back to 0 - only applies when scaling to
        replica count 0
    idleReplicaCount: 0 # Default: ignored -
    minReplicaCount: 1 # Default: 0 - Min. replica count to be maintained
    maxReplicaCount: 100 # Default: 100 - Max. replica count not to be exceeded
    triggers: # List of auto-scale triggers -
    - type: rabbitmq # Type of auto-scaler
        metadata:
            protocol: amqp # Protocol used to scrape the metrics
            mode: QueueLength # Possible metrics: QueueLength or MessageRate
            value: "1" # Message count(QueueLength) or Publish/sec.
                rate(MessageRate) to trigger on.
            queueName: sul_input_q # Target RabbitMQ queue
```

An example KEDA configuration file, which illustrates how a KEDA ScaledObject and its available fields can be configured, is shown in figure 3.4 (more fields and details in [15]). Finally, a ScaledObject offers the modification of the HPA behavior through the use of advanced fields in the configuration file. KEDA simplifies the configuration through the ScaledObject configuration file and allows for modification of the whole behavior of the HPA regarding the target K8S object. By default, scaling down and up can occurs 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 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.[11] At last, limiting the number of pods terminated or created through scaling is also configurable.

Auto-scaling may not work for every setup, especially regarding scaling down replicas. Scaling down implicates the termination of pod replicas, which may lead to problems. In the context of this work, a problem emerges when a SUL terminates while still processing a received MQ. In the worst case, the SUL is killed by K8S before the process finishes, and the MQ might be lost, blocking the whole learning process because the learner is still waiting for a response to this specific MQ.

Nevertheless, based on the implementation of the SUL, regarding the interaction with RabbitMQ messages, messages can be re-queued before termination or are not removed from the queue until processed. 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 SUL 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 SUL 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. In most cases, this is sufficient. However, the grace period has to be adjusted while using applications that run heavy processing tasks that exceed this boundary. This scenario occurs while running the experiments mentioned in chapter 4; therefore, adjusting this value is required.

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 a more efficient use of computational resources than the fixed number of SUL replicas. In contrast to that the auto scaling strategy might be slower regarding the total execution time. Furthermore, using auto-scaling implicates additional management effort regarding the container life cycle depending on the learn setup.

3.5 Constructed Architecture

Setting up and configuring the K8S architecture used in the cluster can be an intimidating task with a steep learning curve. However, using MicroK8s[16] simplifies this process. MicroK8s enables the creation of a minimal K8S cluster by providing the core functionality of K8S and the ability to scale it to a highly available production cluster later. Therefore, the MicroK8s tool was used to set up the architecture.

As explained in the previous chapters, in addition to Kubernetes, the cluster setup consists of Docker as the container engine (for managing the containers), RabbitMQ as the message broker and load balancer, KEDA for automatic scaling based on RabbitMQ metrics, and the more delicate configuration of up/down scaling concerning the HPA.

Starting a learning process in this environment must be done with more care than starting the whole process on a local machine. The following steps must be performed to start a learning process on this architecture:

1. Configure and create a **SUL deployment** including:

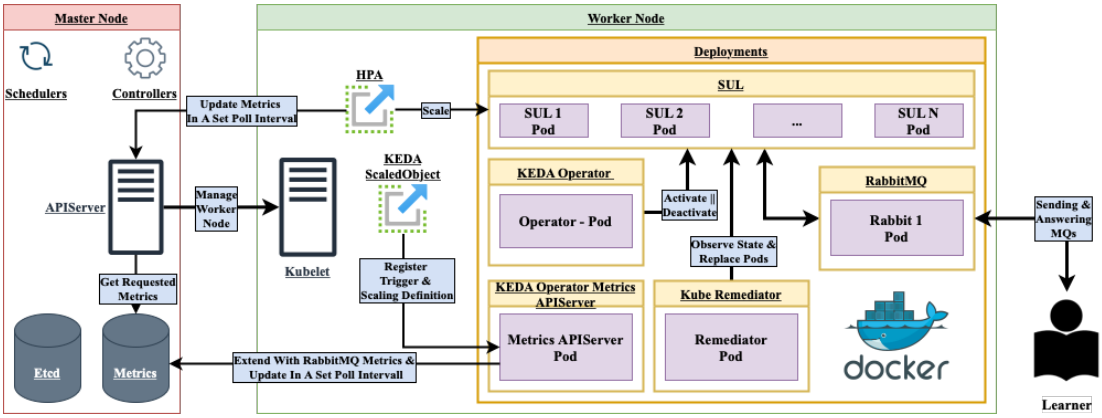

Figure 3.5: Simplified Overview Of the Architecture

- The **SUL image** that should be used. Determines the chosen SUL type, e.g., one-use, and the behavior of the SUL, e.g., coffee machine
- **Minimum and maximum SUL replicas**, which may be overwritten by KEDA when using auto-scaling
- **Configuration values** regarding , e.g., RabbitMQ connection details or other SUL related configurations
- Deployment-related configuration, which may be unique to the SUL requirements, e.g., grace period, pre-stop hook or pull image policy

