GRAN SASSO SCIENCE INSTITUTE
SCHOOL OF ADVANCED STUDIES
Scuola Universitaria Superiore

Doctoral Thesis

Data-Driven Self-Adaptive Architecting Using Machine Learning

PhD Program in Computer Science: XXXIII cycle

Supervisor:

Prof. Henry Muccini
henry.muccini@univaq.it

Author:

Karthik Vaidhyanathan
karthik.vaidhyanathan@gssi.it

Internal advisor:

Dr. Ludovico Iovino
ludovico.iovino@gssi.it

January 2021

GSSI Gran Sasso Science Institute
Viale Francesco Crispi, 7 - 67100 L'Aquila - Italy

Declaration of Authorship

I, Karthik Vaidhyanathan, declare that this thesis titled, 'Data-Driven Self-Adaptive Architecting Using Machine Learning' and the work presented in it are my own. I confirm that:

- This thesis has been composed by myself under the guidance of my supervisor Prof. Henry Muccini for a research degree at this university.
- The overall conceptual approach presented in this thesis is based on my work [1] co-authored with Henry Muccini. Chapters 4 and 5 are based on my works [2] and [3, 4] respectively, co-authored with Henry Muccini.
- Chapter 6 is a re-elaboration of my work [5] co-authored with Javier Cámara and Henry Muccini. The first part of Chapter 7 is based on the work [6] co-authored with Mauro Caporuscio, Marco De Toma, and Henry Muccini. The second part is based on the work [7] co-authored with Martina De Sanctis and Henry Muccini.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.

"Dream is not what you see in sleep; is the thing which doesn't let you sleep"

Dr APJ Abdul Kalam

Abstract

The last decade has seen a significant evolution in software architecting practices as the process of managing and developing software is becoming more and more complex. This is especially true due to the heterogeneous composition of modern software systems coupled with the run-time uncertainties. These include application downtime due to high CPU utilization, server outages, resource constraints, dynamic resource demands, etc. These can have a big impact on the Quality of Service (QoS) offered by the system, thereby impacting the experience of the end-user.

Self-adaptation is nowadays considered to be one of the best solutions to dynamically reconfigure a system in the occurrence of deviations from the expected QoS. However, one of the issues with the existing solutions is that most of them are *reactive* in nature, where adaptation is carried out in the event of uncertainties. Moreover, current adaptation methods: i) do not provide the systems the ability to proactively identify the need for adaptation with good accuracy; ii) may temporally overcome an impending failure, while not preventing the system from the state in the future. In essence, they do not exploit the knowledge gained from every adaptation performed. The use of machine learning techniques to aid self-adaptation has been proposed in the literature as a way to mitigate this problem based on the concept of *self-adaptation through achieving*, but not much work has been done in this area. Moreover, the challenges associated with learning bias and less accurate predictions also need to be handled while using machine learning techniques, which otherwise leads to sub-optimal adaptations.

To this end, in this thesis, we develop a data-driven approach to architecting self-adaptive systems using machine learning techniques. The approach, in principle, shifts the focus from *self-adaptive architectures to self-learning architectures*. It achieves this by using a combination of deep neural networks and reinforcement learning (RL) techniques to ensure that the system continuously learns and improves the quality of adaptation performed over time. It further uses quantitative verification (QV) techniques to overcome learning bias and enable faster convergence towards optimal adaptations. More specifically, the approach i) continuously monitors the system data; ii) uses deep neural networks to forecast any QoS uncertainties; iii) leverages the forecasts using RL techniques to find the adaptation strategy; iv) it further uses QV techniques to verify the decision selected by RL; v) keeps improving the decisions based on the forecasts as well as the feedbacks obtained from QV; vi) continuously keeps performing the loop of learning, selection, verification, and adaptations to converge towards optimal adaptations, thereby enabling the architectures to learn and improve over time.

Acknowledgements

I owe my heartfelt gratitude to my supervisor Prof. Henry Muccini, for his guidance with immense knowledge of research, constant support, and motivation. It enabled me to push forward and expand my boundaries. The professionalism, hard work, passion, and commitment that he displayed has been of great inspiration. This thesis would not have been possible without his guidance, and it has been an honor to work under his supervision. I also want to thank my internal advisor, Dr. Ludovico Iovino, for all his valuable inputs and suggestions.

I want to thank the Gran Sasso Science Institute for giving me this opportunity to pursue doctoral studies and providing me with all the necessary support throughout the research program. Most importantly, I would like to thank the entire CS group at GSSI, especially Prof. Luca Aceto and Prof. Michele Flammini, for their constant support and encouragement.

Many thanks to Javier Cámara, Mauro Caporuscio, and Martina De Sanctis for their tremendous support leading to fruitful collaborations. It was a great experience working with each of them, and I learned a lot more about the research field. I want to thank the Google Research Credits program for providing me with a cloud research grant in the form of credits to run the different experiments for my research. I would also like to thank two master's degree students, Matteo and Marco, from the University of L'Aquila, for their excellent work, whom I was able to co-supervise during my Ph.D.

On a more personal note, during my Ph.D. studies, I met some exceptional individuals, both within and outside the institute. I want to thank each one of them for their incredible support. I would also like to thank all my research group members at the University of L'Aquila for their encouragement. Further, I express my gratitude to all the teachers and friends from my school and university days who have been instrumental in shaping my career. Most importantly, I want to thank Mrs. Anitha, my first computer science teacher. It was her motivation and encouragement that gave me the confidence to pursue a career in computer science. I would also like to thank Mr. Anish, who has always been there as a mentor to support and guide me.

All these would not have been possible without the support of my parents, sister, and wife. There were many challenging situations where I was distraught and disappointed. I am deeply grateful to my wife, who had to make different sacrifices during these three years. It was her support, patience, and encouragement that kept me going despite all the difficulties. Words fail to express the amount of gratitude that I have for my parents for all their sacrifices to make me who I am. It has been their blessing, support, and belief, along with the grace of God, that has fueled me to complete this thesis.

Contents

List of Figures

List of Tables

List of Abbreviations

AI	Artificial Intelligence
ANN	Artificial Neural Network
ARIMA	Autoregressive Integrated Moving Average
CPS	Cyber Physical Systems
CTMC	Continuous-time Markov Chains
DTMC	Discrete-time Markov Chains
EDA	Event-Driven Architectures
IoT	Internet of Things
LSTM	Long Short Term Memory
MAPE-K	Monitor Analyze Plan Execute - Knowledge
MASE	Mean Absolute Scaled Error
ML	Machine Learning
MLE	Machine Learning Engine
MLP	Multi-Layer Perceptron
MSA	Microservice Architectures
NRMSE	Normalized Root Mean Square Error
PRISM	Probabilistic Symbolic Model Checker
QoS	Quality of Service
QV	Quantitative Verification
RL	Reinforcement Learning

RMSE Root Mean Square Error

RNN Recurrent Neural Network

SA Software Architecture

SES Simple Exponential Smoothing

sMAPE Symmetric Mean Absolute Percentage Error

I dedicate this thesis to my parents, Vaidhyanathan and Sudha, my sister Kavitha, my wife Sreelakshmi, all my teachers and the almighty

Chapter 1

Introduction

"In fact what I would like to see is thousands of computer scientists let loose to do whatever they want. That's what really advances the field."

- Donald Knuth

Softwares have impacted billions of life across the world. Starting from its inception in the 1930s, usage of software kept expanding to different walks of life. The expansion also gave rise to the increasing complexity of software as the software was not just a simple computer program using a data structure and algorithm. Instead, it became a collection of programs that achieved different functionalities, and structuring the programs, following a development practice, and managing the overall complexity became challenging. This led to the birth of software architecture [8, 9]. Over the years, software complexity has further increased due to the ever-increasing pervasive nature of modern software systems, resulting in different architecting challenges to ensure better performance, reliability, security, etc [10, 11].

Furthermore, modern software systems generate a tremendous amount of data. In fact, we live in a data-driven world powered by software where we have an abundance of data generated by different sources like web applications, smartphones, sensors, etc [12, 13]. A recent article from Forbes quotes that about 2.5 quintillion bytes of data are created every day. This number is expected to increase drastically in the years to come [14]. Over the years, with the advancements in computing infrastructure, these data have been fueled by Artificial Intelligence (AI), in particular, Machine Learning (ML) to generate actionable insights [15, 16]. It has further paved the way for developing software systems and services that power autonomous vehicles, recommendations in Netflix, search results in Google, etc. As remarked by Andrew Ng, AI is considered the new electricity and is expected to transform the world just like electricity did about 100 years ago [17].

Moreover, as per Gartner, around 40% of the world's organizations are expected to leverage AI in the coming years [18]. However, the increasing adoption of AI has given rise to different challenges associated with development practices, deployments, ensuring data quality, etc. in addition to the challenges of a traditional software system. These challenges call for better architecting practices for addressing the concerns of AI-based software systems [19, 20].

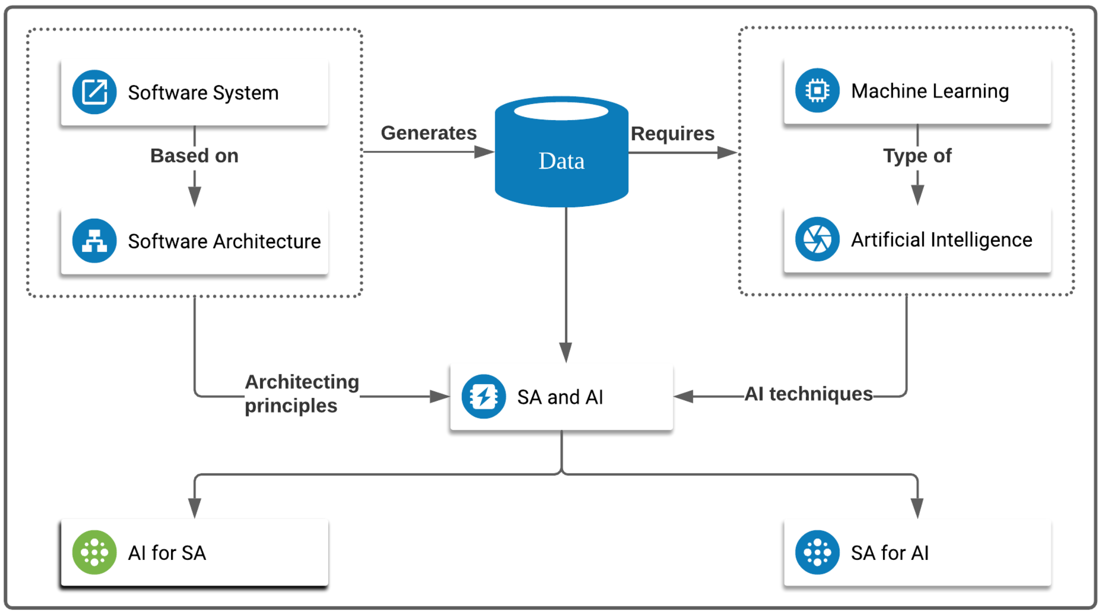


FIGURE 1.1: Research Premise

As represented in Figure 1.1, on the one hand, we have software systems that generate a tremendous amount of data but face different architectural challenges. Some of those challenges can be solved using AI [21], and on the other hand, we have AI systems that thrive on data but require better architecting practices. This combination of challenges in the field of Software architecture and AI has resulted in two broad research areas: i) Software architecture for AI systems. It primarily focuses on developing architectural techniques for better developing AI systems; ii) AI for Software architectures, which focuses on developing AI techniques to better architect software systems. In this thesis, we focus on the later side of the spectrum, intending to develop software systems that can leverage AI's power to autonomously improve their architecture.

In today's world, the purpose of developing software ranges from solving simple tasks like scheduling our daily activities to complex tasks like realizing self-driving cars. However, the more complex challenges they solve, the more challenging it is to architect and maintain these systems. This is especially true due to the heterogeneous composition of modern software systems. Moreover, these systems are subjected to various run-time uncertainties such as application downtime due to high CPU utilization, server outages, etc. These uncertainties can have a big impact on the *Quality of Service (QoS) offered by* *the system*, thereby impacting the end user's experience. This holds especially true for data- and event-driven systems, such as the Internet of Things (IoT) applications [22–24]. In fact, in the last decade, IoT has gained a lot of popularity such that it emerged as one of the most impactful research topics in academia and the industry [11]. Hence, in this thesis, we scope our research area within the context of IoT systems and see how the results can be generalized to other modern software systems.

1.1 Architecting IoT Systems: The Challenges

The emergence of IoT has revolutionized software development as in software development is not just about software, but it is about a system that contains software, a set of connected components such as sensors, actuators, anything, and everything [23]. It is more of a network of a lot of things connected via the Internet, where the definition of thing can range from the smallest sensor to a human being. IoT systems find applications everywhere, starting from a fitness monitoring app to environmental and indoor monitoring applications. The impact of these systems is so high that they have enabled thousands of people to keep a check on their fitness condition, monitor the quality of pollutants in the environment, facilitate security and automation in indoor environments, etc. In fact, IoT systems play a vital role in any domain they are applied to [25].

On the one hand, IoT systems are becoming very popular, and they are widely adopted, whereas on the other hand, architecting and maintaining IoT systems poses a lot of challenges [22, 24]. These include:

Heterogenity: IoT applications often consist of different types of devices (sensors/actuators) such as beacons, temperature sensors, cameras, etc. Each device might provide native support for specific microcontrollers, and this would mean programming the functionality of different devices in different languages (usually using low-level programming languages). Further, IoT is not just about sensors. It also consists of backend services deployed in the cloud, which are usually developed using high-level languages. To summarize, the heterogeneity in IoT applications arises from the devices and the backend services.

Interoperability: Each device in an IoT system might communicate using different communication protocols and standards. For example, a temperature sensor might communicate using LoRa, an Infrared sensor that may communicate via ZigBee, etc. Moreover, each device might follow their data standards. For instance, the camera might send data as bytes or image frames; an RFID reader might send data as text, etc. All these lead to problems of interoperability, and this needs to be managed. One of the primary reasons for interoperability is that many devices are proprietary and designed to operate in predefined or dedicated hardware.

Security: As in the case of any software systems, ensuring security is of utmost importance. However, this challenge is even harder in IoT systems due to the heterogeneity and interoperability issues stated above. The communication between devices and the cloud/fog needs to happen via a secured channel. This might otherwise allow an attacker to extract sensitive information. Moreover, the devices themselves need to be secured. For example, if not secured, a camera in an IoT ecosystem can allow the attacker to obtain real-time sensitive feeds that may violate user privacy.

There has been a significant amount of research in the domain of IoT to solve the above-listed challenges. With the ever-increasing importance of the Fog computing paradigm along with the emergence of architectural styles like Event-driven Architectures (EDA), Microservice-based Architectures (MSA), etc. some of the challenges related to interoperability and heterogeneity are handled [26]. Moreover, with the increasing popularity of blockchain technologies in IoT, some of the security challenges are taken care [27]. But, one crucial aspect of IoT systems, as stated above, is that it is not just about the software, the devices might be deployed in open and dynamic environments, and this gives rise to different types of uncertainties. For example, devices might run out of battery due to resource constraints, weather conditions might affect the functioning of sensors in external environments, etc. These uncertainties, coupled with the other challenges listed above, can significantly impact the *QoS levels offered by these systems* [22, 23]. Therefore, any architecture issues imply that uncertainty in run-time might render the entire system unusable or may have adverse effects on the humans using it [24]. This mandates the fact that the architecture designed should be robust to handle uncertainties in run-time. Better architecture for IoT systems also means that the problems of heterogeneity and interoperability are tackled [28].

In this direction, *self-adaptation techniques* have been proposed as a solution [22]. This is mainly due to their ability to handle the uncertainty as well as the dynamic nature of the environment [29]. In simple terms, self-adaptation techniques equip the system with the ability to adjust its behavior/structure in response to uncertainties or changes in environments. The prefix ”self” in self-adaptation implies that the system will have the capabilities to autonomously (with minimal human intervention) decide the action to be performed in the event of any uncertainty. For example, an IoT system equipped with self-adaptation capabilities shall automatically reduce device communication (to save battery), switch networks (to reduce congestion), etc. Due to this reason, the research in the field of adaptive systems has gained enormous popularity in the last decade.

This trend is expected to continue both in research and industry [11, 30]. However, architecting self-adaptive systems, in particular, self-adaptive IoT systems, poses a lot of challenges. These are listed in the following section.

1.2 Challenges in Self-adaptation

Although a big body of research has been done in the field of self-adaptation [31, 32], a lot of challenges remain open [29, 33]. To better explain the challenges, let us consider a simple self-adaptive IoT system for measuring humidity and temperature in a given room. Let us assume that it operates using battery-powered sensors. For the sensors to last longer, the system should adapt automatically to ensure that the sensors do not consume more than 2 joules every minute. The following challenges exist:

1. Reactive nature of adaptation : Most of the self-adaptive solutions that exist in literature are reactive in nature [31], where the adaptation is performed in the event of uncertainty. In the example given above, this would mean that the system *can adapt only when the energy consumption goes above 2 joules.* In such scenarios, the system might save more with the use of proactive techniques, where the need for adaptation can be foreseen and executed before any uncertainty. There have been some works in the field of proactive self-adaptation [34, 35]. However, one of the biggest challenges in applying proactive techniques, especially in IoT systems (where the rate of change of QoS is quite high), is to generate predictions with good accuracy [31]. Towards this, AI techniques, in particular machine learning techniques, can be used to leverage the tremendous amount of data generated by the IoT systems to predict any possible QoS issues. This implies that the self-adaptive systems should have the ability to learn from the data to proactively foresee any uncertainty to further improve the effectiveness of the adaptation process. This has not been explored much in the literature [31].

2. Systems adapt, but they do not learn: adaptation is more an instantaneous process performed by a system to overcome a given situation. Learning, on the other hand, is more than just an adaptation. It is a process that enables the system to improve its performance over time [36]. In the event of uncertainty, traditional self-adaptive systems use techniques like rule-based algorithms, control theory techniques, model checking, model-driven techniques, etc. [32] to decide on the best adaptation strategy. Off late, self-adaptive systems are also making use of machine learning techniques towards decision making [37, 38]. However, one main issue is that the performed adaptation does not necessarily improve the architecture; it may temporally overcome an impending failure while not preventing the system from entering the same state. Instead, the architecture adapts to the current context, but it does not learn how to react to new families of the same contexts. This process is better known as self-adaptation through *Achieving*, and it has not been explored much in literature [29]. In other words, the systems *adapt, but they do not learn.* Going back to the example given at the beginning of this section. Assume that the total energy consumed at a given moment goes above 2 joules, and to adapt, the system reduces the frequency of sensing. However, this might affect the service accuracy of the system. Since the adaptation method does not have any mechanisms to learn from this experience, next time a similar situation happens, the same decision to reduce the sensing frequency is executed. To mitigate this, AI, in particular, machine learning techniques [15] can be considered towards aiding adaptation as it ensures that the system can learn from multiple scenarios and improve over a period of time. This was suggested by [29, 39], but not much work has been done in this direction.

3. Quality assurances to learning: As stated above, machine learning can be used for decision making in self-adaptation. It can further enable systems to learn and improve with every adaptation. However, one of the issues with machine learning is that it suffers from accuracy and bias [40, 41]. For example, recently, there were reports of a Tesla car in autopilot mode crashing into a barrier [42]. It further led to the death of the driver. The crash's main reason was attributed to the machine learning algorithm's failure in identifying the barrier. Similarly, in another scenario AI algorithms were found to be sending wrong people to prison in the US [43]. The main reason for this behavior was due to the bias in training data. These are some of the significant issues with machine learning in practice, and the same applies to the use of machine learning in self-adaptation. In fact, assuring quality has been listed as one of the significant challenges for using machine learning for decision making self-adaptation [29]. Hence, using machine learning techniques without quality guarantees for the decision-making process may produce erroneous decisions. Such decisions may further lead to sub-optimal adaptations, thereby affecting the execution of the overall system. Going back to the example presented at the beginning of this section, let us assume that as in the scenario stated above, the self-adaptive process based on the learning decides to reduce the data frequency due to high energy consumption. It might happen that there is a sudden instance of fire in the room, and the temperature needs to be sensed at a higher frequency. However, since the learning algorithm had not seen this issue before, it does not have the required knowledge to make the correct decision. This calls for techniques that can verify the machine learning algorithm's decision before executing it. Such techniques can further enable the learning process itself to improve based on the results of verification.

1.3 Research Questions

As we have seen, many challenges exist in architecting IoT systems (or any modern software systems in general), which can greatly impact the overall QoS. Self-adaptation has emerged as a potential solution to mitigate these issues. However, they have their limitations, as listed above. One of the important characteristics of any modern systems, especially IoT, is the amount of data they generate [44]. This data includes the context data generated by the IoT devices and the QoS data (potentially extracted from the system's execution logs). This data availability enables the use of machine learning techniques to leverage the different data generated to predict the need for adaptation proactively and further autonomously decide on the adaptation strategy. However, as mentioned in the previous section, such techniques should also provide quality guarantees of the adaptation performed. Hence the overall goal of this research is to develop a data-driven approach for architecting self-adaptive IoT systems where, *Given an IoT system and a set of QoS requirements to be satisfied, the approach uses machine learning techniques to learn from the data obtained, generate an adaptation decision, verify the decision, adapt the architecture, learn from the feedback and keep iterating over this loop of learning, selection, verification, and adaptation thereby enabling the architecture to improve continuously such that the QoS of the system is not compromised.* This overall goal can be further decomposed into four main research questions, namely:

- **RQ1** *How to perform effective and efficient proactive adaptation using machine learning techniques?*
- **RQ2** *How can machine learning be used to improve the adaptation process continuously?*
- **RQ3** *How to guarantee the quality of the adaptation performed by such a machine learning process, and how can such guarantees, in turn, help the machine learning process?*
- **RQ4** *How can the approach be generalized to other class of software systems?*

While RQ1 is based on the first challenge of the reactive nature of adaptation, as mentioned in Section 1.2, RQ2 focuses on addressing the second challenge of enabling systems to learn with adaptations. RQ3, on the other hand, aims to identify mechanisms that can guarantee the quality of decisions produced by machine learning and further aid the learning process. Unlike the first three questions, RQ4 targets the approach's generalizability by understanding if the approach can be applied to a more general class of software systems.