2. If the deployment is **to scale automatically**, a **KEDA ScaledObject** must be configured and applied to the K8S cluster - including:
   - **Target deployment** mentioned above
   - **SUL replica** 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**
     - Threshold **value**
     - **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 as well
3. The last step is to **launch the learner** and start the learning process

At last, all components that make up the architecture and are usable to perform scale-able active automata learning processes were introduced. Furthermore, Figure 3.5 illustrates a simplified overview of the primary modern technologies used and how they fit together. Moving the active automata learning to a clustered system or, in this case, a K8S cluster adds management overhead, especially regarding the newly introduced one-use SUL type. Overall, how the architecture is presented makes scale-able active automata learning practicable. However, it is still open whether the overhead introduced by the technologies counteract the possible improvement in performance and lead to limitations down the road.

4 Evaluation Method

-> how is the data acquired -> tools, etc... -> impact on performance? -> how is the process of running an experiment / strategy -> diminish cold start by waiting for all initial replicas being created -> except when using autoscaling with mino -> sim. real-life applications good example: A performance comparison of cloud-based container orchestration tools

Before explaining the interaction of the previously explained components/tools that make up the architecture (illustrated in Figure 3.5) that enables active automata learning on the cluster, it is essential to mention the underlying hardware of the cluster. The term ”cluster” could be misleading in this case, as the experiments are performed on a single-node cluster, not on at least two or more nodes, as the term ”cluster” suggests. As mentioned in section 3.3, it is possible to set up a K8S cluster where a single node takes the role of both master and worker nodes. For clarity, the simplified overview of the architecture in Figure 3.5 shows the single cluster node as a separate master and worker node. The cluster is equipped with 126 GB of RAM and an AMD EPYC™ 7443P CPU with 24 cores @ 2.85 GHZ with 48 threads (more details on ()).

4.1 Data Acquisition

For each execution of a strategy for an experiment the following data will be acquired while running them:

- The strategy that is used
- Processing time (min, max, avg) of each SUL
- Start up time (min, max, avg) of each SUL
- The interval between each response the learner gets (min, max, avg)
- The time it takes to process a batch of queries (min, max, avg)
- The size of batches sent (min, max, avg)
- Number of sent queries
- Number of batches sent
- The total execution time it took to learn the model

4.2 Chosen Experiments

The EQ used in the experiments is a simulator EQ meaning that this EQ knows the correct behaviour model of the SUL enabling more goal oriented queries and reducing the total queries sent and therefore the execution time. This decision was made for time purposes.

4.2.1 Simple Coffee Machine

As the first experiment the coffee machine was chosen because it is small in size, not complex and its execution time to learn the behaviour model is fast. A perfect example to get insights about scaling a SUL with fast processing time.

Number of States	**5**
Number of Inputs	**4**
Number of Outputs	**3**
Number of Queries Sent	**118**
Minimum Number Queries in a Batch	**1**
Maximum Number Queries in a Batch	**4**
Average Number Queries in a Batch	**1**

Table 4.1: Simply Coffee Mealy Machine Statistics

4.2.2 Linux TCP Server Protocol

Number of States	**56**
Number of Inputs	**12**
Number of Outputs	**9**
Number of Queries Sent	**18046**
Minimum Number Queries in a Batch	**1**
Maximum Number Queries in a Batch	**17**
Average Number Queries in a Batch	**2**

Table 4.2: Linux TCP Server Protocol

4.2.3 Getting Closer to Real-Life Systems

Slowing Down the SULs

The experiments are run a second time with a artificial delay to the processing time of the SUL. This delay is calculated based on the query length that the individual SUL is processing. For each symbol in the query that needs processing a random time interval between 3000 and 5000 milliseconds is chosen and summed up over all symbols to calculate the total wait time before continue to process the query. Delaying the process of the query simulates systems that need a lot of time to process requests.

Reboot - SULs without Reset Functionality

4.3 Learning Setups

4.3.1 Performance Difference: JVM and GraalVM

4.3.2 Establishing the Baseline

To have a reference to compare to a evaluation baseline is needed. In this thesis a setup of one SUL replica at a time is the baseline of the evaluation.

4.3.3 Setup Types and Variations

Fixed

The first setup the experiments are run on, is a setup with a fixed number of SULs at a time including the one SUL baseline. This setup is executed with: 1 (baseline), 2, 4, 6, 8, 10, 12 and 16 SUL replicas. // WHY?

Autoscaling Based Queue Length

The scaling strategy that gets evaluated is based on the queue length of the SUL input queue. For every query that is in the input queue, at the time the KEDA scaler calculates the new demand, new SUL replicas are created or old ones terminated to fit the new set replica count.

delay values one use and no one use

// HOW DOES THE SCALING WORK IN KEDA

This strategy is executed variants as well. The minimum replica count is modified through each execution. The following minimum replica count values are used: 0, 1, 2, 3 and 4.

5 Results

5.1 Strategies

5.1.1 Strategy 1...

5.1.2 Strategy 2...

5.1.3 Strategy 3...

5.1.4 Best Performing Strategy

5.1.5 Slowing Down the SULs

5.1.6 Rebooting the SULs

5.1.7 Combination - Slow and Rebooting SUL

6 Evaluation

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

7 Future Research

7.1 Using Real-Life Systems Through Mapping

7.2 Reducing Orchestration Overhead

7.3 Using Better Suited Orchestration Tools

7.4 Custom Autoscaler

7.5 Evaluation of Alternative Metrics to Improve Autoscaling

7.6 Asynchronous Sending of Membership Queries and Equivalence Queries

7.7 Using Custom Control Plane

8 Conclusion

Bibliography

[1] [n.d.]. CloudAMQP Official Website. https://www.cloudamqp.com/, note = Accessed: 2022-03-31.

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

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

[4] [n.d.]. Container-based operating systems virtualization - LXC. https://linuxcontainers.org, note = Accessed: 2022-09-14.

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

[6] [n.d.]. Container Orchestration Tool - Apache Mesos. https://mesos.apache.org, note = Accessed: 2022-09-14.

[7] [n.d.]. Container Orchestration Tool - Docker Swarm. https://docs.docker.com/engine/swarm/, note = Accessed: 2022-09-14.

[8] [n.d.]. Container Orchestration Tool - Kubernetes. https://kubernetes.io, note = Accessed: 2022-09-14.

[9] [n.d.]. Container Orchestration Tool - Kubernetes - HPA. https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/, note = Accessed: 2022-09-29.

[10] [n.d.]. GraalVM - high-performance JDK. https://www.graalvm.org, note = Accessed: 2022-09-14.

[11] [n.d.]. Horizontal Pod Autoscaling. https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/#configurable-scaling-behavior. Accessed: 2022-10-04.

[12] [n.d.]. Java Official Website. https://www.java.com/de/. Accessed: 2022-10-04.

[13] [n.d.]. K8S Pod Lifecylce. https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/. Accessed: 2022-10-04.

[14] [n.d.]. Kube Remediator. https://github.com/ankilosaurus/kube_remediator. Accessed: 2022-10-04.

[15] [n.d.]. Kubernetes Event-driven Autoscaling. https://keda.sh, note = Accessed: 2022-09-14.

[16] [n.d.]. Kubernetes Setup Tool - Microk8s. https://microk8s.io, note = Accessed: 2022-09-14.