1.4 Self-adaptation to Self-learning: Solution overview

To address the challenges in the field of self-adaptation and to further answer the research questions mentioned above, in this thesis, we develop a data-driven approach to architecting self-adaptive systems. The approach in principle shifts the focus from *self-adaptive architectures to self-learning architectures.* Going by the definition of machine learning [45], we define the learnability of software architecture as the ability such that *"Given a task T of solving an architectural problem, given the architecture A of the IoT system, and a set of quality attributes Q that models the QoS of the system, the architecture A of the system improves with Data D, such that the QoS Q of the system is not compromised"* [1].

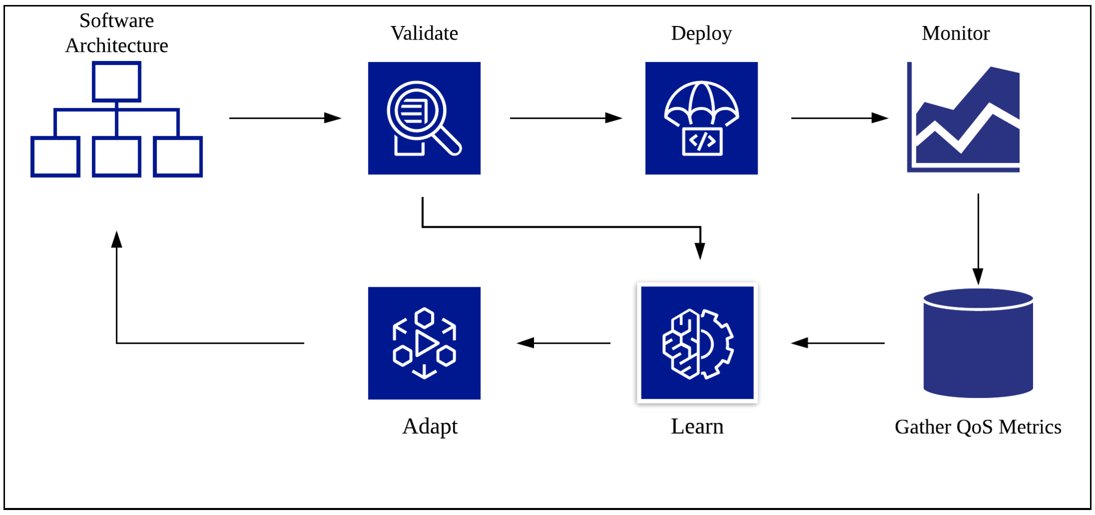


Figure 1.2: Self-learning software architectures: Solution Overview

Figure 2.1 shows the conceptual overview of the overall approach presented in this thesis [46]. As in the case of any software systems, at first, the system's architecture is designed, validated, implemented, and deployed. Post the deployment, at run-time, the *Monitor* process keeps a check on the system's QoS metrics such as energy consumption, data traffic, response time, etc., as well as on the data gathered by the system. The *Learn* process further processes the collected data to extract actionable insights. It uses machine learning techniques to proactively identify the need for adaptation and further decide the optimal adaptation strategy. It then communicates the decision to the *Adapt* process, which adapts the architecture. Additionally, to guarantee the adaptation's correctness, the adapted architecture is sent to a *Validate* process that checks if the modified architecture satisfies the QoS requirements. Furthermore, the *Deploy* process deploys the architecture. After every deployment, the approach uses the variation in QoS metrics as feedback to measure the quality of adaptation. This feedback and the feedback from the *Validate* process are used as feedback to the *Learn* process for further improvement. This process continues throughout the software lifecycle. In this manner, the approach identifies any possible QoS issues at a much earlier stage and adapts the architecture accordingly. Moreover, over time, the approach ensures that the architecture automatically learns to handle the uncertainties optimally.

1.5 Research Activities

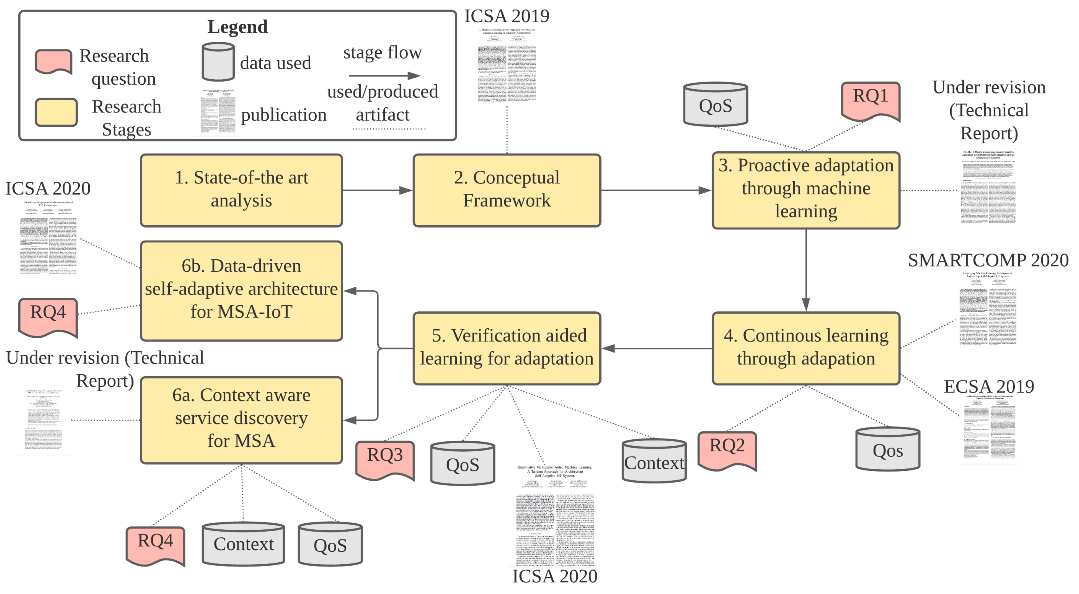

Legend
Research question
data used
stage flow
used/produced artifact
Research Stages
publication
ICSA 2019
QoS
RQ1
Under revision (Technical Report)
1. State-of-the art analysis
2. Conceptual Framework
3. Proactive adaptation through machine learning
ICSA 2020
6b. Data-driven self-adaptive architecture for MSA-IoT
RQ4
SMARTCOMP 2020
5. Verification aided learning for adaptation
4. Continous learning through adapation
Under revision (Technical Report)
ECSA 2019
6a. Context aware service discovery for MSA
RQ3
QoS
Context
RQ2
Qos
RQ4
Context
QoS
ICSA 2020

FIGURE 1.3: Stage by stage development of the overall research

Figure 1.3 shows an overview of the different activities conducted during our research. We represent them using six stages. In the first stage, we performed a *state-of-the art analysis*. This process involved gathering the different works that have been done in the field of self-adaptation. These works were further analyzed to understand the different techniques to architect self-adaptive systems, challenges, and limitations. Based on this, we developed the overall research goal. This goal, the research challenges, and our domain knowledge were used in the second stage to formulate a conceptual approach to achieve the overall research goal. The resulting approach was published in the proceedings of the International Conference on Software Architecture (ICSA 2019) [1]. The rest of the stages focused on realizing the different parts of the conceptual approach to answer the various research questions.

Stage 3 of our research focused on the development of a proactive approach to architect self-adaptive IoT systems. The approach mainly leveraged the QoS data, in particular energy consumption data. This was accomplished using deep neural networks that were used to generate short-term and long-term forecasts of the expected QoS of a given IoT system. Further, the approach was extensively evaluated on an IoT system based on a case study. The stage resulted in a journal work which is currently under revision [2].

In stage 4, the research was extended to combine the proactive forecasts with automated decision making to generate a machine learning-driven proactive decision-making approach. It leveraged two types of QoS data, namely data traffic, and energy consumption. The approach further enabled the architecture to adapt and learn continuously based on the feedback in the form of forecasts obtained after every adaptation. The approach was further evaluated on an IoT system and was published in the proceedings of the International Conference for Smart Computing (SMARTCOMP 2020) [4]. The tool resulting from the approach was published in the proceedings of the European Conference on Software Architecture (ECSA 2019) [3].

While the research activity of stage 4 focused on the continuous learning part, stage 5 combined proactive learning with quantitative verification to provide quality guarantees to the machine learning decisions. It took three different data types into consideration: two types of QoS data (energy and data traffic) and context data (acquired by the sensors). The approach further enabled machine learning to continuously improve based on the feedback from verification and forecasts. After evaluations of the approach on a case study, the work done was published in the proceedings of the International Conference on Software Architecture (ICSA 2020) [5].

In the final stage of research, stage 6 focused on extending the approach to a more general class system. The stage, in-turn, consisted of two parallel activities. The first one (stage 6a) focused on extending the proactive learning approach developed in Stages 3 and 4 to develop a machine learning-driven service discovery mechanism for microservice architectures. The approach leveraged machine learning techniques on two different types of data, namely context and QoS data, in particular response time. The approach was evaluated on a prototype microservice application. The process also resulted in a potential research publication, and it is currently under revision [6]. The second activity (stage 6b) consisted of applying some of the concepts developed in the different stages to create a data-driven self-adaptive architecture for microservice-based IoT systems (MSA-IoT) and this was published in the proceedings of the International Conference on Software Architecture (ICSA 2020) [7]. Further, a short article on the concept of *Self-Learnable Software Architectures* based on all these different research activities was published in the quarterly magazine of European Research Consortium for Informatics and Mathematics (ERCIM News, referenced by DBLP[1]) [46].

[1]https://dblp.org

1.6 Thesis structure

The remainder of this thesis is organized as follows:

In Chapter 2, we provide background details of various concepts underlying this thesis. This chapter provides a detailed overview of self-adaptive systems, machine learning with a particular focus on neural networks and reinforcement learning, quantitative verification, microservices, and a detailed description of an IoT case study that we will be using throughout this thesis.

Chapter 3 provides a detailed analysis of the state-of-the-art (stage 1 in Figure 1.3). First, it describes the different works that apply self-adaptation in IoT systems and machine learning to self-adaptive systems. These form the basis for the approach presented in Chapter 4 and 5. It then discusses the works that combine quantitative verification, and machine learning, which forms the base works for the approach presented in chapter 6. Further, the chapter elaborates on the works done in service discovery in microservices and self-adaptation applied to microservice-based IoT systems. These form the state-of-the-art for the work presented in Chapter 7.

In Chapter 4, we describe the first part of the approach (stage 3 in Figure 1.3), which uses machine learning techniques to perform effective and efficient proactive adaptation in IoT systems. The chapter addresses the research question, RQ1.

Chapter 5, describes the second part of the approach (stage 4 in Figure 1.3) which further extends the proactive approach presented in Chapter 4 to support automated decision making and continuous learning using machine learning. The chapter further addresses the research question RQ2.

Chapter 6 provides answers RQ3 (stage 5 in Figure 1.3) by describing the third part of the approach, which extends the proactive machine learning-driven decision-making approach presented in Chapter 5 with quantitative verification techniques.

In Chapter 7, we address RQ4 (stage 6 in Figure 1.3) by applying the approach to perform machine learning-driven context-aware service discovery in microservice architectures. Further, the chapter also describes a data-driven self-adaptive architecture for microservice-based IoT applications.

Finally, we conclude the thesis in Chapter 8 by summarizing the overall contributions of this thesis. We also provide directions for future work.

Additionally in Appendix A, we have described in detail, the different technologies used for the implementation of the approach presented in this thesis.

1.7 Research Publications

The research presented in this (as depicted in Figure 1.3) has resulted in the following peer-reviewed publications[2]:

- Muccini, Henry, and Karthik Vaidhyanathan. ”A machine learning-driven approach for proactive decision making in adaptive architectures.” In 2019 IEEE International Conference on Software Architecture Companion (ICSA-C), pp. 242-245. IEEE, 2019 (*Thesis author contribution:* Overall idea, approach framework and writing under the guidance of the supervisor).
- Muccini, Henry, and Karthik Vaidhyanathan. ”ArchLearner: leveraging machine-learning techniques for proactive architectural adaptation.” In Proceedings of the 13th European Conference on Software Architecture-Volume 2, pp. 38-41. 2019 (*Thesis author contribution:* Tool implementation, experiment setup, evaluation, and writing under the guidance of the supervisor).
- Muccini, Henry, and Karthik Vaidhyanathan. ”Leveraging Machine Learning Techniques for Architecting Self-Adaptive IoT Systems.” In 2020 IEEE International Conference on Smart Computing (SMARTCOMP), pp. 65-72. IEEE, 2020 (*Thesis author contribution:* Overall idea, methodology, algorithm, implementation, experiment setup, evaluation, and writing under the guidance of the supervisor)
- Cámara, Javier, Henry Muccini, and Karthik Vaidhyanathan. ”Quantitative Verification-Aided Machine Learning: A Tandem Approach for Architecting Self-Adaptive IoT Systems.” In 2020 IEEE International Conference on Software Architecture (ICSA), pp. 11-22. IEEE, 2020 (*Thesis author contribution:* Overall idea along with other authors, machine learning framework of the overall methodology, implementation of the different machine learning algorithms, integration of machine learning with verification, experiment setup, evaluations, and equal contribution to writing).
- De Sanctis, Martina, Henry Muccini, and Karthik Vaidhyanathan. ”Data-driven Adaptation in Microservice-based IoT Architectures.” In 2020 IEEE International Conference on Software Architecture Companion (ICSA-C), pp. 59-62. IEEE, 2020 (*Thesis author contribution:* Overall idea conceptual methodology along with the other authors, machine learning part of the methodology, the overall architectural framework with other authors and equal contribution to writing).

[2]Author names are listed in the alphabetical order

The following publications are under revision and are available as technical reports[3]:

- Henry Muccini and Karthik Vaidhyanathan, ”PIE-ML: A Machine learning-driven Proactive Approach for Architecting Self-adaptive Energy Efficient IoT Systems”, DISIM, University of L’Aquila, L’Aquila, Italy, TRCS: 002/2020, July. 15, 2020.[Online]. Available: https://tinyurl.com/y98weaat (*Thesis author contribution:* Overall idea, development of the IoT system, implementation of the managing system and machine learning algorithms, evaluation of experimental results and writing under the guidance of the supervisor).
- Mauro Caporuscio, Marco De Toma, Henry Muccini and Karthik Vaidhyanathan, ”A Machine Learning-driven approach to Service Discovery for Microservice architectures”, DISIM, University of L’Aquila, L’Aquila, Italy, TRCS: 003/2020, September. 20, 2020.[Online]. Available: https://tinyurl.com/y67krcn9 (*Thesis author contribution:* development and implementation of the machine learning algorithms, evaluation of experimental results, and equal contribution to writing).

[3]Author names are listed in the alphabetical order

Chapter 2

Background

This chapter provides a brief overview of the main concepts and techniques underlying the research works presented in this thesis. We begin this chapter by providing an overview on *Self-adaptive systems* (Section 2.1). This is followed by an overview of machine learning (Section 2.2, with in-depth details on some of the machine learning techniques used in this thesis. In this research, we use quantitative verification to verify the correctness of machine learning. To better understand this, in Section 2.3, we provide some background details on quantitative verification techniques, in particular probabilistic model checking. The generalizability of our approach is demonstrated by extending it to solve some of the challenges in microservices. Hence, in Section 2.4, we provide an overview of microservice architectures. Finally, in Section 2.5, we provide details on the case study that has been used throughout this thesis for evaluations.

2.1 Self-adaptive systems

The field of *Self-adaptive systems* evolved over the years from the term *"Software Crisis"* which was first coined in 1968 at the NATO Software Engineering Conference in Brussels. The term was mainly referring to the issues related to the management of Software projects and the software's problem of not delivering its objectives. [47]. As the field of Software Engineering progressed, different tools and programming paradigms evolved, which ensured that the problems of software crisis are in control under the hands of software architects, project managers, and software developers. On the other hand, as the software's complexity kept increasing day by day, the cost of handling the complexity also increased [48]. The major reason for this trend is because the software components were becoming more and more heterogeneous, and the goal/run-time requirements of the software keep changing with time. This was referred to as the *Looming Software*

Complexity Crisis in 2003. This trend has been clearly observed in the last decade with software becoming more pervasive in nature, especially with the advent of the Internet of Things. One of the major consequence of this evolutions, as stated by [49] has been that the software systems need to become more flexible, versatile, flexible, resilient, dependable, robust, energy-efficient, recoverable, customizable, configurable, and self-optimizing by adapting to changing operational contexts, environments or system characteristics. These requirements have resulted in the rapid growth of research in self-adaptive systems.

In the reminder of this section, we provide an in-depth overview of self-adaptive systems starting from the different definitions of the various frameworks that can be used to develop a self-adaptive system.

2.1.1 Definitions

Kephart and Chess came up with a concept of *Self-management* as the only way to tackle the complexity crisis [50]. *Self-management* refers to the systems that can adapt autonomously to achieve their goals based on high-level objectives. These systems are also called as *Self-adaptive systems* [32]. There has been no single definition for self-adaptive systems, especially for self-adaptive software in literature. Few of the definitions are :

- Self-adaptivity is the capability of the system to adjust its behavior in response to the environment. The ”self” prefix indicates that the systems autonomously decide (i.e., with minimal or no interference) how to adapt or to organize themselves so that they can accommodate changes in their contexts and environments. While some self-adaptive systems may be able to function without any human intervention, guidance in the form of higher-level objectives (e.g., through policies) is useful and realizable in many systems [39]
- Self-adaptive software can evaluate and change its own behavior whenever the evaluation shows that the software is not accomplishing what it was intended to do, or when better functionality or performance may be possible [33].

2.1.2 Conceptual Model of a Self-Adaptive System

The figure below shows the conceptual model of a self-adaptive system [29][32] and it basically consists of four components :

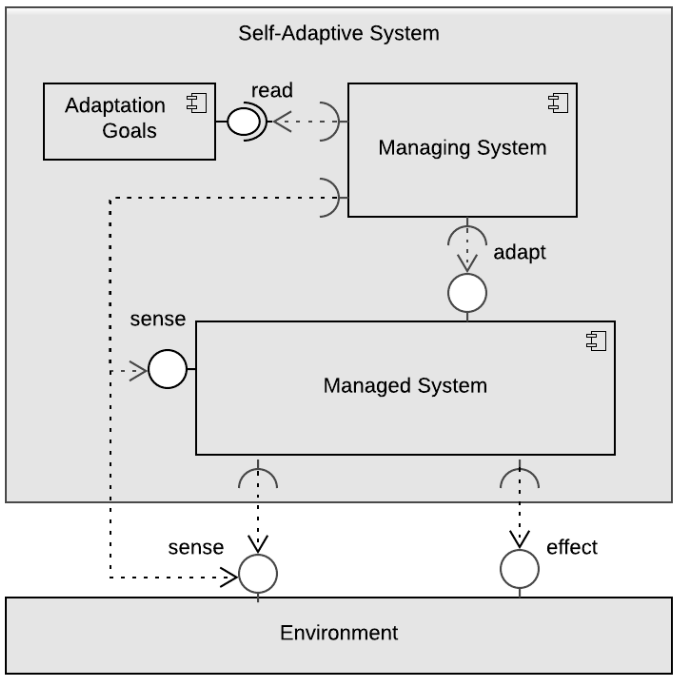


Figure 2.1: Conceptual Model of a self-adaptive system

1. Environment

The environment refers to anything that is not under the control of software and the outcome of which can affect the software. It generally represents the external world with which the self-adaptive system interacts. The environment comprises both the physical as well as the virtual entities. For example, in an IoT system, the environment represents the space with which the sensors interact, such as a room in a house being monitored using a camera. In this case, the environment also comprises the camera and the software drivers running in this camera.

2. Managed System

The managed system mainly refers to the application code that implements the functionality expected out of a system. The concerns of the managed system are generally the concerns of the environment. They are responsible for sensing the environment using sensors to observe changes. In the event of any uncertainties, they are responsible for making the necessary changes in the environment with the help of actuators. Different frameworks refer to the managed system using different terminologies. The MAPE-K refers to this as *Managed Element* [50]. The Rainbow framework coins it as *System Layer* [51]. The 3 layer reference framework terms it as *goal management* [52] and the FORMS reference model refers to it as *base level subsystem.*

3. Adaptation Goals

These can basically be viewed as the concerns of the managing systems over the managed system. The goals essentially focus on the software quality attributes of the managed system. Four different adaptation goals have been defined in literature, *Self-Configuration*, *Self-Optimization*, *Self-healing* and *self-protection.* [50] [32].

4. Managing System

The managing system is responsible for managing the managed system. It comprises of the adaptation logic depending on the adaptation goals. It is responsible for continuously monitoring the environment. It then adapts the managed system based on the changes in the environment. This is referred to as *Autonomic manager* in MAPE-K [50], *Architecture Layer* in the Rainbow Framework [51], *Controller* in 3 layer reference framework and *Reflective subsystem* in FORMS reference model.

2.1.3 Engineering Self-adaptive Systems

Different ways to engineer self-adaptive systems have been proposed. This was categorized into six waves by *Waynes et al.* in [32] and table 2.1 summarizes the different waves, the corresponding uses, the motivation behind the approach, the framework that can be used for realization.

The approach presented in this thesis is based on the MAPE-K framework for performing self-adaptation. We selected MAPE-K considering the problem in hand and the set of functionalities supported by the MAPE-K framework. Further, we provide an overview of MAPE-K in the following subsection.

2.1.4 The MAPE-K Framework

The concept of MAPE-K control loop was first introduced by Kephart and Chess in [50], which was further presented by IBM in [55]. The idea of using MAPE-K in the context of self-adaptive systems was later discussed by Brun et al. [33]. It consists of four key activities namely *Monitor (M), Analyze (A), Plan (P)* and *Execute.* Further, these activities share a common *Knowledge (K)* base. In the context, of a self-adaptive systems (refer Figure 2.1), the combination of these activities are used to accomplish the functionalities of the *Managing System.*

Figure 2.2 represents the conceptual view of a self-adaptive system that implements the managing system using a MAPE-K control loop. The key activities of the MAPE-K are as follows:

Waves	Uses	Motivation	Framework
Automating Tasks	Automation of management tasks from human administrators to machines	Management of the system by human administrators are often complex and error prone process.	MAPE-K [50]
Architecture-Based Adaptation	Identification of key concerns of Self-Adaptation, Systematic management of runtime changes and software complexity	Need for Systematic perspective for engineering self-adaptive systems.	3-Layer Architectural Model [52]
Models at Runtime	Run-time adaptation and decision making	Concrete realization of architecture based adaptation is complex due to a large amount of information gathered at runtime.	Model-Oriented Architecture [32]
Goal Driven Adaptation	Handling adaptation in systems exposed to uncertainties	The difficulty in managing requirements when the system is exposed to uncertainties.	Goal Model [32]
Guarantees Under Uncertainties	Provides guarantees for the compliance of adaptation goals for systems under uncertainties	Ensuring that the system will meet a goal once exposed to uncertainties.	QoSMOS Architecture [53]
Control-Based Approaches	Application of control theory to perform adaptation	Need for formal design and verification of self-adaptive systems.	PBM [54]

Table 2.1: Different Engineering approaches to Self-Adaptive Systems

1. *Monitor:* The monitor activity is responsible for continuously monitoring the managing system by regularly collecting different types of execution data, execution metrics, logs, etc. of the *Managed system.* It further sends them to the *Analyze* activity.

2. *Analyze:* The Analyze activity comprises different functionalities that can further identify the need for adaptation based on the data obtained from the *Monitor* activity. It uses adaptation goals (as defined in Section 2.1.2) to accomplish this. On identifying the

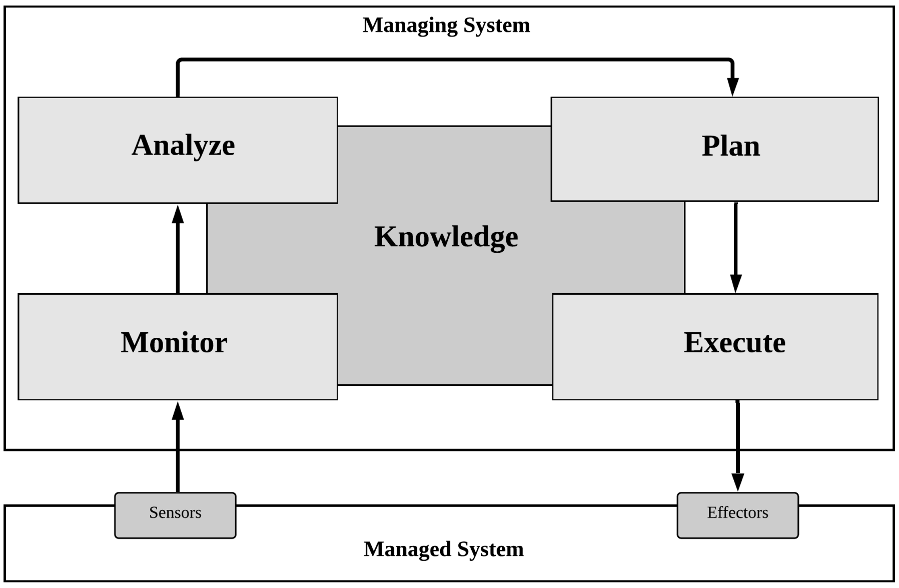


FIGURE 2.2: Self-adaptive System based on MAPE-K

need for adaptation, it immediately triggers the *Plan* activity to generate an adaptation plan.

3. *Plan:* The responsibility of Plan activity is to generate an adaptation plan/strategy which can be applied to the system to satisfy the overall goals. Such an adaptation plan can range from a simple reconfiguration action to complex strategies. To achieve this, it uses different techniques such as model-checking, model-driven techniques, machine-learning, control theory, etc. The plan generated is further forwarded to the *Execute* activity.

4. *Execute:* As the name suggests, the role of the *Execute* activity is to execute the adaptation plan as suggested by the *Plan* activity on the *Managed System.* It results in the change in behavior/structure of the *Managed System.* It achieves this with the help of Effectors, which are similar to simple interfaces. They support dynamic adaptation.

5. *Knowledge:* It acts as a common shared space that consists of the data that can be shared by the different activities of the MAPE. These data may consist of execution logs, data gathered by the *Monitor* activity, models for the *Analyze* activity, policies that are needed by the *Plan* activity for the generation of an adaptation plan, and knowledge gathered post the execution of an adaptation.

2.1.5 Architectural Patterns for Self-adaptation in IoT

In general, the type of adaptation performed can be categorized into two main categories: i) Behavioral adaptation, where the performed adaptation results in reconfigurations on

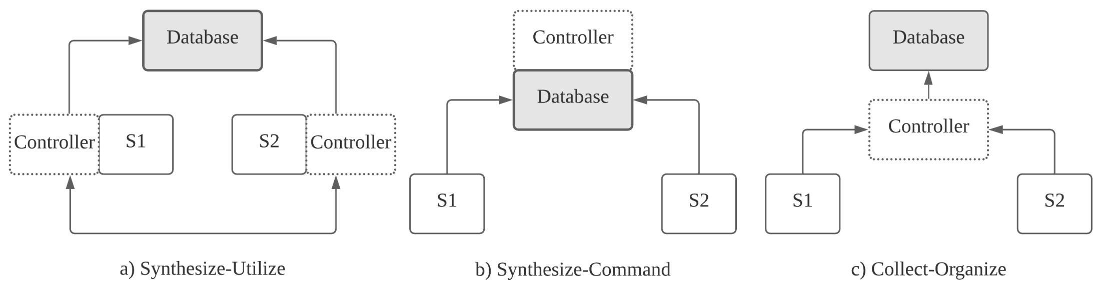


Figure 2.3: Architectural Patterns for Self-adaptation in IoT

the system behavior. For example, there might be a batch process in the cloud that gets executed every 1 hour and results in high costs. A simple reconfiguration in this scenario can be to change the frequency to 3 hours. ii) Structural adaptation is where the adaptation results in the change in the structure of the underlying system. For example, all the components in a given system might be communicating to a central cloud resulting in a central point of failure. A simple reconfiguration can introduce an additional component that can provide the system with semi-decentralized processing capabilities. In this direction, in literature, Musil et al. in [56] proposed three different architectural patterns for self-adaptation in IoT or cyber-physical systems in general.

Each of the three patterns consists of an application layer, a service middleware layer, a communication layer, a proxy layer, and a physical layer. Figure 2.3 gives a high level overview of the different architectural patterns applied to IoT. $S1$ and $S2$ denotes the sensors and *Controller* denotes the MAPE controller responsible for making decisions based on the data received from the sensors. These patterns include:

1. Synthesize-Utilize (SU): This is a fully decentralized pattern which consists of autonomous entities on the application layer and the MAPE-K based adaptation mechanism on the service middleware layer. This layer receives data from physical resources and autonomous entities. It then periodically sends data to the application layer that the autonomous entities further use to perform adaptation. For example, In a typical IoT architecture, this pattern can be intuitively thought of as an edge-centered pattern (fig. 2.3a)) which will provide each sensor node with the ability to perform decision-making without resorting to an external controller.

2. Synthesize-Command (SC): This is a fully centralized pattern which consists of autonomous entities on the service middleware layer, adaptive algorithms on the proxy layer, and self-organization mechanisms on the application layer. For example, an IoT architecture based on this pattern will behave similarly to a cloud-centered pattern (fig. 2.3b)). All the sensor nodes and other components communicate to a central server, which will then perform the decision-making process.

3. Collect-Organize (CO): This is a semi-decentralized pattern which consists of multi-agent systems on the application layer which handles the physical resources and MAPE-K based adaptation mechanism on the service middleware layer. For example, an IoT architecture with this pattern will use additional controller components similar to a fog-centered pattern (fig.2.3c)), to receive the data from the sensor nodes and for performing decision making or in other words, there will be multiple controllers which will act as the local coordinator for a group of sensor nodes.

2.2 Machine learning

The rapid advancements in computing resources and the increased availability of on-demand cloud resources have led to the increased adoption of AI techniques.

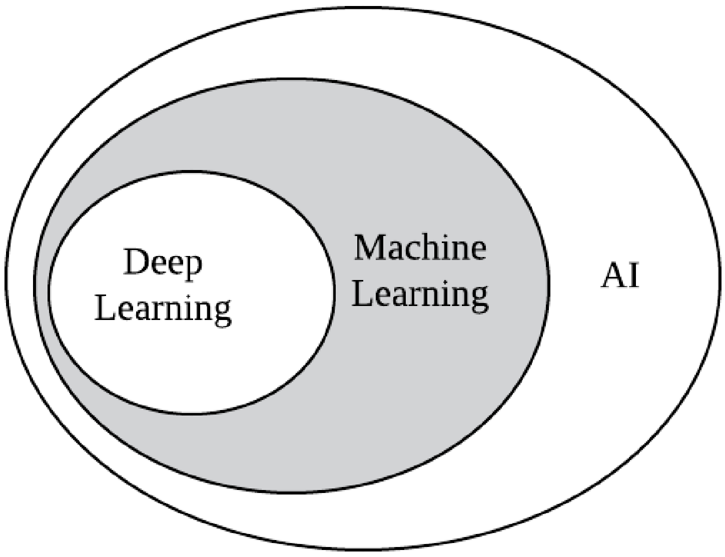


Figure 2.4: Venn diagram representation of the overall field of AI

AI is one of the most active research fields with a plethora of practical applications [16]. Figure 2.4 shows the Venn diagram depicting the different areas of AI. Among the different areas of AI, Machine Learning (ML) has gained popularity over the years due to its broad application scope. In other words, most of the AI techniques that are in use belong to the specific class of ML techniques [16]. In the following section, we provide an in-depth overview of ML techniques starting from the different definitions from the literature to specific techniques used in the remainder of this thesis.

2.2.1 Definitions

Many different definitions for ML has emerged over the years. In simplistic terms, ML is a class of AI techniques that allows a system/software/process to improve with experience continuously. A few of the more concrete definitions for ML are as follows:

- One of the key definitions of ML was given by Tom Mitchel [45]. Machine Learning is a process where a program is said to learn from Experience E, with respect to some class of task, T, and Performance measure P, if its performance at its task, T measured by P, improves with Experience E.
- Another definition was given by Alpaydin in [15]. Machine learning is programming computers to optimize a performance criterion using example data or past experience.

2.2.2 Types of Machine Learning

ML techniques fall into different categories based on how the learning is performed. Louridas et al performed such a classification in [57], where the authors classified ML types into primarily supervised and unsupervised. We further extend this was based on [58–61] into four categories namely: *Supervised Learning*, *Unsupervised Learning*, *Semi-supervised Learning* and *Reinforcement Learning*. This is represented in Figure 2.5[1]. As we can see from the figure, each category further consists of different types of algorithms that can be used based on the type of learning tasks to be solved and other information such as the problem domain, statistical nature of the data, etc. A brief overview on each of the four categories of ML is provided below:

Supervised Learning: As the name suggests, it is a type of learning in which a process/program learns how to perform a specific task using different types of well-defined examples (in the form of data). In other words, the algorithm is given a set of data consisting of inputs and corresponding outputs. This data is known as *training data*, and each attribute in the training data is called a feature. The objective of the algorithm is then to generate a function that maps the input to the output. Supervised learning is used for solving two types of problems: i) Classification and ii) Regression.

Classification refers to the process of classifying a given input into one of the predefined categories. In this type of problem, the algorithm is provided with data samples consisting of different input features and an output variable representing a categorical

[1]SVM - Support Vector Machines, PCA - Principle Component Analysis, MDP - Markov Decision Process, ANN - Artificial Neural Networks

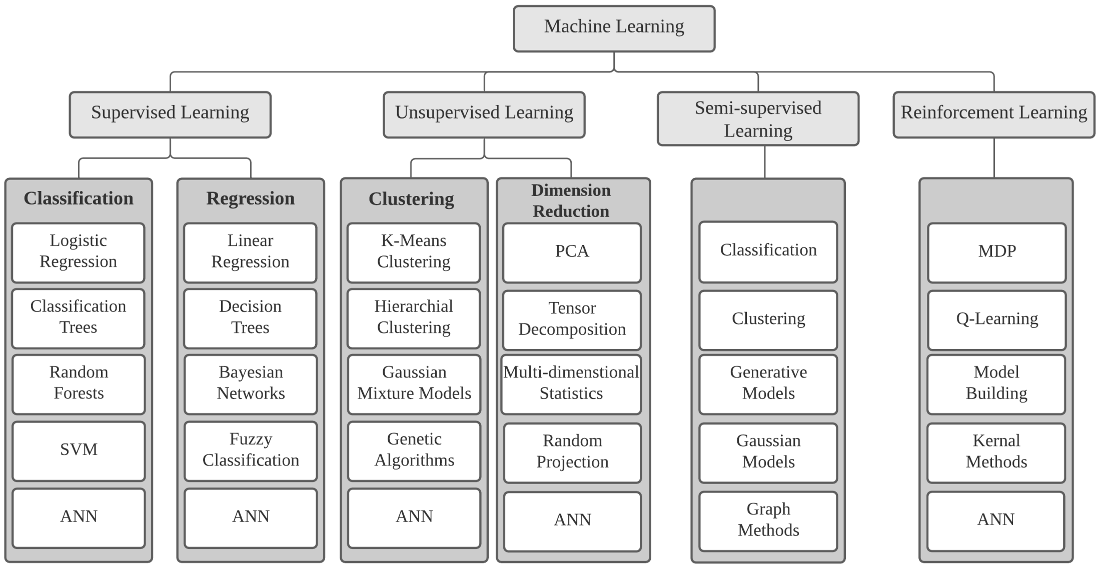


FIGURE 2.5: Types of Machine Learning

value. The algorithm's objective then is to generate a mapping function that can map the input to output such that given a new input, the algorithm will be able to predict the expected category. Many real-life problems are solved using classification algorithms, such as prediction of rain (yes or no), disease prediction, etc.

Regression, on the other hand, refers to the problem of predicting a real number based on a given input. In this type of problem, the algorithm will be provided with data samples consisting of different input features and an output variable representing a real number. In this case, the algorithm's objective is to generate a mapping function between the input features and the output (real number) such that given a new input, the algorithm will be able to predict the expected value. Some of the real-world examples of regression include weather forecasting, stock prediction, etc.

Unsupervised Learning: Contrary to Supervised Learning, it is a type of ML where the program/process learns to perform a given task using training data with no information on output labels/real-number. In other words, the algorithm is given a training data consisting just of inputs with no information on expected output (as in supervised learning). The algorithm's role is then to automatically infer the pattern/identify the hidden relationship among the input data such that given a new input data, the algorithm understands the action to be performed on the data. Unsupervised learning is generally used for solving two classes of problem: i) Clustering and ii) Dimensionality Reduction.

Clustering is a process of dividing a given set of input into different groups based on the similarities in the data. In this type of problem, the algorithm is provided with data samples consisting of unlabelled inputs. The algorithm's objective is then to identify the relationships among the input data samples to divide them into different categories. Given a new input, the algorithm will use the learned knowledge to group the new data into one category or generate a new category. Clustering techniques have a lot of practical applications. These include separating the news articles into different categories (Google), fake news identification, etc.

Dimensionality Reduction is a process of reducing the number of features (dimensions) in the input training data by removing irrelevant or redundant features. These techniques are generally used in pre-processing the training data. The objective of these techniques is to identify the relationships among the training data features, such that they can identify the independent features or the features that best represent the data without much loss of information. For example, a training dataset with a feature "date of birth" does not require another feature, "age". These techniques are mainly used in image processing tasks to reduce the input features in the training dataset.

Semi-supervised Learning: As the name suggests, this is a type of ML which falls in between supervised and unsupervised learning. In this technique, a program/process learns to perform a given task using training data consisting of both supervised and unsupervised input samples. The algorithm is given training data consisting of some labeled data (data consisting of inputs features and corresponding outputs) and many unlabelled data. The algorithm's role is to infer the relationship among the unlabelled data based on the labeled data samples. They are used for different classes of problems, including regression, classification, and clustering. Semi-supervised learning is used in scenarios where manual labeling of the entire training data is a tedious task. It has a lot of practical applications. These include Speech analysis (manual labeling of different audio samples can be a challenging task),web-content classification, etc.

Reinforcement Learning:, As opposed to traditional types of learning as discussed above, this type of ML is also called "learning with a critic" [15]. In this technique, the process/program learns to accomplish a task by trying a possible action that can be performed to achieve the task. It then receives feedback for the action performed. Based on the received feedback, the process/program tries to select a different (better) action when given the same task again. This process continues, and the process/program keeps learning. In other words, given a task, a set of actions that can be performed to complete the task and a set of rewards for every state reached by performing the action, the algorithm's objective at every point is to select the best action that maximizes the overall reward.

Among the different ML techniques mentioned above, the most commonly used one where extensive research is being done is supervised learning [16]. There has also been substantial research growing recently in unsupervised or rather self-supervised learning[2]. In this thesis, we will be using two main ML techniques, Recurrent Neural Network (RNN), in particular Long Short Term Memory Network (LSTM - supervised learning) , a class of Artificial neural networks, and Q-learning (Reinforcement learning) in the rest of the thesis. These are further explained below.

2.2.3 Artificial Neural Networks

An artificial neural network (ANN) is similar to an information processing system with characteristics similar to a biological neural network. McCullough and Pitts invented the first neural network model in 1944 [62]. It was called the McCullough Pitts model or the MCP model. This invention formed the basis of modern machine learning. Any artificial neural network consists of the following components:

- A set of simple information-processing units called as *neurons*
- The neurons are connected using links (like edges in a directed graph), and each link has an associated weight (like weights in a graph). The weight is representative of the amount of relevance given to a particular connection link.
- Each neuron processes the input information by applying an activation function (a non-linear function) to produce the output.

Preliminaries

Before we go further, in this section, we provide an overview of some of the commonly used terminologies while using neural networks. To better understand the terminologies, we explain the different processes involved in building a neural network through a small example. Figure 2.6 shows the image of a simple neuron with multiple inputs represented by X_1, X_2 and X_3 and an output value Y. w_1, w_2 and w_3 represents the weights given to the different inputs. These input variables are called as *features*. As shown in the figure ((fig. 2.6), the value of Y is determined by applying a function, f on the sum of the input features multiplied by their respective weights, and a constant value b.

The function, f applied on the weighted sum of input features, is known as *activation function*. As the name suggests, it determines if a neuron needs to be activated or not and how it has to be activated. There are different types of activation functions such as *Rectified Linear Unit (ReLU)*, *logistic sigmoid, tanh, softmax*, etc. In this thesis, we

[2]https://tinyurl.com/y2wsqvop

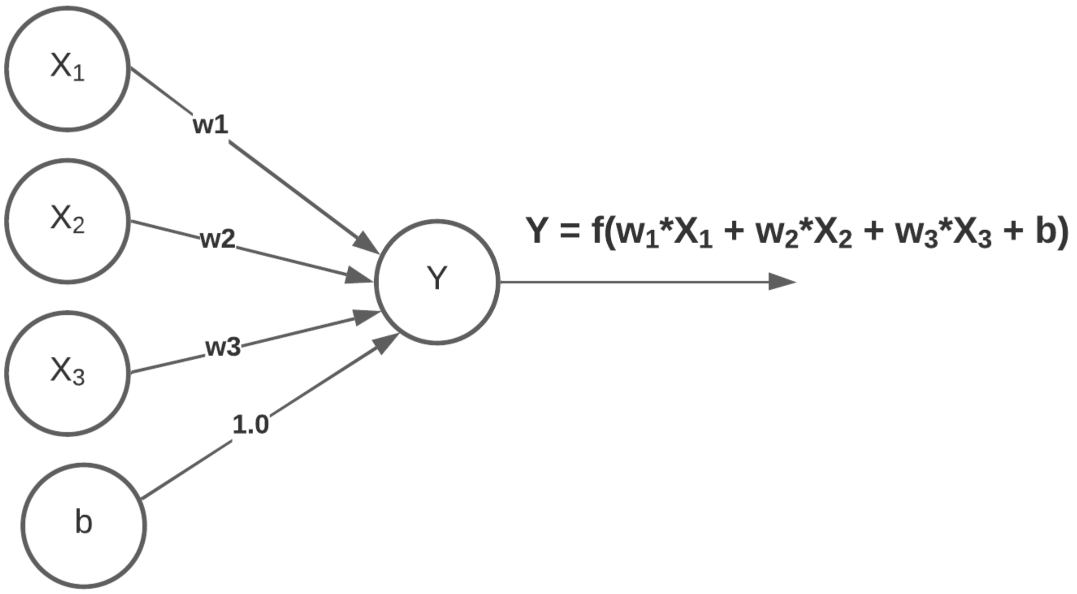


FIGURE 2.6: Simple neural network

will be using two specific activation functions, namely *sigmoid* and *tanh*. The equation below represents the sigmoid activation function:

$$f(x) = \frac{1}{1 + e^{-x}} \tag{2.1}$$

The sigmoid activation is used in scenarios (well used in classification problems) where we need f(x) to be in a range (0,1). This is useful where the output of Y should be a value between 0 and 1 or just a binary value. For example, depending on the value of f(x), the output can be classified as 1 if f(x) $\geq$ 0.5 and 0 if f(x) $<$ 0.5. On the other hand, the equation below represents the tanh activation function:

$$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{2.2}$$

The tanh activation is useful in scenarios where f(x) needs to be in range (-1,1). It gives more output range compared to a sigmoid and is also very well used in classification problems. The tanh is sometimes used as an alternative to sigmoid or in combination with a sigmoid (in complex neural networks). While sigmoid maps negative inputs to near-zero values, tanh maps negative inputs to negative values due to the inherent nature of the function. This property of tanh is beneficial in scenarios where such flexibility is needed.