[17] [n.d.]. LearnLib Official Website. https://learnlib.de/. Accessed: 2022-03-31.

[18] [n.d.]. LearnLib Official Website - Performance. https://learnlib.de/performance/. Accessed: 2022-10-04.

[19] [n.d.]. libalf Official Website. http://libalf.informatik.rwth-aachen.de. Accessed: 2022-10-04.

[20] [n.d.]. Most Loved and Dreaded Technologies. https://survey.stackoverflow.co/2022/#most-loved-dreaded-and-wanted-tools-tech-love-dread. Accessed: 2022-10-04.

[21] [n.d.]. Project AutomataLib. https://learnlib.de/projects/automatalib/. Accessed: 2022-09-14.

[22] [n.d.]. RabbitMQ Official Website. https://www.rabbitmq.com/. Accessed: 2021-11-21.

[23] [n.d.]. Spring Boot Official Website. https://spring.io/projects/spring-boot. Accessed: 2022-10-04.

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

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

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

[27] Dana Angluin. 1987. Learning regular sets from queries and counterexamples. *Information and computation* 75, 2 (1987), 87–106.

[28] Carlos Arango, Rémy Dernat, and John Sanabria. 2017. Performance evaluation of container-based virtualization for high performance computing environments. *arXiv preprint arXiv:1709.10140* (2017).

[29] S Anish Babu, MJ Hareesh, John Paul Martin, Sijo Cherian, and Yedhu Sastri. 2014. 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*. IEEE, 247–250.

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

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

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

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

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

[35] Carl Boettiger. 2015. An introduction to Docker for reproducible research. *ACM SIGOPS Operating Systems Review* 49, 1 (2015), 71–79.

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

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

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

[39] Philippe Dobbelaere and Kyumars Sheykh Esmaili. 2017. 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*. 227–238.

[40] Patrick Th Eugster, Pascal A Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. 2003. The many faces of publish/subscribe. *ACM computing surveys (CSUR)* 35, 2 (2003), 114–131.

[41] Marco Henrix. 2015. *Performance improvement in automata learning*. Ph.D. Dissertation. Master thesis, Radboud University, Nijmegen.

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

[43] Falk Howar, Malte Isberner, Maik Merten, and Bernhard Steffen. 2012. LearnLib tutorial: from finite automata to register interface programs. In *International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*. Springer, 587–590.

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

[45] Valeriu Manuel Ionescu. 2015. The analysis of the performance of RabbitMQ and ActiveMQ. In *2015 14th RoEduNet International Conference-Networking in Education and Research (RoEduNet NER)*. IEEE, 132–137.

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

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

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

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

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

[51] Maik Merten. 2013. Active automata learning for real life applications. (2013).

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

[53] Claus Pahl. 2015. Containerization and the paas cloud. *IEEE Cloud Computing* 2, 3 (2015), 24–31.

[54] Claus Pahl, Antonio Brogi, Jacopo Soldani, and Pooyan Jamshidi. 2017. Cloud container technologies: a state-of-the-art review. *IEEE Transactions on Cloud Computing* 7, 3 (2017), 677–692.

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

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

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

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

[59] Robert Rose. 2004. Survey of system virtualization techniques. (2004).

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

[61] Cristian Ruiz, Emmanuel Jeanvoine, and Lucas Nussbaum. 2015. Performance evaluation of containers for HPC. In *European Conference on Parallel Processing*. Springer, 813–824.

[62] Gigi Sayfan. 2017. *Mastering kubernetes*. Packt Publishing Ltd.

[63] Sasidhar Sekar. 2020. 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. Accessed: 2022-09-29.

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

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

[66] Stephen Soltesz, Herbert Pötzl, Marc E Fiuczynski, Andy Bavier, and Larry Peterson. 2007. 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*. 275–287.

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

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

[69] Malte Isberner1 Falk Howar2 Bernhard Steffen. [n.d.]. LearnLib: An Open-Source Framework for Active Automata Learning. ([n. d.]).

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

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

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

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

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

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

[76] Deepak Vohra. 2017. Using autoscaling. In *Kubernetes Management Design Patterns*. Springer, 299–308.

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

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