The constant value, b depicted in figure 2.6 is known as *bias* and it helps the algorithm to converge towards optimal solution. In other words, b is similar to the constant, C that appears in a linear equation $Y = mx + C$. It provides better control over the output.

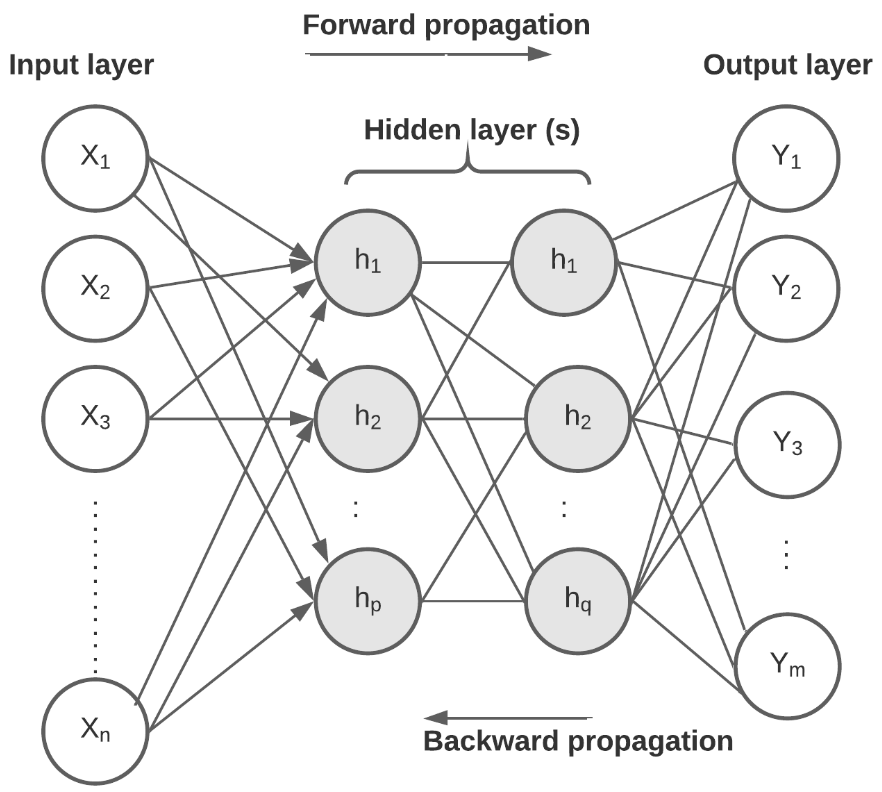


FIGURE 2.7: Conceptual view of a deep neural network

For example: Assume that if b does not exist in the previous equation (fig. 2.6) and one of the features, X_1 is 0. Then this means that the output will be strongly influenced by X_2 and X_3. Such an issue can be controlled by using a constant term, b, which enables the neuron to produce more reliable outputs.

The learning process happens when the neuron is provided with different data samples, each consisting of a set of $\{X_1, X_2, X_3\}$, along with the expected Y value. This process is known as *training*. The goal of training is to identify the best values of w_1, w_2 and w_3 that can map a given set of input, X to Y.

What we have seen above is a representation of a simple neural net. The advancements in the theory of machine learning and computing infrastructure, coupled with the high availability of data, have led to the birth of complex neural networks consisting of millions of neurons, thereby giving rise to deep learning [16]. Figure 2.7 shows the conceptual view of a deep neural network. It mainly consists of three layers: i) The input layer consists of a set of neurons which are responsible for handling the input data and passing to the hidden layer; ii) Hidden layer(s) (a typical deep neural network consists of 1 or more hidden layers) consists of a set of neurons which processes the input data using one or more activation functions to generate output and iii) Output layer consists of one or more neurons which provides the output by processing the data obtained from the hidden layer.

Once the data moves from the input to the hidden layers during the training process, it moves through the hidden layers to the output layer. This process is known as *forward propagation.* The neural network where the data only moves in one direction, i.e., from input to the hidden layers to the output (no cyclic information flow), is known as a *feedfoward neural net.* Most such neural networks also have an additional information flow from the output layer back to the hidden layers. This is better known as *back-propagation* [63]. This backward flow allows the hidden layers to adjust their weights appropriately based on the feedback from the output layers.

Having looked at some of the basic terms and terminologies in neural networks, let us look at some key terminologies which we will be in the remainder of this thesis.

Training data: It represents the set of data samples given as input to the neural network during training to enable learning.

Model: It is the output of the training process. It represents the mathematical function and the values of the parameters obtained as an output of training. For example, if the neural network represented in figure 2.6 is trained with some training data then the process will result in the generation of f with values for w_1, w_2 and w_3. This function f with the values of W represents the model and the values of w_1, w_2, w_3 forms the *model parameters* or *model weights.*

Epochs and Iteration: The training dataset usually consists of a large number of samples. The dataset is further split into smaller chunks known as *batches.* Then during training, each batch is fed into the neural network one by one. The process of passing each batch of training data repeatedly through the neural network such that with every pass, the model updates its parameters is known as *Iteration. Epoch*, on the other hand, refers to the pass of the entire training dataset repeatedly through the neural network. Therefore, each epoch comprises of multiple iterations depending on the number of batches (batch size).

Loss Function and Optimization: The purpose of training a neural network is to reach a point (*convergence*) where the difference between the predicted value and the actual value is as minimal as possible. This difference is known as *loss* and it is often expressed using a mathematical function which is better known as a *loss function.* Some of the commonly used loss functions are *Mean Absolute Error*, *Root Mean Squared Error*, etc. [16]. *Optimization* is the process which enables the minimization of the loss. Commonly used optimization methods include *Gradient Descent* [64], *Adam optimization* [65], etc.

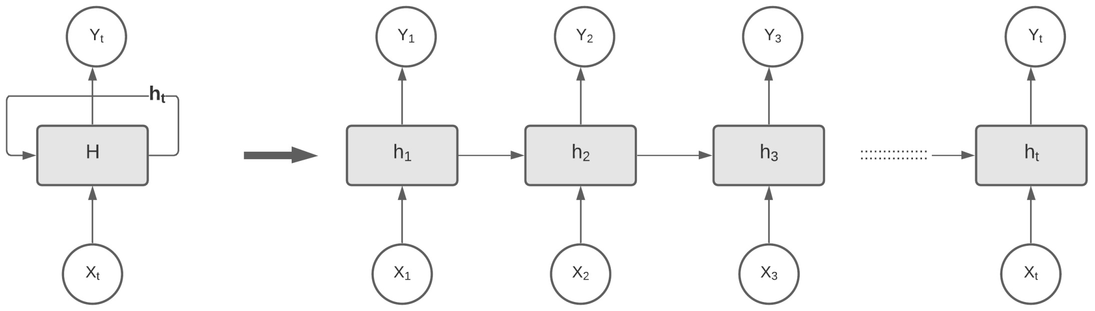


Figure 2.8: Conceptual View of an Unrolled Recurrent Neural Network

Testing Data and Cross-Validation:, One of the important aspects of developing a ML model is to test the quality of the trained model. Before training a model, the input data is divided into two datasets: i) Training data and ii) Testing data. Once a model has been trained, it is then applied to the testing data to test the accuracy, precision, etc. of the model. One thing that might happen in this scenario is that the model will work very well on the *training data* and not well on the testing data. This problem is known as *Over fitting* [66]. This problem is better solved using a technique known as *cross validation.* As per this technique, the training data, or rather the input data, is partitioned into n subsets, usually of equal size. The training is performed on $n-1$ subsets, and the n^{th} subset is used as a testing or validation set. This process is repeated until all subsets form a validation set at least once.

Having looked at some preliminaries and important terminologies which we will be using throughout this thesis, in the following subsection, we provide details of a particular neural network, LSTM, a class of RNN, which forms one of the central parts of the approach presented in this thesis. Following this, we provide some background on Q-learning, a class of Reinforcement learning that forms the next central part of the approach.

2.2.3.1 Recurrent neural networks

Traditional neural networks do not have the ability to store or persist information in the network, which can be very useful in processing data like video streams, speech signals, etc. This issue arises due to the sequential nature of data, where data at a particular instant is dependant on its past instances. For example, assume that we want the neural network to predict the next word in a sentence ”yesterday was a sunny day, and hence I think today also it will be ...... ”. The output of this can be anything as unless and until the neural net can understand the context of the given text and predict the word ”Sunny,” but this will require the neural net store, understand the word ordering and further understand the dependency between the different words. This type of data is known better as *Sequential data* in which the ordering of data plays an important role in prediction or classification. This challenge was the key motivation behind developing a new class of networks known as Recurrent Neural Networks (RNN), which are networks with inbuilt loops that further allow them to persist information on the sequence order and handle the problem of dependencies.

Figure 2.8 represents the conceptual view of the architecture of an unrolled RNN[3]. They also have primarily three layers, namely, input, output, and hidden. However, unlike traditional ANN, as we can see from the figure, there is a flow of information from one neuron (better known as a recurrent cell) in the hidden layer to another in the same layer (in traditional ANN, this will be from one neuron in one layer to neuron in the next layer). This flow allows the transfer of information from one step to another. It is due to this recurrent flow of the information they are known as RNN.

Working: Let X_t denote the input vector, h_t the hidden state and Y_t the output vector at an instance t of a sequence. Then the value at hidden state is updated using the formula:

$$h_t = f(W_{hh}^T * h_{t-1} + W_{xh}^T * x_t) \tag{2.3}$$

As it can be seen, the update is performed using a recurrence equation, where f denotes the activation function ($tanh$ is the most commonly used one in the case of standard RNN's); W_{hh}^T denotes the weight of the connection from one hidden cell (neuron) to another; W_{xh}^T denotes the weight of the connection from the input to the hidden cell, and h_{t-1} denotes the value of the previous hidden cell. In essence, the value at every step is determined based on the value at the previous time step and based on the input at the current step. This property allows the network to store dependency and preserve the ordering of sequences. RNN's are one of the most popularly used neural networks for performing sequence-related predictions and classifications. They find a lot of applications in real-world scenarios. These include speech recognition, machine translation, stock market prediction, etc. For more information on RNN's, we refer the readers to Graves et al. [67].

Long Term Dependency Issue: Even though RNN's are found to be very powerful when it comes to handling sequence data, one issue that exists with traditional RNN's is their ability to handle long-term dependency. Going back to the example of word prediction, presented at the beginning of this subsection, let us assume that we use RNN to make the prediction. Each word in the sentence can be fed as an input to different input neurons in the RNN (refer figure 2.8). What might happen is that as the network takes more and more words as input, the recurrent cell with the last word of

[3]This is just a conceptual view. In reality, there can be one or many outputs and multiple hidden layers based on the problem at hand

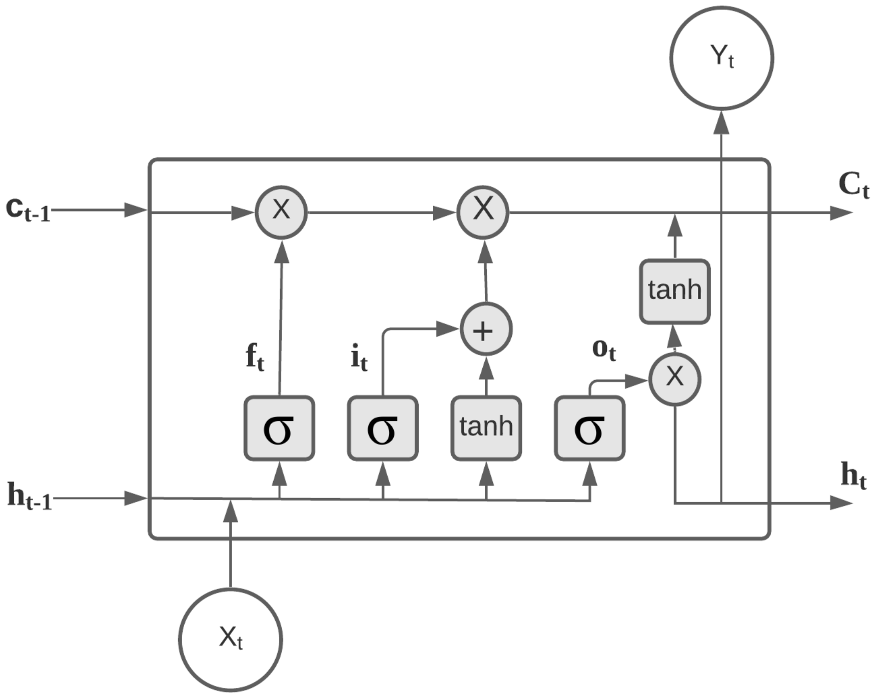


FIGURE 2.9: LSTM Cell

the sentence (”be”) will have more information on the previous words (like ”today also it will”). But the network will have less or zero information on the first words in the sentence (like ”yesterday was a sunny”). This is because as the length of the sequence increases, due to the recurrence formula, the cell will have more information on the near previous states than that of a farther state. This problem is better known in the literature as *Vanishing gradient problem* [68, 69].

2.2.3.2 LSTM : Long Short Term Memory Network

Long Short-term Memory networks were introduced in 1997 by Hochreiter et al. to handle the problem of long-term dependencies which exist in traditional RNN’s [70]. They belong to the class of RNN with additional capabilities. The overall conceptual view of LSTM is similar to what presented in Figure 2.8. However, the main change is that the cells in the LSTM (also known as LSTM cell or memory block) performs a set of complex operations compared to the simple operation performed by a cell in a traditional RNN (refer equation 2.3).

The hidden layer in LSTM contains a set of cells and the input to these cells are controlled by three multiplicative gates namely, *input gate* i_t, *output gate* o_t, and *forget gate* f_t (as denoted in the figure 2.9). These gates determine the flow of information within and between the cells.

The forget gate, f_t, is responsible for determining the duration to which information needs to be stored in the network.

$$f_t = \sigma(W_f * [h_{t-1}, x_t] + b_f), \tag{2.4}$$

where W_f and b_f represent the weights and bias used by the forget gate. The sigmoid function, σ, ensures that the value of f_t stays between 0 and 1. The value, in turn, indicates if a piece of information needs to be stored for a long time or not. Going back to the sentence prediction example, this will mean that words like "yesterday" and "Sunday" stay longer.

The input gate, i_t is responsible for determining how much of the current input (x_t) is important.

$$i_t = \sigma(W_i * [h_{t-1}, x_t] + b_i), \tag{2.5}$$

where W_i and b_i represent the weights and bias used by the input gate. The input gate also uses the sigmoid function, σ as in the case of forget gate for ensuring that the value of i_t stays between 0 and 1 this, in turn, indicates the relevance of the current input. This value is then multiplied with the output of the Cell state, C_t.

$$C_t = tanh(W_c * [h_{t-1}, x_t] + b_c) \tag{2.6}$$

$$C_t = f_t * C_{t-1} + i_t * C_t \tag{2.7}$$

where W_C and b_c represent the weight and the bias used by the cell state. Unlike input and output gate, the cell state uses *tanh* function. The reason being the fact that the output from applying *tanh* can be -1 or 1 allowing the increase or decrease of the values in the cell state. The cell state is then updated based on the values of f_t and i_t. If f_t gives a value closer to 0 then the previously stored information is not given much importance and vice versa. On the other hand, if i_t generates a value closer to 0, the current value of C_t is not considered for updating the cell state and vice-versa. Going back to the word prediction example, the words like "I", "think" will not be given much importance as they are not really relevant in the context.

The output gate, o_t determines how much information (h_t) needs to be sent out to other cells.

$$o_t = \sigma(W_o * [h_{t-1}, x_t] + b_o) \tag{2.8}$$

$$h_t = o_t * tanh * C_t \tag{2.9}$$

where W_o and b_o represent the weight and the bias used by the output state and h_t represents the output produced by the cell at a time t.

The combination of gates inside the LSTM ensures that relevant information is persisted longer in the network and thereby handles long-term dependency problems in a much efficient manner. Most of the RNN's that are used today, particularly in language translation, stock prediction, etc., are LSTM. In our thesis, we use LSTM to forecast the expected QoS and behaviour of a given system for performing proactive adaptation. The generation of forecasts is accomplished by converting the QoS data as sequential data, specifically *time-series data*. Time-series data in general is a type of sequential where the data points are ordered in time [71]. The process of converting QoS into time-series data and further using them for generating forecasts is explained in detail in Chapter 4.

2.2.4 Q-Learning

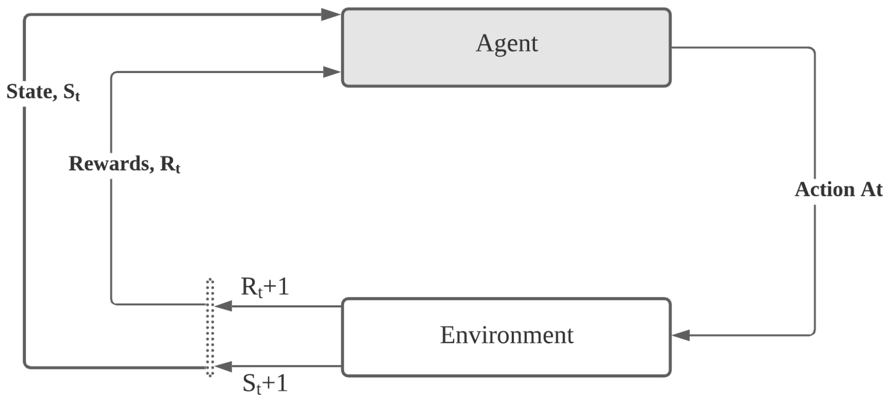


FIGURE 2.10: Reinforcement learning process flow

Quality learning, better known as Q-learning, belongs to the class of reinforcement learning (discussed in Section 2.2.2) [72]. It is also known as a model-free or off-policy reinforcement learning algorithm. This is because it does not require any prior model to perform learning, or rather the algorithm learns by performing actions and gathering feedback for the actions performed.

Figure 2.10 represents the overall framework of a reinforcement learning process. The *Agent* is the central part of any reinforcement learning algorithm. It represents the process/program which needs to learn. The *Environment* represents the world, space, or system with which the agent interacts. As depicted in the figure (fig.2.10), the agent interacts with the environment by performing an action. For every action performed, the agent receives a *Reward*, and it results in the agent moving from one state to another. Hence, the crucial part of the reinforcement learning process is to divide a given problem space into states, action, and rewards.

For example, assume that a robot wants to learn to navigate a given room and reach a particular destination in the room by avoiding obstacles. The robot in this scenario represents an agent. The actions it can take in this scenario will be moving left, right, front, or back. The room in which the robot has to navigate represents the environment. Every time the robot performs an action, it receives a high reward if it has performed the correct action (e.g., an action that resulted in a movement in the right direction, avoiding obstacles, etc.). On the other hand, if the robot performs an action that results in a robot moving in the wrong direction or hitting an obstacle, it is given a low reward (penalty). With this feedback in the form of rewards, the robot learns better to navigate the given space. Further, at each point, it tries to perform the action that maximizes the reward and thereby reaches the destination.

In the case of *Q-learning*, the agent will not have any model of the environment, and it learns by performing actions (There are other types of reinforcement learning techniques known as model-based reinforcement learning techniques where the model of the environment is explicitly available to the agent [61, 73]). Going back to the example above, the robot will not have any information on the shape of the room or location of the obstacles. In Q-learning, the agent achieves the selection of actions through the help of a simple lookup table known as *Q-table.*

Q-table: Q-table forms the heart of q-leaning. It is a simple lookup table, an NXM matrix where N represents the number of states, S, and M represents the number of actions, A. Each value in a q-table corresponds to a (s, a) pair $\forall s \in S$ and $a \in A$. This value is known as *Q-Value.* It denotes the relevance of taking action, a from a state, s.

Selection: As mentioned above, at every point, the agent selects the action that maximizes the reward, or in other words, the agent selects the best possible action that will eventually help the agent to achieve the goal (in the example of robot provided above, this goal will be to reach the destination without colliding with obstacles). The selection of optimal action is made possible with the help of the equation:

$$a' \leftarrow argmax_a Q(s, a) \tag{2.10}$$

where a' represents the new action. This equation represents that every time the agent selects an action from a state, s such that it has the maximum Q value.

Learning Strategy: An agent using Q-learning performs learning and continuously improves the selection through effective use of two strategies:

- **Exploration**: The agent occasionally performs random action as it allows the agent to explore and discover new states that otherwise remain unexplored during the usual selections. This type of exploration is important because it might happen that a particular action would have given a high reward. The agent then keeps selecting this action (even if there are better choices), as it would not have the chance to try any other action.

- **Exploiting**: The agent uses the learned information to select action using the q-table given a state, s. The agent then selects the optimal action from the q-table using the equation presented above.

Q-Function: Every time, the agent performs an action, the Q-table is updated using a value function better known as Q-function given by the formula:

$$Q(s_t, a_t) = (1 - \alpha) * Q(s_t, a_t) + \alpha * (r_t + \gamma * max(Q(s_t + 1, a))) \tag{2.11}$$

where, $0 < \alpha \leq 1$ represents the learning rate. It controls the exploration phase. The higher the learning rate more will be, the exploration. This will allow the agent to see different states. As time progresses, the learning rate can be reduced to move more into the exploiting phase. On the other hand, $0 < \gamma \leq 1$ represents the discount factor, which can be considered as the weight given to the next action, r_t represents the reward for the action chosen at step t.

The agent continuously keeps learning through the exploration and exploitation phases, as explained above, and starts selecting optimal actions as time progresses. Such selections are ensured by the fact that Q-learning converges over a period of time, allowing the agents to select the optimal action given any state [74, 75].

In this thesis, We will be using Q-learning to enable systems to perform adaptations and continuously learn and improve with every adaptation. This part will be further presented in Chapter 5.

2.3 Quantitative Verification

Quantitative Verification (QV) or probabilistic model checking [76, 77], is a set of formal techniques used to verify the correctness of a system that exhibits stochastic behaviour. The correctness is verified based on certain specified properties, which are generally based on the system's non-functional requirements. These techniques makes use of mathematical reasoning to guarantee the correctness. They are exhaustive by design which implies that they check every possible execution trace of the system to perform verification.

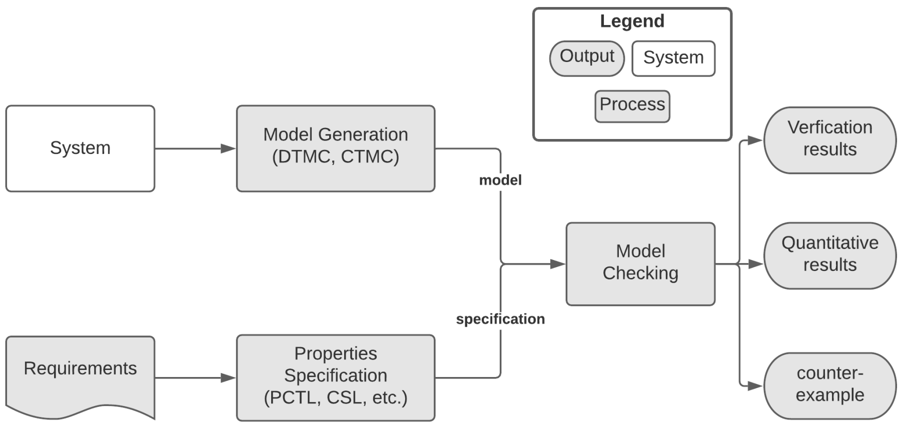


FIGURE 2.11: An overview of the probabilistic model checking process

These technique support the modeling of the systems and analysis of quantitative properties that concern costs/rewards (e.g., resource usage, time) and probabilities (e.g., of an invariant violation, reachability, etc.). Due to QV techniques' verification capabilities, they are widely used mainly in the safety-critical systems to prove the correctness of different components in the system [78, 79].

Figure 2.11 shows the overview of the different process involved during QV or probabilistic model checking. As it can be seen, given a system and a set of non-functional requirements, the first step is to generate a stochastic model of the system based on its behaviour and to generate the formal specification of the properties to be verified. In QV, systems are modeled as state-transition systems augmented with probabilities such as discrete-time Markov chains (DTMC) and continuous-time Markov chains (CTMC).

DTMC: DTMC consists of a set of discrete states where each state represents the different possible configurations of the system. The transition between the states are based on a discrete probability distribution.

Definition 2.1. A labelled Discrete Time Markov Chains (DTMC) is a tuple $\mathcal{C} = (S, s_i, R, L)$, where S is a finite set of states, $s_i \in S$ is the initial state, $R : S \times S \rightarrow \mathbb{R}^+$ is the transition rate matrix, $L : S \rightarrow 2^{AP}$ is a labelling function which assigns to every state $s \in S$ a set $L(s)$ of atomic propositions valid in that state

Each value in R(s,s') denotes the probability of transition from s to s'. Further each transition is assumed to occur at a discrete time step. The specifications of the properties to be verified for the generated DTMC model are written using PCTL (Probabilistic Computation Tree Logic [80]). The DTMC model along with the specification in PCTL can be extended with a reward structure that assigns a reward for every transition made from one state to another as well as for attaining every state. With the help of reward structure, verification of properties like what will be the expected response time of the system after n execution steps, how much energy will be consumed by the system after n execution steps, etc. can be performed. However, real-time systems like IoT, where the sensors acquire and communicate the data continuously cannot be modeled using DTMC as the expression of time is no longer discrete, hence we make use of CTMC for modeling our system in this thesis. This is further described below.

CTMC: As in the case of DTMC, CTMC also consists of discrete states, However, the transition from one state to another is not just based on a discrete probability distribution rather is determined based on a time value, t such that on the lapse of t units, a transition from s to s' is enabled with a probability, λ where the value of λ is based on a negative exponential distribution. CTMC's are widely used for performance and dependability evaluation [81].

Definition 2.2. A labelled Continuous-Time Markov Chain (CTMC) extended with rewards is a tuple $\mathcal{C} = (S, s_i, R, L, \iota)$, where S is a finite set of states, $s_i \in S$ is the initial state, $R : S \times S \to \mathbb{R}^+$ is the transition rate matrix, $L : S \to 2^{AP}$ is a labelling function which assigns to every state $s \in S$ a set $L(s)$ of atomic propositions valid in that state, and $\iota : S \times S \to \mathbb{R}^+$ is a transition reward function that assigns a reward every time a transition occurs in the CTMC.

In the definition above, the transition rate matrix R determines how transitions between states (e.g., capturing message exchanges between nodes in an IoT system) are triggered in a CTMC. Concretely, the probability of a transition being triggered within t time units is equal to $1 - e^{-R(s,s')\cdot t}$ (a transition of rate $1/t$ will take on average t time units to be triggered). Moreover, the reward assignment function can be used to encode rewards and costs, e.g., in an IoT system, the energy consumed by devices every time a message is exchanged between two nodes in the network.

System properties are expressed using some form of probabilistic temporal logic, such as Continuous Stochastic Logic (CSL) [82, 83]. In particular, CSL reward quantification properties can be employed to analyze different QoS properties in an IoT system described as a CTMC. For instance, the class of property $\mathsf{R}^{\mathsf{r}}_{=?}[\mathsf{C}^{<=\mathsf{t}}\ \phi]$ allows quantifying the reward r accrued up to time t across all execution paths, ϕ of the system. An example of a property employing this operator for quantifying energy consumption in an IoT system might be $\mathsf{R}^{\mathsf{energy}}_{=?}[\mathsf{C}^{<=600}]$, meaning "accrued energy over the next 10 minutes (600 seconds) across all system execution paths."

Going back to Figure 2.11, the model of the system generated along with the formal specification of the properties to be verified forms the input to the model-checking process. The model checking process performs an exhaustive verification on the system model by ensuring that the different properties defined in the specification are verified.

As an outcome, model checking process provides three different types of output: i) verification results, which is more like a boolean output which says if the verification of a given property was successful or not. For example: if an IoT system in the next 10 minutes consumes more than 10 joules or not; ii) quantitative results which provides the quantified value of the property to be verified for the given time interval. For example, the expected value of the total energy that shall be consumed by the IoT system over the next 10 minutes; iii) counter-examples, generate traces of the system that will negate the property specified. For example, if the property to be verified was if the system consumes more than 10 joules in the next 10 minutes, then this output will provide the execution paths in which the total energy consumption is less than 10 joules. The topic of quantitative verification is very broad and for further reading we refer the readers to the works in [77, 81, 84].

The concepts presented in this section will be used further in this thesis to verify the quality of adaptation decisions produced by ML algorithms and demonstrate how effective the use of these techniques can further allow the ML to converge towards optimal decisions. This will be presented in detail in Chapter 6.

2.4 Microservice Architectures: An Overview

In this section, we will provide a brief overview of microservice architectures. We then provide details on two specific techniques used in microservice based systems: 1) Service Discovery and 2) Service Mesh. We use this further in Chapter 7 to demonstrate the generalizability of our overall approach.

As defined by Martin Fowler in [85], *Microservice architectural style is an approach to developing a single application as a suite of small services, each running in its process and communicating with lightweight mechanisms, often an HTTP resource API*, accomplished using REST. Microservice based architectures (MSA) have gained tremendous popularity among the industry practitioners and researchers ever since technology giants like Amazon, Netflix, etc. started adopting them [86, 87]. The main reason for such popularity is also due to the out-of-the-box features provided by microservices such as scalability, flexibility, maintainability, agility, etc. crucial for developing any modern software systems.

Traditional enterprise applications are composed of three tiers, i) A client tier, which consists of the HTML interface powered by JavaScript or its variants; ii) database, which consists of the different tables with all the information required by the application, usually accomplished with the help of a relational database management system and iii) a server-side back-end application, which consists of the business logic and is responsible for interacting with the database and accomplishing the functionalities of the application by serving the requests of the client. Such a back-end application is usually implemented using one of the frameworks like J2EE, .NET, etc. by following all the conventions such as separation of concerns through classes, packages, design patterns, etc. This type of an application, in general, is referred to as a *Monolithic application*. However, many challenges exist: i) The entire code is deployed as one piece and making even small changes requires the deployment of the entire application back-end; ii) Strong coupling between the development teams. One team might have to wait for another team to complete a specific feature, which slows down the overall development process; iii) Even though horizontal scalability can be achieved by distributing the back-end across multiple servers, vertical scalability is a big issue as adding new features or components might lead to modifications in different parts of the application back-end; iv) The entire application is tied to one language, which reduces flexibility as adding a new feature that might have native support in other languages is not feasible; v) The complete data is stored on one database, and this usually has strong coupling with the application code, which further reduces flexibility and maintainability, and the list goes on.

The challenges of monolithic application listed above have led to the emergence of the *MSA* where: i) Each service focuses on addressing specific functionality of the overall application; ii) The development teams are no longer tied together as each team can be assigned a particular functionality and can concentrate on developing specific microservices leading to increased business value; iii) Both horizontal and vertical scalability is supported as adding a new feature means adding microservice or just modifying one specific service and not the entire application code; iv) Different microservices can be written in various languages, and the communication between them can be achieved via REST API's; v) There is no centralized coordinator, and every service can have its database (which can, of course, be implemented using different technologies). In fact, in an ideal MSA, no two microservices share the same database [85].

MSA arises from the broader area of Service Oriented Architecture (SOA) [88], and there are several noticeable differences between them. For example, MSA's design is driven by a share-nothing philosophy to support agile methods and promote isolation and autonomy. Instead, SOA adopts a share-as-much-as-you-can philosophy to promote a high degree of reuse. Another significant difference is that MSA mainly focuses on service choreography, while SOA relies on both service orchestration and service choreography [89]. However, the main difference is the dynamicity of microservices because each microservice could be deployed and executed using multiple instances in the same or different servers. The number of locations of running instances could change very frequently. This issue may happen due to scaling up/down, failure of instances, etc.

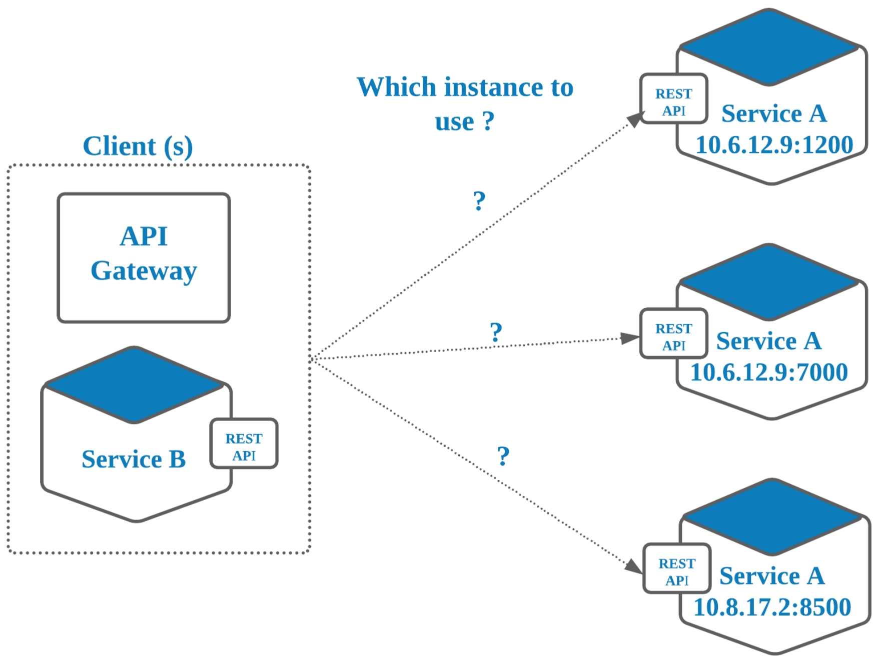


FIGURE 2.12: Why we need service discovery ?

Hence, whenever a client tries to communicate to a specific microservice, it needs to know which instances are available. This is achieved in MSA using a dedicated mechanism known as *Service Discovery*. Even though MSA provides many advantages and is in line with the modern software systems' requirements, there are also a lot of challenges that exist when using MSA [90]. These include: i) handling failures of microservices; ii) integration testing and deployment are challenging as each service release needs to be properly versioned and managed; iii) service instances might be subjected to resource constraints, thereby affecting the overall QoS; iv) Inter-service calls over the network may have a higher cost in terms of network latency and message processing time than in-process calls within a monolithic service process.

In this thesis work, we will be applying the approach proposed to MSA based system to solve two specific challenges: i) To perform effective and efficient service discovery in generic microservice-based systems ii) To develop a self-adaptive architecture to handle the adaptation needs when MSA is applied to IoT. This approach will be further presented in Chapter 7. In this next subsections, we will be elaborating on some background details of two specific MSA mechanisms, which will be later used in Chapter 7.

2.4.1 Service Discovery in Microservices

Service discovery is a mechanism used by MSA, as described above, to enable the clients/application front-end to discover the location of the available instances of the desired microservice. When a client (which can be another microservice itself) needs to invoke a microservice, using a REST API, it needs to know the network location (IP address and port) of an instance of the microservice to be invoked. In a traditional application running on physical hardware, the network locations of service instances are relatively static. For example, your code can read the network locations from a configuration file that is occasionally updated. However, in a modern, cloud-based microservices application, this is a much more difficult problem to solve, as shown in figure 2.12. As we can see, a client (which can be an API gateway, or even a microservice itself) wants to communicate to microservice A. As per the MSA, the client sends the request to an API gateway or a router component (which can be a microservice itself), responsible for routing the request to the concerned microservice. However, as we can see, there are multiple instances of microservice A. The client(s) will not be able to decide, which instance the request needs to be routed to (represented by dotted lines in figure 2.12). This issue arises because every instance, in general, is deployed in a different ports in the same machine or entirely different machines. Moreover, each instance offers a different QoS, and this also needs to be considered. Hence the client requires the support of a mechanism that supports the services to be routed to the respective instances. This is achieved through the process of *Service Disovery.*

There are two main service discovery patterns: *client-side* and *server-side* [91]. When using client-side discovery, the client is responsible for determining the network locations of available service instances and routing requests across them. The client queries a *service registry*, which contains a simple registry of available service instances. The client then uses an algorithm (for example a load-balancing algorithm) to select one of the available service instances and sends a request to it. In the case of a server-side discovery, the client sends a request to a load balancer or a secondary component, which further queries a service registry and routes to one of the available instances based on some criteria (e.g., load balancing). The usage of both patterns has its advantages and disadvantages. For instance, in a client-side pattern, the client has the complete autonomy of selection while there is an overhead that every client needs to be made aware of the services in their registry. On the other hand, server-side discovery abstracts this complexity from the client, but this would mean an additional component needs to be deployed and maintained to handle service discovery.

In Chapter 7, we provide further details on the different challenges in service discovery in MSA and how using a combination of ML techniques (based on the overall idea presented in this thesis) can support effective and efficient service discovery.

2.4.2 Service Mesh

In reality, an industry-grade system based on an MSA may consist of thousands of microservices replicated to thousands of instances. Each microservice may have to communicate with other microservices; few microservices might share a common database, and the communication with the database needs to be handled reliably; at a given point, a given microservice might receive a large number of requests or might face some performance issues, etc. Besides, each communication between the microservices or between microservices and databases must be secured. It must happen through an encrypted channel to satisfy the security requirements. One way to handle this issue is to develop a microservice with all these features of routing, security, etc. However, this would mean that all the microservices should implement similar logic and offer the same functionality regarding security, routing, load balancing etc. Such implementation will result in a massive overhead for the development teams as they would have to focus on handling the infrastructure level issues in addition to focusing on the business logic. Moreover, once microservices are deployed, it is challenging to debug and understand the root cause [90] in case of failures. This is where the role of *service mesh* becomes important as it provides all these features out of the box, thereby allowing the development teams to focus on the business logic.

Service mesh (also known as Sidecar) is a configurable low-latency infrastructure layer that can reside within each microservice. Figure 2.13 shows the image of an MSA based system with service mesh (adapted from [92]). It consists of two parts, *Data Plane* and *Control Plane*. The data plane represents the service mesh section, which resides with the microservices as sidecars and provides the different features. The Control plane, on the other hand, represents the centralized section of the service mesh, which updates the sidecars configurations, collects different types of metrics data obtained from sidecars, etc. It provides a command-line interface (CLI) for the developers/admins to manage the sidecars. Further, it also provides a GUI for managing the application. Some of the key features provided by the service mesh are as follows:

Circuit breaker : Service meshes provide inherent support for *Circuit breaker* pattern [93], which allows identifying and removing unhealthy instances from the main instance pool. Further, it also can add the instance back to the pool after proper guarantees on their status. This service mesh functionality allows the prevention of cascading failures, which otherwise might affect the overall QoS of the system.

Security: The service mesh provides support for authentication and authorization mechanisms for requests made from external clients and microservices within the application back-end. It thereby ensures that only requests that are validated are passed

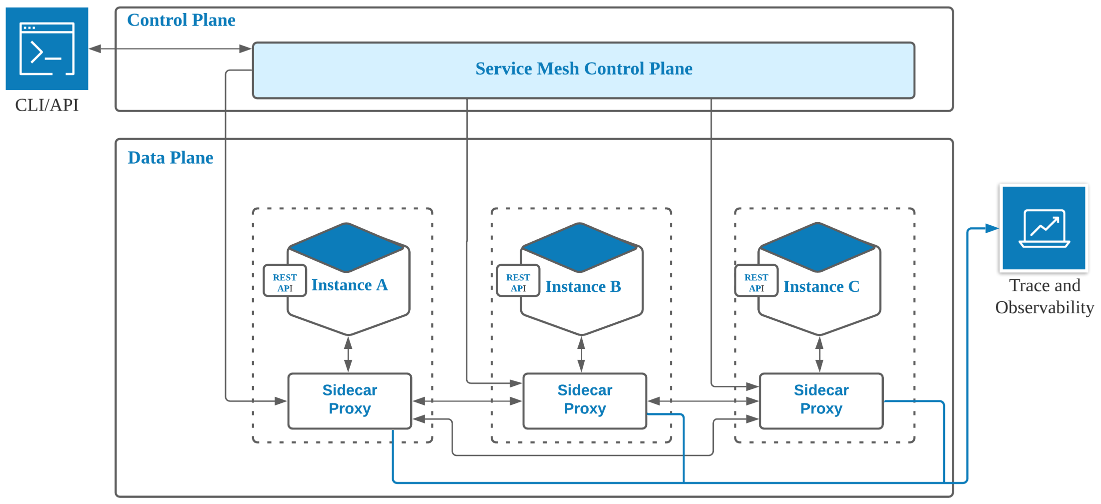


FIGURE 2.13: Overview of Service Mesh

to the instances. Further, to strengthen the security, the mesh also provides mechanisms to encrypt and decrypt all the requests sent to and responses that go out from the service instances.

Traceability and Observability: Since every request and any communication made to/from the instance passes through the service mesh, and it provides a clear trace of the events allowing the developers or admins to troubleshoot in the event of any failure or a security breach. Further, the sidecars can collect the different metrics data of each microservice, such as the response time, network delay, throughput, etc. and this can be further obtained from the control pane. These metrics can be leveraged to improve the overall quality, and they can also be visualized using different tools like Grafana[4].

The usage of a service mesh is recommended as one of the means to perform self-adaptation in MSA [90]. We incorporate service mesh to develop a data-driven self-adaptive architecture for MSA based IoT (MSA-IoT) systems, and this is explained in detail in the second half of Chapter 7.

2.5 Univaq Street Science Case Study: The NdR

In this section, we will be providing details on the case study, which we will be using to evaluate the different parts of our approach in Chapters 4, 5, and 6.

The Univaq Street Science is an event organized in the context of the European Researchers Night (Notte dei Ricercatori (NdR)) in L'Aquila. It is a scientific exhibition

[4]https://grafana.com

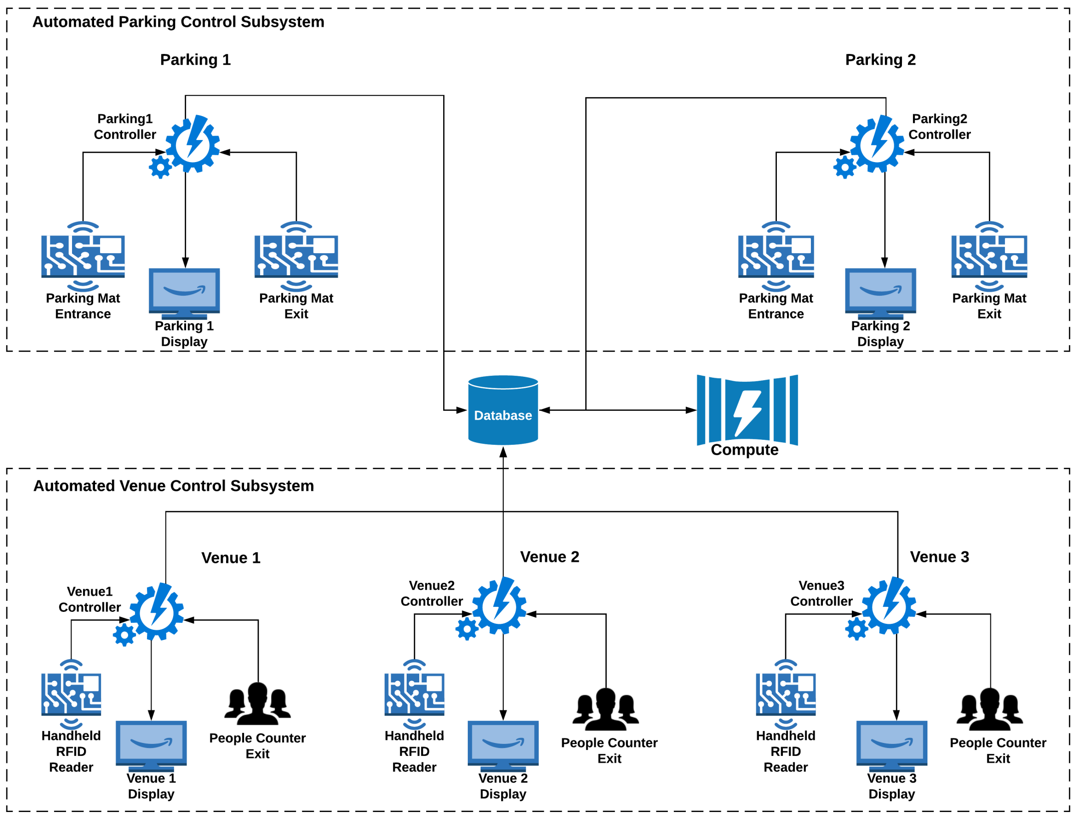


FIGURE 2.14: High-level architecture of the NdR Case study

event organized by the University of L'Aquila [5]. In this event, the research community and public are brought together from the morning until late night to share a combination of entertainment and information. This event takes place throughout the entire city. As an example of a demonstrative scenario, we will take the NdR held in the city center, in which performances, lectures, demonstrations, workshops take place in its squares, main streets, and buildings. From our experience in organizing this event in the city, we captured some sources of evidence: i) About 35,000 visitors are coming to the NdR every year; ii) Late hours usually have more crowds than early hours; iii) The weather (e.g., rain, hot) and its changes influence visitors' preferences in what to see and where to stay more; iv) Visitors cannot quickly locate the availability of parking lots, thus increasing the traffic in the center; v) There are different entertainment events in various venues with limited capacity, and there is no provision for visitors to get the seat availability of venues. Our research group has been invited to provide new services to improve the quality of the visiting experience. Without the loss of generality, in this chapter, we focus on two such IoT services planned for NdR (Figure 2.14 shows the overall architecture of the NdR IoT system). These services are related to parking lot control and venue crowd management.

[5]https://nottedeiricercatoriaq.it/

2.5.1 Automated Parking Control

The city center is the most attractive place. Thus, more crowds are found in this location. However, only two parking lots are available with a capacity of 200 and 150 parking slots, respectively, and these parking lots are created in an ad-hoc way on a day before the event. This means that the visitors need to be notified about the availability in a particular parking lot and redirected accordingly to reduce the traffic congestion in the center. These parking lots are located closer to the main venue, where the most important event takes place.

2.5.2 Automated Venue Entry Management

During the NdR event, there will be some big events in multiple indoor venues in the city center. Each of these venues hosts some entertainment as well as infotainment events. However, there are space limitations in each of these venues, which allows entry to only a certain number of visitors with entrance tickets at a given instant of time. (In this work, we focus on three venues located in the city center with a seating capacity of 500, 200, and 200 seats, respectively). This calls for services that can automatically manage the venue entry by considering the number of people inside the venue and providing real-time updates to the visitors on venue availability.

One of the important constraints we have in the NdR case (Section 2.5) is the lack of availability of external power points in the parking lots or the venue entrances. Hence, we developed an IoT based solution (Figure 2.14) for automated parking control which uses battery-powered parking mats on the entrance and exit points of each of the parking lots. It then provides real-time notifications via a display on the parking lots and sends information to the central database that can be further sent to mobile applications. Similarly, for the automated venue management, we developed a solution that allows the venue manager to verify the entrance tickets using hand-held RFID readers. It also uses people counters at the venue exits to get real-time venue availability. In this way, the users can be redirected to the nearest parking lot based on their venue preference and venue availability.

This chapter described the different techniques underlying this thesis. In the next chapter, we provide a detailed overview of the various works done in self-adaptation related to the work presented in this thesis.

Chapter 3

State of the Art

Self-Adaptive systems, by definition, are systems that can handle different types of uncertainties. There has been an extensive amount of literature work that has been done in the area of self-adaptive systems. An elaborate survey on different approaches to engineer self-adaptive systems was presented by Macías-Escrivá et al. [39] and Krupitzer et al. [31]. Further, a detailed survey on self-adaptation for Cyber Physical Systems (CPS) was presented by Muccini et al. [30].

In this chapter, we present the subset of state of the art in the vast domain of self-adaptive systems related to the overall approach presented in this thesis. Based on this, we first divide the related works done into three different categories based on the parts of the approach presented in Chapters 4, 5, and 6. These categories presented as each subsection in this chapter are as follows: i) Self-adaptation in IoT; ii) Machine learning in Self-adaptation; iii) Quantitative verification and Machine learning in Self-adaptation. Besides, part of the approach presented in Chapters 4, 5 and 6 is applied to perform context-aware service discovery in MSA, and further to develop a self-adaptive architecture for MSA based IoT systems (Chapter 7. Based on this two more categories of related works are presented namely: i) Context-aware service discovery in MSA; ii) Self-adaptation in MSA based IoT.

3.1 Self-Adaptation in IoT

IoT systems are known for the inherent dynamism primarily due to heterogeneity and interoperability that exists [22]. Further, they are subjected to different uncertainties from the environment. Towards this, there have been some works done towards self-adaptation in IoT systems. These are reported below.

Francisco et al. in his *doctoral thesis* [94] came up with a Model-Driven Engineering (MDE) based approach for performing self-adaptation in IoT systems. The work mainly came up with two new contributions: ii) A new middleware to manage software updates in IoT systems. This approach uses the concept of models@run.time for performing adaptation at run-time coupled with the *Kevoree for IoT* framework [1] which is built based on the concepts of Component-based software engineering (CBSE). This approach is unique in the fact that it takes into account the memory constraints on the IoT devices as well as energy consumption. The proposed method was also validated for efficiency using a real use case; ii) Another important contribution was an algorithm named *Calpulli*, which will allow the distribution of components in IoT systems from a central repository in the event where adding/removing of components is necessary. Such a distribution was again accomplished through the use of *models@run.time*. Even though the approach is very efficient and scalable, the approaches have a set of prefixed behaviors/components to replace, and the adaptations are performed reactively. Moreover, the type of adaptation performed does not improve over time.

Another MDE approach for tackling the challenges associated with IoT architecture was demonstrated by *Ciccozzi et al.* in [95]. The proposed approach also uses *models@run.time* method to carry out self-adaptation in the event of any uncertainties. The authors demonstrate the approach's effectiveness using a smart light use case where the controller intelligently switches configurations based on an observable event. Even though the proposed approach was effective in the case study used, the different possible adaptation scenarios during run-time are modeled into the system at design time. This may not hold in a practical scenario as it entirely depends on the potential uncertainties that may arise from the deployment environment. Moreover, as in the previous work, the adaptation performed is reactive in nature with knowledge gained.

Fahed Alkhabbas et al. came up with the idea of *Emergent Configuration (EC)*) for engineering IoT systems [96, 97]. They provide an architecture for EC, which consists of an adaptation manager responsible for performing self-adaptation in the event of any uncertainties in the environment or if there is any change in user goals. They use the MAPE-K loop to achieve self-adaptation, and the adaptation is based on a goal-oriented approach. The trigger to adaptation is a change in user goals or possible future events (expressed by messages). The architectural framework supports both proactive and reactive scenarios based on a set of rules; however, the approach focuses more on the overall framework and does not discuss how proactive adaptation can be performed. Moreover, the performed adaptation itself does not improve or evolve with time, and the adaptation performed is enacted at the application level and not at the level of architecture.

[1] http://kevoree.org

Mirko D'Angelo et al. developed an MDE based framework for self-adaptive CPS [98]. It uses the MAPE-K loop to implement self-adaptation by providing a domain-specific environment (DSE), which allows the users to specify the desired MAPE-K control loop model. It also provides a simulation platform for simulating the designed self-adaptive CPS and produces simulation results which can further be used to improve the system architecture. The approach is more for performing self-adaptation at design-time. As the authors describe the fundamental limitations, the adaptation happens at design-time, the scalability of the framework is not known yet, and performing validations of the adaptations performed requires more investigation.

An approach to performing self-adaptation in IoT systems using the ActivForms [99] run-time environment with support for formal guarantees (provided using statistical model checking) was presented by *Iftikhar et al.* in [100]. The environment supports the deployment and execution of MAPE-K based feedback models to realize specific adaptation goals. Further, the environment supports visualization of the models as well as run-time updating of models. The approach's effectiveness was demonstrated by applying to an IoT system to satisfy energy and packet loss constraints. The adaptation logic used was reactive in nature. Also, as per the evaluations, the approach takes more than 2 minutes for executing an adaptation. The time taken is due to the overhead of run-time model checking. This may not be optimal as the plan is identified and executed post the goal violation.

A QoS-driven approach to self-adaptive critical IoT systems based on a custom DYNAMICO reference model and non-functional properties was proposed by *Gatouillat et al.* in [101]. The approach synthesizes a dynamic controller for adaptation using the Label Transition system. States denote the system's possible failure states. The transition represents the different rules that allow the system to adapt. The proposed approach was further evaluated on an e-health based case study with safety as the critical non-functional properties. The limited evaluation of the approach shows promising results. The approach is reactive in nature, and this is in line with the target domain. However, one key concern is the scalability of the overall approach. Also, generating custom rules for all possible scenarios upfront is a costly process. Also, the adaptations suggested are reactive in nature and not proactive.

D.Weyns et al. proposed an Architecture-based adaptation approach, MARTAS, to manage IoT deployments [102]. The approach uses a combination of run-time models and statistical model checking techniques to accomplish different self-adaptation goals. The proposed approach was developed to reduce the exhaustive verification in state-of-the-art techniques to generate adaptation decisions in run-time. The approach's effectiveness was demonstrated by applying to a real-world IoT system developed for security monitoring with energy consumption and packet loss constraints. Even though the approach was found to be efficient with respect to state-of-the-art exhaustive verification mechanisms, the approach, as others presented above, performs reactive adaptation. The time taken is close to a minute on average due to the overhead of run-time model-checking in reactive setup. Moreover, with every adaptation, the approach does not gather feedback to automatically learn and improve future adaptations' effectiveness.

A framework for architectural self-adaptation in IoT, IAS, was proposed by *M. T. Moghaddam et al.* in [103]. The approach uses Queuing networks to perform adaptations of the architecture. The performed adaptation switches the architecture from one architectural pattern to another based on the QoS constraints. The approach's effectiveness was demonstrated through the simulated version of a real IoT ecosystem developed for a smart grid system with constraints on response time. The proposed approach is reactive in nature, and the use of queuing networks makes it more a design-time approach than a run-time approach. The approach does not improve the selection of patterns based on historical feedback. Moreover, the correctness of such selection is not guaranteed.

What listed above are works that are generic concerning the application of self-adaptation in IoT. There have also been many works that have been done in the field of energy-based adaptation. These works have been reported in the survey by Moghaddam et al. [104]. The survey also states that only a small number of works focus on energy-based adaptation in IoT or CPS in general. In the remainder of this section, we provide some of the works explicitly related to Energy-based self-adaptation in IoT and CPS domains. These works also serve as one of the motivations for the part of the approach presented in Chapter 4.

A generic context-aware adaptive framework for reducing energy consumption in Machine to Machine communication systems was proposed by *cost et al.* [105]. The framework uses a set of operational elements such as inter-arrival time, average packet size, data filter setting, etc. to perform adaptation. The different context information related to the various machine devices is first fed into a Machine To communication Server (MTC), responsible for analyzing the information and suggesting modification in the operational elements to improve energy efficiency. The approach was shown to be effective in different scenarios. However, the system's energy efficiency as a whole can be improved more if it is possible to foresee the energy consumption and perform the necessary adaptation at an early stage to avoid a possible energy violation.

Moreno-Cano et al. came up with a smart environment management architecture supported with IoT capabilities to achieve smart and sustainable environments [106]. The approach mainly uses the user-centric data generated along with the location and context information for performing adaptation. These data are then analyzed using data mining and AI techniques for automatically adjusting the appliance's operational parameters, such as automatically reducing or increasing heating inside a room. The approach was also shown to be effective in a smart building scenario. However, the approach does not consider the energy efficiency from the architectural perspective but rather from the operational perspective.

Wunderlich et al. propose a way to improve the performance in communication among IoT devices and thereby improve energy efficiency using network coding techniques [107]. The approach was evaluated on Big.Little architecture was found to be very effective in maximizing throughput and minimizing the energy consumption, as it allows scheduling of encoding and decoding of data in multiple threads than restricting to use of single threads. Though the performed approach was effective in reducing the energy consumption of the IoT motes, thereby increasing their lifetime, one of the main issues is an added complexity that needs to be handled due to the encoding and decoding. In near-real-time systems, this can have an impact on the performance. Moreover, the encoding and decoding operation, though little, consumes energy. This complexity can be minimized if the behavioral or structural adaptation performed does not necessarily add complexity in the executing system itself (managing system).

Massimiliano Raciti et al. proposed an approach based on anomaly detection to detect security vulnerabilities of a CPS [108]. The approach uses a combination of machine learning and data-mining techniques to detect anomalies in the system. The approach was shown to be effective in various scenarios. However, as the authors mention, the adaptation mechanisms used are rudimentary and require human intervention. Also, the adaptation is applied specifically to the security aspect and not on the general context.

Seiger et al. propose a self-adaptive workflow for CPS using the MAPE-K loop [109]. They extend the concept of BPM processes to manage workflows in CPS. The approach applies the MAPE-K loop to monitor and analyze a given system's execution using sensor data and context data. The correlation of the collected data and the system's execution flow is used to determine the need for adaptation. The approach is further shown to be effective by applying it in a smart home scenario where the overall approach aims to ensure optimal lighting conditions in an energy-efficient way. The adaptation performed is again reactive, and the proposed approach performs adaptation at the application level (changing lighting condition by increasing or decreasing brightness) and not at the architectural level.

Differently from the related work reported above and other self-adaptation approaches in general [31] which are majorly reactive in nature, we have developed an approach (Chapter 4) that identifies the need for adaptation at an early stage by proactively forecasting energy consumption using machine learning models and time-series analysis. This forecast further leverages the decision-making process. The effectiveness and efficiency and efficiency of the approach were evaluated by applying it to the IoT system developed for NdR case study (Section 2.5). Moreover, to the best of our knowledge, we are not aware of any work which extensively discusses with detailed analysis on how the choice of prediction horizons, along with decision periods, can affect the effectiveness and efficiency of adaptations in a proactive setup.

3.2 Machine Learning and Self-adaptation

In this section, we report the works which use machine learning for performing self-adaptation. While this section elaborates only on the works specific to machine learning and self-adaptation, the next section provides details on the works that use a combination of machine learning and quantitative verification for performing self-adaptation.

Dongsun Kim et al. provided a reinforcement learning-based approach [37] in planning dynamic adaptation for architecting self-managed software. The authors categorized planning into two types, 1) off-line planning, where the developers of the software consider all the possible scenarios that the system may encounter and creates a rule-based engine which will aid adaptation 2) online planning where the software automatically learns and understands the relationship between environmental changes and software configurations. The online planning makes use of Q-learning. The effectiveness of the approach is demonstrated with the help of a robotic scenario. The adaptation applied is more at the application level (e.g., enabling the robot to perform better action and not adapting the robot's architecture itself). Moreover, as pointed out by the authors, one of the issues is that the q-leaning may take time to understand the optimal configuration. Hence, an off-line phase is used. This adds an extra layer of complexity. Moreover, since the approach is reactive, it takes time to apply the configuration and get the feedback of the adaptation performed.

Feature oriented self-adaptation framework, FUSION was presented by *Esfahani and Elkhodary et al.* [110, 111]. The approach uses machine learning techniques with support for online learning to select the best adaptation plan in a user-goal violation. The approach uses two processes, sharing a knowledge base in the managing system, namely the learning cycle and adaptation cycle. The former is responsible for determining the effect of adaptations to create better adaptation plans using machine learning, while the latter is responsible for detecting any goal/threshold violations to trigger adaptations. The approach uses features and inter-feature relationships to reduce the configuration space for performing run-time analysis and planning. It supports different algorithms like linear regression, M5 model tree, and regression using SVM [45] to achieve the same. The effectiveness of the approach is demonstrated through a case study. As described by the authors, the approach is reactive in nature. It requires the engineers to define the possible features that can be used for run-time adaptations for various possible uncertainties.

An Adaptation Approach for Self-Adaptive Sensor Networks using predictions was proposed by *Anaya et al.* [112]. They proposed a predictive self-adaptation approach using machine learning and models@run.time techniques for performing architectural re-configurations. The approach is built on the top of MAPE-K loops, where it uses predictive models in the Analyze activity to predict the need for adaptation proactively. The approach was shown to be effective using a case study of forest fires, where the goal is to predict the increase/decrease in temperature and adapt the sensor configurations. It uses a classifier built using a Multi-layer perceptron to predict the event of a threshold violation (we use MLP in Chapter 4 as a baseline). The approach uses the time-series data produced by the sensors (and not QoS itself) for performing the adaptation. Training a classifier is a time-consuming process, and the approach uses a pre-trained model inside the Analyze activity and does not elaborate on how the issues related to concept drift [113] are handled. Moreover, the adaptation process, as such, does not improve with every adaptation performed.

Han Nguyen Ho et al. came up with a model-based reinforcement learning for planning in a self-adaptive software system [114]. The authors point out the fact that many a time, when a software faces an uncertainty, it is very imperative that it has explicit information of the possible consequences that might happen when a particular policy of adaptation is taken. The approach uses a Bayesian-based reinforcement learning where the different possible uncertainties are taken into consideration at design-time and whenever an uncertainty happens at run-time, the model performs an adaptation step, and this will be given a reward. The adaptation is applied if the chosen policy gets the best reward when applied to the model. This adaptation is then applied to the system. The approach was demonstrated with the help of a case study, and it was shown that with the prior knowledge of uncertainties through models, a self-adaptive software system could achieve higher and more stable performance. The proposed approach is reactive in nature, where every adaptation before applying on the system is applied to the model to check the effectiveness. This reactive nature adds to the cost in terms of the time taken to perform the adaptation after a violation is detected and the QoS violation that sustains due to the delay. Moreover, as stated by the authors, there is an added computational complexity of using model-based reinforcement learning.

Pawel Idziak et al. [115]performed an analysis of different decision-making techniques in dynamic self-adaptive systems. They considered a use case of a VM placement problem for performing the analysis. They used three different algorithms for performing decision making for self-adaptation, *Artificial Neural Networks (ANN)* and *Constraint Satisfaction Problem (CSP) based method* and *Q-Learning.* It was observed that the quality of the decision made was good when the training data was good enough in the case of ANN, and Q-Learning performed better for decision-making than ANN. However, it was slower, and CSP performed well in terms of speed and decision quality. However, the proposed approach was reactive. As discussed by the authors, one of the major limitations is that the approach was implemented for a minimal use case where the system's complexity was also very less.

Tao Chen et al. came up with a multi-learner approach for self-adaptive and online QoS Modeling for Cloud-Based Software services [38, 116]. The authors combined multiple machine learning algorithms for adaptively modeling QoS function for cloud-based Software services. They demonstrated the approach using a use-case with a realistic workload. The basic QoS properties such as reliability, throughput, response time, and availability were considered for evaluation, and it was found that using multiple learners to model the QoS function offered better accuracy for different QoS properties. This, in turn, provides the cloud engineers with detailed predictions on the possible QoS values. Also, it can be used for auto-scaling of the cloud dynamically, depending on the QoS predictions. The approach is proactive in nature with a short horizon of 1 time-step and uses a semi-automated adaptation approach.

Moreno et al. employ time-series forecasts combined with probabilistic model checking [34] and stochastic dynamic programming to perform proactive latency-aware adaptations based on a look-ahead horizon considering the uncertainty that may arise from the environment. The approach's effectiveness was demonstrated by applying the mRUBIS case with a utility-based goal to increase revenue and maintain response time within a specified threshold. The approach uses time-series forecasts for predicting the request arrival rate and not the system QoS itself. Moreover, the approach does not use any feedback to improve the adaptation process continuously. In other words, the approach itself does not gather the knowledge gained by performing the adaptation to improve the adaptation process further.

An agent-based framework for performing self-adaptation for IoT application was proposed by *do Nascimento et al* [117]. The framework consists of three layers and supports neural networks, evolutionary algorithms, and finite state machine based controllers to carry out decision making for performing adaptation. The usefulness of the approach is demonstrated through use on two different IoT applications. Identifying different learning algorithms that could be used for self-adaptation is put forward as one of the future research directions. The proposed approach is reactive in nature, where the adaptation is triggered in the event of a goal violation. In this work, the adaptation using machine learning is performed only at the application layer and not at the architectural level.

A self-adaptive mechanism using machine learning and reinforcement learning for adaptation in autonomous systems was proposed by *Piergiuseppe Mallozzi et al.* in an extended abstract [118]. The approach proposes a reinforcement learning mechanism for deciding at run-time such that the safety invariants of the system are preserved. Suppose any decision made by the system violates the invariants. In that case, the monitoring process prevents the decision from being executed, and this, in turn, is given as feedback in terms of a reward function by the monitoring process to the decision making process. This ensures that the system learns and evolves continuously over a while. The concrete realization of the approach and validation of the approach is something that remains open. Moreover, similar to [37, 114], this approach uses reactive reinforcement learning which can be costly due to the exploration phase needed for reinforcement learning (refer Section 2.2.4).

The related works reported above that use machine learning are mainly reactive in nature. Further, in most cases, the adaptation performed does not use feedback to improve the adaptation process. Differently to this, our approach identifies the need for adaptation at an early stage using machine learning models. It uses model-free reinforcement learning techniques (compared to the traditional techniques) to determine the optimal adaptation strategy. Unlike the reinforcement learning-based approaches presented in [37, 114, 118], the proactive nature of the approach ensures that any mistakes during decision making are handled effectively. This is because the approach allows the decision to be modified before the actual occurrence of any threshold/goal violation. The feedback of the strategy selected is used to improve the decision-making process, thereby enabling the architecture to constantly learn with every adaptation performed and conversely execute better adaptations due to the gained knowledge. This is further presented in Chapter 5.

3.3 Quantitative Verification and Machine Learning in Self-adaptation

This section lists the works which combine machine learning and quantitative verification to perform self-adaptation. This field of applying the combination of both techniques to self-adaptation is relatively new. There has been only less research done so far in this area to the best of our knowledge. We report these works below.

An approach that uses a combination of machine learning and statistical model checking for performing effective adaptation was proposed by Van Der *Donckt et al.* [119]. The approach is integrated with the Analysis activity of the MAPE-K loop. It uses machine learning techniques to identify the best set of adaptation options from a larger adaptation space. These options are further verified using statistical model checking, and the best option is selected using a cost-benefit based analysis. The approach's effectiveness is demonstrated by applying on an IoT system where the approach reduced the adaptation space by a good margin compared to traditional techniques. The proposed approach is reactive in nature, where the trigger for adaptation is a violation of any goals or constraints. The approach does use the feedback to improve the model further, but this improvement is not immediate in a sense. The approach still requires some adaptation cycles to train the model. Moreover, this process adds to the time consumed by the approach for generating an adaptation decision.

Jamshidi et al. [120] describe an approach that uses machine learning techniques with quantitative planning based on probabilistic model checking for self-adaptation of autonomous robots. The approach was implemented by extending the Rainbow framework of self adaptation [51]. Machine learning is used in this approach to find a set of Pareto-optimal configurations from a large configuration space. The identified Pareto-optimal set is further verified using probabilistic model checking to determine the best adaptation strategy. Evaluations show the approach's effectiveness in a robotic scenario (adaptation space with million configurations) with a set of goals and constraints. The approach demonstrated significant improvements over the baselines. However, the approach is reactive in nature, where it uses learning to aid quantitative verification and not the other way around. Also, machine learning does not receive feedback to improve its decision-making based on the results of verification.

Quin et al. presented an approach that enhances the traditional MAPE-K loop to support the use of machine learning for efficiently analyzing large adaptation spaces [121]. The approach enhances the MAPE-K loop by adding a learning module that is further used at run-time to reduce the adaptation space consisting of a large set of adaptation configurations to a smaller subset. This is then fed to an analyzer module, which executes the analysis on the reduced subset of configurations. The approach uses statistical model checking to analyze further and verify the configuration subset to select the optimal adaptation plan. Besides, the approach employs online learning to improve the machine learning model in run-time. It is applied to an IoT system to show the overall effectiveness. The approach, similar to other approaches sated above, is reactive in nature. This reactive nature further reduces overall efficiency as the total time to perform prediction, selection, and further verification is high. Moreover, similar to [119], the time taken to obtain the feedback and further learning is not an immediate process.

A recent work that employs a combination of deep learning techniques and statistical model checking for performing self-adaptation was presented by *Van Der Donckt et al.*[122]. The approach proposes an architecture based adaptation using the MAPE-K loop. Similar to [121], this approach extends the MAPE-K loop. However, instead of traditional machine learning, it leverages deep learning techniques to reduce adaptation space with a large set of available configurations. The approach works in two steps, where it first identifies the set of possible adaptation options using a deep learning-based classifier. The approach then uses a deep learning-based regression model to rank the adaptation set in the order of preference. This ranked order of a reduced set of adaptation options is verified by a statistical model checker to identify the first one satisfying the goal. The approach is evaluated by applying to an IoT system with multiple adaptation goals. Even though the approach is shown to reduce adaptation space effectively, one issue similar to other approaches is the approach's fully reactive nature. Also, the use of two deep learning models with statistical model checking in real-time affects efficiency. Although the approach uses feedback to perform online learning, deep learning usage implies that this time will be higher to further improve the models.

Apart from the one presented by Moreno et al. [34], there has also been some work done in proactive adaptation. An approach based on model predictive control principles with analytical models that capture the relation between control parameters and system outputs was proposed by Angelopoulos et al. [35]. In their approach, models are used to predict system behaviour and compose adaptation plans. While Moreno's approach [34, 123] does not use feedback from performed adaptations to continually improve decision-making, Angelopoulos' [35] does. However, it focuses on parameter value tuning as control actions instead of the complex structural changes present in (IoT) architectures.

The final part of this thesis's overall approach uses a combination of model-free reinforcement learning and quantitative verification (using probabilistic model checking) for *proactive decision-making.* Unlike all the existing approaches mentioned above, it employs quantitative verification as a *means to verify the feasibility of decisions produced by reinforcement learning* concerning the system context. It further uses the feedback of the verification to help reinforcement learning achieve faster convergence towards closer-to-optimal decisions. Moreover, since the approach uses model-free reinforcement learning, the time taken to adapt is significantly less. Further, the approach uses two-way feedback obtained from quantitative verification and the continuous proactive QoS forecasts to converge faster towards optimal adaptation decisions. In short, works reported above that combines machine learning and quantitative verification employ machine learning as an aid for reducing the decision space for quantitative verification whereas, we use quantitative verification as an aid for verifying the decision of machine learning. This part of the approach is described in detail in Chapter 6.

So far, we elaborated on different works done in the field of self-adaptation in the IoT domain. We further discussed the works that used machine learning to perform self-adaptation and works that used quantitative verification alongside machine learning to achieve better effectiveness. As described at the beginning of this chapter, the works listed so far form state-of-the art for the different parts of the overall approach presented in Chapters 4,5 and 6. To further show the generalizability of our overall approach, the combination of the approaches presented in particular Chapter 4 and 5 was extended to the domain of microservice architectures in 7. In the following two sections, we will be discussing the works related to the approaches presented in Chapter 7.

3.4 Context Aware Service Discovery in Microservices

Research Works related to the general concept of *discovery* is manifold and ranges from architecture (e.g., centralized, decentralized) to matching mechanisms (e.g., QoS-aware, context-aware, and semantic-aware) and selection criterion (e.g., single objective, multi-objectives). To the best of our knowledge, most of the literature about *service discovery* refers to Service Oriented Architecture (SOA), whereas a little attention has been devoted to service discovery for Microservices Architectures. While a large body of work exists in the context of SOA for each of these categories, we summarize hereafter only those approaches which consider QoS in conjunction with context.

Contextualization refers to the ability to discover, understand and select services of interest deployed within the environment. To this end, a key role is played by ontologies, which have been employed to make service descriptions context-aware [124][125]. These ontologies have been largely exploited in SOA to extend UDDI registry with annotations, and to provide matching relying on service signature matching. Such an approach which exploits signature matching which allows for matching a requested service against the set of advertised services stored in the registry was presented by *Paolucci et al.* [126]. Both requested and advertised services are described as a set of required inputs and provided outputs enriched with context information defined according to ontologies. The proposed approach was found to be well suited for dynamic discovery of web services. One of the main short comings is that the approach does not take into account contextual parameters implicitly and the discovery does not consider the QoS of the web services for matching.

A framework named, EASY for preforming dynamic service discovery of web services in pervasive environment was proposed by *S. Ben Mokhtar et al.* [127]. The matching algorithm implemented by EASY exploits the service signature matching method [126] to rate services according to user preferences on extra-functional properties. It achieves this by providing two specific languages: i) EASY-L which supports semantic description of web services. The context, QoS, functional and non-functional properties can be represented using EASY-L which will be used further during matching; ii) EASY-M allows users to define the different conformance relations and further define the order in which the relations needs to be applied. Both EASY-L and EASY-M are used in service repository to find the matching web service in the event of a service request. The approach did improve the state of the art however, the same approach does not fit well in the context of service discovery in microservice as the QoS keeps changing due to context influences and using QoS declared by the instance might not yield the optimal match.

The SAPERE framework [128] that extends the signature matching algorithm to account for contextual and QoS factors in pervasive networking environments was presented by *Stevenson et al.* [129]. The approach makes use of semantic-match degree as a means to enable communication between the nodes in the network.The approach explicitly considers both application specific as well as contextual factors while performing the matching. The effectiveness of the approach was demonstrated through evaluations on a simulated version of a Smart phone application. Although the approach performs well, in the case of microservices the dynamism and uncertainty are quite high such that prefixed matching algorithms cannot be used as the QoS can have a high variance and the influence of the context also contributes a lot to such variance which makes selection even more difficult.

GoPrime, a fully decentralized middleware for performing self-adaptive service assembly was presented by *Caporuscio et al.* [130]. Every service is encoded with a set of QoS and structural attributes in the form of a utility function. The goal of the middleware is to ensure that at every point, the services are assembled in such a way that the local utility of the involved services as well as the global utility is optimized. It makes use of the signature matching algorithm for considering contextual factors while discovering and assembling services of interest satisfying QoS constraints. In particular, the proposed approach exploits Context-awareness and QoS-awareness for providing a fully decentralized service assembly. The effectiveness of the approach is proven via an e-health case study. Although the approach performs well for service assembly in SOA, as rightly pointed out by the authors, the middleware does not support instances of the same service (rather services of the same type is supported) while performing the assembly. This is crucial when it comes to service discovery in microservices (refer Section 2.4). However, this can be used in combination with service discovery for performing dynamic service composition.

Machine learning techniques in general, have been already employed for developing recommendation systems for service discovery in Web Services, and demonstrated to be effective and efficient. In [131], *Nayak et al.* propose a number of data mining methods to improve service discovery and facilitate the use of Web services.

J. Andersson et al. introduce a framework for optimizing service selection based on consumer experience (i.e., context), and preferences (i.e., utility) [132]. The proposed approach is mainly used to perform service composition in open market scenario where performing optimal composition of services is hard due to the difference in the service quality ensured by the service provider and the service quality experienced by the service consumer. To mitigate this the framework uses machine learning technique to ensure optimal service composition (measured in the form of a utility function) based on context of service consumer. In particular, the framework maintains a set of predefined selection rules that are evolved at run time by means of a reinforcement learning strategy. The proposed framework was proved to be effective in performing optimal service composition by evaluating on a stock market based scenario. Again the focus of this work is on service composition of services while in microservices the discovery targets instances of the same microservice that might appear and disappear during the system lifetime making the uncertainty even higher. This also results in high variability in QoS which also needs to be handled.

A data-driven approach to service discovery in microservices was presented by *Houmani et al.* [133]. The approach uses a hybrid service discovery pattern (combination of client side and server side discovery pattern, refer Section 2.4 for the process of service discovery. The approach requires the clients to define a data model of the required services with information on the requested functionality as well the the expected quality profile. On the server side, each service registering to the service registry will present their data model. These are further used by the discovery framework to allow service consumers to discover the available functionalities and microservices and further identify the best fit by matching the QoS profiles. Further the approach uses custom scaling algorithms and load balancer to manage the overall process. The approach is demonstrated and further evaluated using a testbed. The approach does not consider the continuously changing QoS properties of the instance in microservice which might affect the quality of selection during the process of service discovery. While the approach considers the context information for performing a match, it employs load-balance algorithm to select the instance following a match but it only considers the declared QoS of the instance and not the expected QoS. Moreover selection does not take into account the contextual attributes and the approach does not exploit any online learning to continuously improve the selection process.

An approach that uses decentralized online learning to perform self-adaptive service assembly was proposed by *D'Angelo et al* [134]. The approach builds on the top of [130] with support for online learning and automated load balancing. The approach exploits MAPE-K loop and makes use of reinforcement learning with a two layer learning strategy to generate a service assembly that maximizes the overall QoS. In a decentralized setup, multiple nodes might host one or more services of the same type. The approach uses reinforcement learning in the individual nodes as well as in the selection layer over multiple nodes to ensure that the QoS of the overall selection is maximized. The approach is shown to be highly effective by evaluating on a simulated version of a social sensing application. The approach is reactive in the sense it just considers the momentary QoS and not the expected QoS while making the selection. This aspect plays a crucial role as the QoS especially in the case of microservice system can continuously change and proactive approach might provide more guarantees.

Departing from the aforementioned approaches, we make use of a combination of machine learning techniques for selecting services of interest that fulfill the QoS requirements in a given context. In particular, we rely on machine learning to mitigate the uncertainty and variability emerging from frequent changes in services context and QoS profiles. Specifically, the approach uses deep neural networks to predict the evolution of QoS for every instances. It then uses reinforcement learning techniques based on the context information of the service provider as well as the consumer to perform optimal selections. This is described in detail in Chapter 7 (Section 7.1).

3.5 Self-adaptation in Microservice-Based IoT

In the previous sections, we have seen different works where self-adaptation and machine learning have been used in IoT, service assembly for pervasive systems, etc. In this section, we discuss in detail the works related to self-adaptation in microservice-based IoT (MSA-IoT). Applying MSA to IoT brings in many challenges primarily because adaptation concerns can emerge from different levels of the system, architecture, application, or IoT devices. This is elaborated in Chapter 7. However, to the best of our knowledge, a full-fledged approach, framework, or reference architecture for self-adaptation that combines MSA and IoT does not exist. Hence, in this section, we describe the works done in the field of self-adaptation for microservices. This further serve as the state of the art for the framework presented in Chapter 7 (7.2).

GRU, an approach that handles self-adaptation in MSA-based systems through a decentralized MAPE loop, was presented by *Florio et al.* [135, 136]. The approach uses the concept of multi-agent systems for executing the adaptation where each agent or rather GRU agent, as stated by the authors, is responsible for controlling a set of docker containers and managing their adaptation concerns. The agent also receives information on the QoS metrics of other containers from other agents. The approach uses this while performing adaptation to ensure that the adaptation does not violate the overall QoS goals. The approach is further evaluated on an MSA-based video streaming application. One of the issues with the approach is the use of a decentralized MAPE-K loop. It adds an extra layer of complexity to every microservice [137]. Moreover, as pointed out by the authors, it uses only the reactive adaptation method, and the approach could be better improved if it supports proactive adaptation as well. Moreover, the approach does not provide support for application-level adaptations.

A reference architecture for self-adaptation in MSA based on SOA's standard concepts was proposed by *Baylov et al.* [138]. The architecture at a high-level uses the concept of a decentralized MAPE-K loop, as presented above. While the work described above uses agents for one or more services in a container, in this work, every instance of the service provider consists of two components, namely the managed component and autonomic manager. The former accomplishes the microservice's core functionality while the latter, as the name suggests, handles the adaptation. The approach also uses the concept of adaptation registry to share common adaptation tactics across different microservices. Like what stated above, one of the key issues here is the added complexity, which mandates the developers to ensure that every service is deployed with an adaptation layer. As per a recent study [137], MAPE-K based approaches are not the best suited for self-adaptation in microservices. Further, the close coupling of the adaptation component with microservice means no overall view of the system QoS. Also, it restricts the extensibility of the approach.

Khazaei et al. [139], instead, introduced the idea of using self-adaptation as a service for managing adaptations concerns in microservice-based architectures. The approach introduced four services, namely, security as a service, configuration as a service, healing as a service, and optimization as a service. Each of the services performs the MAPE-K activities and is given specific responsibilities to manage different adaptation concerns. The approach is further demonstrated through an MSA based IoT application. The benefit of the approach lies in the fact that the adaptation is handled by an external service, unlike other approaches. Although this takes away the developer's overhead from implementing the managing system, adding additional services to manage adaptation implies that adaptation concerns of those four services also need to be handled, which mandates the need for another managing system. Moreover, the approach is only reactive and does not add support for adaptation concerns of IoT devices.

To further improve the adaptation process, Kubow, an approach in the form of a tool for performing architecture-based self-adaptation in cloud-native applications, was proposed by *Aderaldo et al.* [140]. The approach leverages the use of the Rainbow framework [51] for performing the adaptation. It uses two different ADL's for defining and specifying the architectural model of the application and defining the specification of the different types of adaptation tactics. The approach makes use of Kubernetes metrics API to collect the metrics of the various deployed services, which are further used to trigger adaptation as per the defined tactics. The tool is demonstrated using a sample cloud-based application. The approach improves the state of the art as it handles the adaptation logic at the infrastructure layer. However, the adaptation performed is still reactive in nature and does not consider the adaptation concerns from the application level.

Further advancing state of the art, *Magableh et al.* presented a self-healing microservice-based architecture for systems based on docker [141]. The approach follows a MAPE-K approach to perform self-healing. It continuously collects the metrics data of the different microservices and uses a neuroscience-based algorithm to detect anomalies relatively or predict any unnatural spike in the metric data. This is further used by the approach to select an adaptation action based on utility scores. The selected decision is also verified based on a consensus algorithm (to ensure that the adaptation does not affect other services). The effectiveness of the approach is evaluated on a prototype of an MSA application in a docker swarm. The approach is very much effective, considering that it performs both proactive and reactive adaptation. The approach, however, does not consider the adaptation concerns of the application level. Further, applying this to the MSA-based IoT system mandates the adaptation concerns because IoT also needs to be handled. Moreover, the decision-making process used by selecting the adaptation strategy does not improve over time to better improve the adaptations.

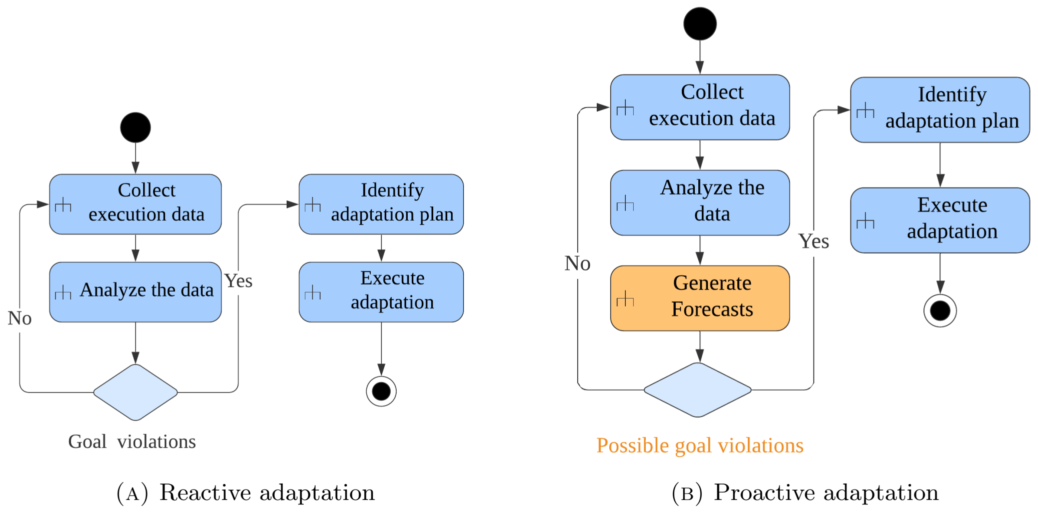
As mentioned at the beginning of this section, to address the different challenges in MSA based IoT domain, we have developed a data-driven self-adaptive architecture for MSA-IoT systems. The architecture considers adaptation concerns that may arise from different levels, such as IoT devices, microservices, and users. It also considers the adaptation challenges that emerge when IoT devices and microservices are used in tandem. Furthermore, besides reactive adaptation, the architecture leverages machine learning techniques to perform proactive adaptation in scenarios where such adaptation guarantees high effectiveness. Further details of the architecture is presented in Chapter 7 (Section 7.2).

Summarizing, in this chapter, we have presented state of the art related to the different parts of the approach presented in this thesis. A brief description of how each part of the approach differs with respect to state of the art has also been provided at the end of every section. In the upcoming chapters, elaborate details on different parts of the approach are provided. As a starting point, in the next chapter, we provide a detailed description accompanied by extensive evaluations on how our approach uses deep neural networks to perform effective and efficient proactive adaptations in IoT systems.

Chapter 4

Leveraging Machine Learning for Proactive Architectural Adaptation

Self-adaptive architectures have been emerging as a promising solution towards managing run-time uncertainties in IoT so to improve the overall QoS [29][32]. However, one of the issues with the existing self-adaptive solutions is that most of them are *reactive*, where the adaptation happens in the event of any uncertainty [31](refer Section 1.2). Figure 4.1(A) shows the generic process flow during such a reactive adaptation setup. As we can see, the process continuously monitors the system execution data and performs adaptation as and when a deviation from the goal occurs.



(A) Reactive adaptation (B) Proactive adaptation

FIGURE 4.1: Process flow of reactive and proactive adaptation techniques

Although this works well, one of the main issues is that *the adaptation happens only when the system has already moved to an undesirable state (deviated from the expected QoS), and this can be costly, particularly in the case of a data-driven or event-driven systems such as IoT*. One way to tackle this issue, as suggested in the literature [142], is to use proactive adaptation techniques that enable the system to foresee uncertainties and adapt accordingly. However, as we have seen in Section 1.2, one of the main issues in using proactive adaptation techniques is that they do not provide the systems the ability to proactively identify the need for adaptation with good accuracy (which can save the system from reaching an undesired state)[31].

Hence, in this chapter, we move the focus from a traditional self-adaptation approach to a machine learning-driven proactive approach, thereby answering the RQ 1 of *How to perform effective and efficient proactive adaptation using machine learning techniques?* Figure 4.1(B) represents the generic process flow of a proactive adaptation mechanism. As we can see, there is an additional step (as opposed to the reactive adaptation process flow) that uses the monitored data to forecast the expected QoS. The adaptation is further carried out based on the forecasts made.

Such a process can be achieved using Machine Learning (ML) techniques as they can leverage the QoS data to provide accurate forecasts of the expected QoS over a given time interval. One of the important characteristics of any QoS data, in general, is that it has a *temporal dimension.* This property can be exploited to convert energy consumption data into time-series data. Hence, *the problem of foreseeing the uncertainty becomes a time-series forecasting problem.* However, four main challenges exist:

- Traditional time-series forecasting are not the best, particularly when it comes to long-term forecasts, due to the high-variability of energy consumption trend in IoT systems;
- The forecasts need to be highly accurate as it might otherwise lead to sub-optimal adaptations
- Learning the patterns in time-series for generating forecasts is a time-consuming task and performing this in real-time is not feasible;
- The choice of the forecasting and adaptation interval may affect the effectiveness of the adaptation.

Towards this direction, in this chapter, we describe the first part of the approach, which performs machine learning-driven proactive adaptation by *extending the traditional MAPE-K* loop to *support the learning time required for generating the forecasts.*

It is further applied to an IoT system for continuously monitoring the QoS data, particularly the energy consumption data, forecasting the expected energy consumption using deep neural nets, and performing the adaptation of the system architecture based on the forecast. The adaptation is executed by reconfiguring the sensor components' data acquisition frequency based on a greedy algorithm. It offers the following *advantages*:

1. Training a deep neural network being a time-consuming activity is only performed at periodic intervals as a batch process and uses pre-trained models for generating forecasts in real-time

2. It ensures that the system never reaches a state of high energy consumption by providing abilities to generate accurate short-term and long-term forecasts.

3. The proactive architectural adaptation ensures that the system becomes more energy efficient as the need for adaptation is detected at a much early stage compared to a reactive counterpart.

4. Since the adaptation is performed at an early stage, the approach also ensures that the number of adaptations performed is much less than that needed for reactive based approaches, thus providing higher efficiency.

4.1 Motivating Scenario: Energy Efficient NdR

One of the important requirements in the NdR case study (refer Section 2.5) is to ensure that the system is *energy efficient* as there is no availability of power points in the parking lots or the venue entrances. The parking mats, hand-held RFID readers, and people counters being battery powered needs to operate in an energy-efficient manner. Even though charging can be performed, it might happen that during peak hours, the charge goes down, and plugging in a new sensor/charging the existing one can be costly for crowd management. Hence, the challenge for us in this scenario is to create:

- Energy-efficient system which can proactively forecast the possible energy outrages
- Automatically adapt the architecture in the best possible manner such that the system can stand longer, to avoid recharging as much as possible.

This energy efficiency problem does not just concern this case study, as energy efficiency has become more of a global concern with ICT expected to consume around 21% of the world's electricity by 2030 [143]. Moreover, different policies are coming up at the international level to reduce the overall energy consumption. This is evident from the fact that the EU union has set a target of 20% overall energy savings by the end of 2020 [144]. Therefore, handling energy consumption is extremely important. A recent systematic literature review focusing on self-adaptation approaches for energy efficiency shows that self-adaptation can be used as an effective solution for improving energy efficiency [104]. However, the survey also points out that only a limited number of approaches focus on self-adaptation to improve the energy efficiency of IoT and cyber-physical systems (CPS).

4.2 MAPE-K with Machine Learning

This section provides details on how our approach uses ML techniques for proactive adaptation of the architecture, thereby reducing the energy consumption of a given IoT system. Figure 4.2 shows an overview of the approach. It is based on the MAPE-K loop [50]. Hence, it primarily consists of two systems namely the *Managed System* and *Managing System*.

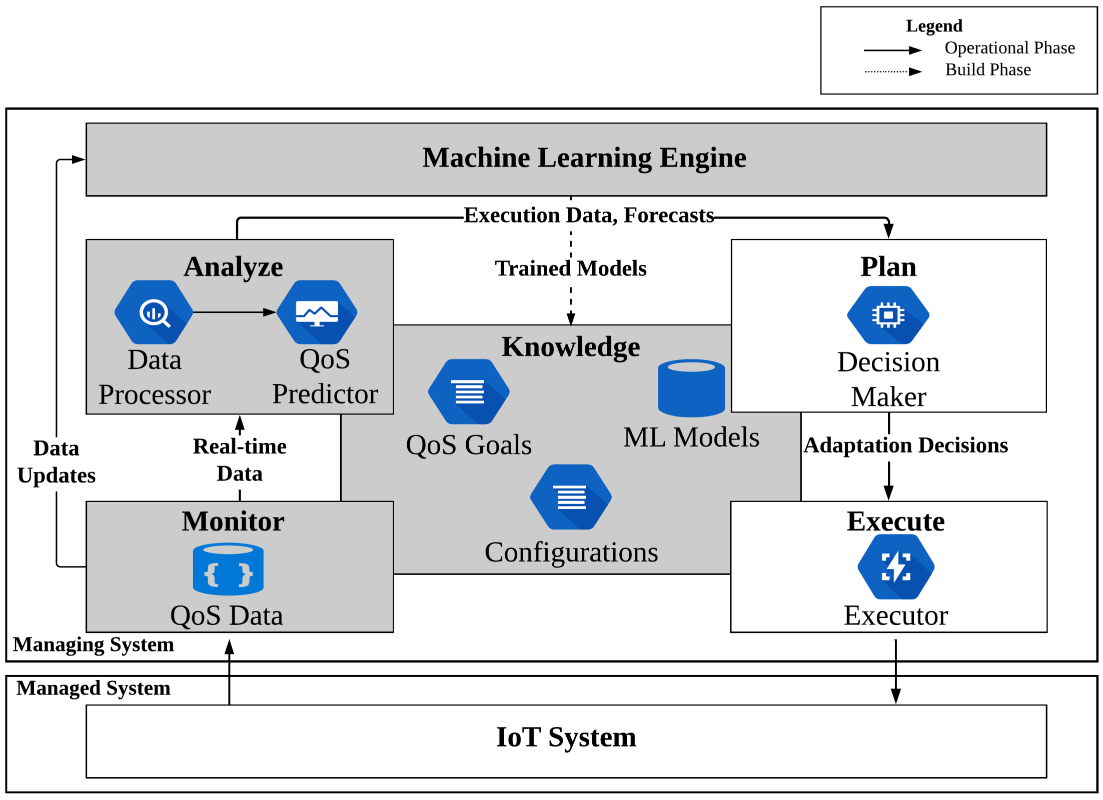


FIGURE 4.2: Approach Overview

The *Managed System* represents the running IoT system, which conforms to underlying software architecture. Designing the software architecture of the IoT system means we also need to consider the underlying hardware configurations of the different components as well as take into account the constraints from the environment in which these components will be deployed. This is important as each of these decisions impacts the functionality of the system and the energy consumption of the system. During the architecture design phase, we define the operational mode of sensor components. This includes modes like *normal mode* (sensor components gather data at standard frequency rate) and *critical mode* (sensor components gather data at a higher frequency rate). We also define the conditions for transitions between the different modes during this phase. For example, we may want a *Parking Mat Entrance* Sensor to gather data every 1 minute (normal mode), and there could be a sudden increase in the arrival rate of cars. Hence, we may need the sensor to gather data every 30 seconds for a higher service accuracy (critical mode). Here the increase in arrival rate acts as the condition for mode change.

The *Managing System* is responsible for performing the activities of MAPE-K. It accomplishes this with the help of a Machine Learning Engine (MLE). Learning being a time-consuming activity, the managing system executes MLE as a batch process at periodic intervals. (represented by dotted lines in the Figure 4.2).

In this chapter, we focus more on the Monitor, Analyze, Knowledge part of the MAPE-K, and how MLE works in conjunction with each of these activities (Marked in grey in the Figure 4.2). While this section introduces the conceptual methodology and is technology-independent, Section 4.3 provides details on the technologies we used for realizing our methodology.

4.2.1 Monitor activity

As in the case of traditional MAPE-K approaches, the monitor activity is responsible for continuously monitoring the IoT system. In our approach, we exploit this activity to monitor and collect the energy consumption data or, in general, QoS data of the IoT system. This data contains information on the energy consumed by different system components at every instant of time. The monitor activity ingests these logs into the *QoS Data* component, and the Analyze activity then uses this for further processing. The process of ingesting the data into the QoS Data keeps continuing throughout the system lifecycle. This means that our approach can be applied to a running system, and this data can be extracted, assuming that the system provides ways to extract the desired QoS data. These data are also sent to the *MLE* for training the ML models.

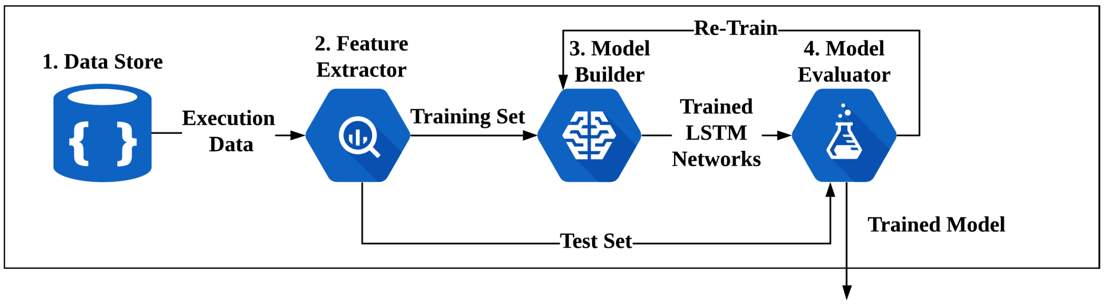


FIGURE 4.3: Machine Learning Engine Pipeline

4.2.2 The Machine learning engine

Figure 4.3 shows the ML pipeline of the MLE. It explodes the component presented in figure 4.2. MLE is the key component of the approach as it is responsible for supporting the *Analysis and Planning activities* of the MAPE-K loop. This component is invoked at periodic intervals to train and build the ML models for predicting the expected energy consumption. In our approach, we refer to this phase of model building/training as the *Build Phase*. The models developed during this phase are updated to the *Knowledge* base of the approach. During real-time execution of the IoT system, these trained ML models are further used to adapt to the architecture proactively. We refer to this process of using the trained models to make predictions in real-time as the *Operational Phase*.

During the operational phase, the Build Phase is run in parallel at regular intervals to ensure that the ML model is up-to-date with new patterns in data, which might otherwise lead to the problem of concept drift [113]. This process of forecasting and adaptation keeps continuing. The use of ML models ensures that the system proactively learns and adapts the architecture, resulting in an energy-efficient IoT system. The functionalities of different components of the MLE are described below.

4.2.2.1 Data Store and Feature Extractor

The periodic updates of QoS data obtained from the *Monitor* activity are ingested into the *Data Store* component (component 1) inside the MLE. This data contains information on the energy consumed by different system components at every instant of time. The intervals of time may not necessarily be uniform. It is based on the fact that whenever there is any interaction within or between the components during the simulation/execution, the energy consumed for corresponding interaction is logged. These logs are then passed to the *Feature Extractor* component for further processing.

The *Feature Extractor* (component 2) is responsible for processing and transforming the raw data into the form as required for the ML algorithm. It internally consists of four different processes [145], which include: i) Time series Modeling, ii) Normalization, iii) Conversion to Supervised learning problem, and the iv) Train and Test data generation.

1. Time Series Modeling: This is the first step in feature extraction. The data stored in the *data store* is structured such that we have the amount of energy consumed for every component at different instants of time. This data has a temporal nature, and we exploit this property to convert the problem of predicting energy consumption into a *time-series forecasting* problem. The energy consumption data with respect to time forms a *continuous time series* [71]. For the ease of analysis, we first convert this into a *discrete time series* by aggregating the data into 1-minute intervals. This process is known as *temporal aggregation*, and it is performed for every component in the system, thereby resulting in a discrete-time series with equal intervals of time.

For example: let us assume that the given system has N components and has been simulated for M units of time. Then, the observation at any instant of time t can be represented by a 2D Vector, $E \in R^{MXN}$ where R denotes the domain of the observed features. The process of temporal aggregation results in formation of a sequence of the form $E_1, E_2, E_3....E_t$. *The problem of forecasting energy consumption is then reduced to predicting the most likely K-length sequence in future given the previous J observations* which include the current one:

$$E_{t+1}, ..., E_{t+k} = \underset{E_{t+1},..E_{t+k}}{argmax}\ P(E_{t+1}, ..., E_{t+k} | E_{t-J+1}, E_{t-J+2}, ..., E_t) \tag{4.1}$$

where P denotes the probability. Since the energy consumed by one component also depends on other components with which it shares interactions, it is a multivariate data, and as the forecasting needs to be done for the next K steps, the problem can be formalized as a multivariate multi-step time-series forecasting problem [71].

2. Normalization: This forms the second step of data transformation. It is one of the important preprocessing steps before applying any ML technique to ensure uniformity in the scale of data. This is especially more important in the case of time series data as the data will have the effects of trend and seasonality [71]. We use *feature scaling* for normalizing data to the range [0,1]. This is performed in the following manner.

Let e_c representing the energy of a component c, Ne_{ct} represent the normalized value of energy for component c at an instant of time t, e_{ct} representing the energy of the component c at time t, e_{cmin} representing the minimum energy consumed by the component, and e_{cmax} representing the maximum energy consumed by the component. Then, the normalized value Ne_{ct} is given by:

$$Ne_{ct} = \frac{e_{ct} - e_{cmin}}{e_{cmax} - e_{cmin}} \tag{4.2}$$

3. Conversion to a supervised learning problem : The previous step creates a normalized multivariate time series data, which can be considered more like a set of multiple columns, each containing observations of energy consumption ordered based on time. However, to facilitate forecasting, *we need to clearly define the input and output patterns*, which can allow the ML algorithms to understand the relationship between the input data and the expected forecast. Towards this, the *Feature Extractor* performs a *shifting* operation on the normalized multivariate time series data to model the series into a primitive supervised learning problem. To formalize this, we use two terms, *forecasting horizon (H)*, which defines the number of steps for which the forecast needs to be performed from an instant of time t, and *lag (l)*, the number of previous time steps that need to be considered to make a forecast from an instant of time t. For example, values of $H = 10$ and $l = 10$ would mean that the approach can predict the energy consumption in the next 10 steps based on the energy consumption in the last 10 steps. Based on the above, the *Feature Extractor* performs the shifting operation as follows:

Let input series, $X = \{e_1, e_2, ..., e_n\}$ represent the energy consumption of a component for n intervals of time. The shifting operation will basically create a new series by advancing the input series, forward by H time steps where $H \geq 1$. For example, a value of $H = 1$ will basically create the output series, $y = \{e_2, e_3, ..., null\}$ and a value of $H = 2$ will generate the series $y = \{e_3, e_4, ..., null\}$. The same process can be applied to recede the series by l steps where $l \geq 1$.

The choice of values for H and l affects how the LSTM model is trained. A lag of 10 minutes implies that LSTM can observe the pattern in the last 10-minute data before making the forecast. This can be made high so as to make better predictions. However, increasing the lag too much can lead to the problem of over-fitting [66]. Similarly doing a forecast for too long intervals may affect accuracy.

4. Train and Test data Generation: The next important part of data preparation is to *divide the data into training and testing sets* for cross-validation and evaluate the accuracy of the generated model. The *Feature Extractor* divides the normalized data into the standard ratio of 7:3 [146] where 70% of data becomes the training set and the rest 30% forms the testing set. It further passes the training set to the *Model Builder*, and the testing set to the *Model Evaluator*.

4.2.2.2 Model Builder and Model Evaluator

The Model Builder (component 3) forms the key component of the MLE as it is responsible for building the forecasting model. As the name suggests, it achieves this by making use of LSTM networks (refer Section 2.2.3.2). LSTM Networks belongs to the class of Recurrent Neural Networks [70]. They have the ability to handle the problem of long-term dependency better known in the literature as the *Vanishing Gradient Problem* [68] [69], as compared to traditional Recurrent Neural Networks (RNN) [16] (refer Section 2.2.3.2). Due to this reason, they have shown to be very effective in time series forecasting [147, 148]. Moreover, existing time-series approaches are not effective when it comes to multi-variate time-series forecasting [149].

Architecture of LSTM:

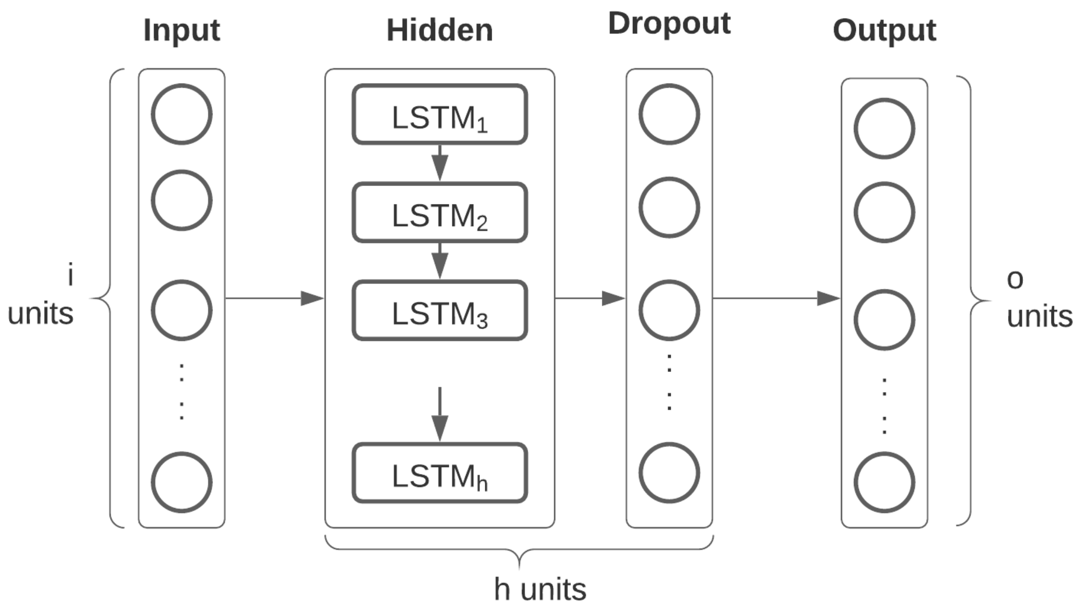


Figure 4.4: Architecture of the LSTM used in our approach

Figure 4.4 shows the overall architecture of the LSTM network used by our approach. It is similar to the traditional LSTM network architecture (refer Section 2.2.3.2) with few modifications to support effective multi-variate time-series forecasting. It consists of four layers, the input layer, hidden, dropout, and output.

The *input layer* consists of neurons (units) to pass the input data to the hidden layers. The number of input units, i is equal to the number of components in the system where, for every neuron, the input will be a vector consisting of l values. In practical terms, this implies that the past l energy observations of every component in the system are passed to the input layer. For example, if the forecasts need to be performed for a system with a number of components, $C = 10$, with lag value, $l = 10$, then the data to the input layer will be a matrix $10X10(CXl)$. This matrix is sent to the hidden layer.

The hidden layer consists of recurrently connected LSTM units (there can be more than one hidden layer in practice. What represented here is just a high-level view). We use an additional dropout layer such that the output from the hidden layer passes through this dropout layer. The number of LSTM units, h in the hidden layer, is decided based on the experiments as this number depends on the lag value, horizon, the number of features, etc. The dropout layer's use randomly drops the effect of the outputs of a few LSTM units in the hidden layer based on a probabilistic value. The number of dropout units in the dropout layer is the same as the number of LSTM units, h in the hidden layer (as every dropout unit determines if the output of a particular LSTM unit needs to be considered or not). The value of dropout is determined using experiments. For example, if dropout is assigned with a value of 0.1 is, this layer will randomly drop the output from 10 percent of LSTM units. The dropout layer is added in neural network architectures to perform regularization and thereby to avoid problems of overfitting [66, 150]. This is especially crucial in time-series forecasting as this problem can easily cause the trained model to produce incorrect forecasts in real scenarios.

Finally, the *output layer* consists of neurons that output the forecasted energy consumption. The number of neurons, o in the output layer determined based on the number of components in the system (C) and horizon, H using the formula:

$$n_o = C * H \tag{4.3}$$

This number is because, since the problem is a multivariate, multi-step time-series forecasting problem, each component's forecast needs to be done for the next H steps, thus contributing to n_o. As explained above, the number of hidden layers and the number of neurons in the hidden layers are determined based on experimentation.

During training, the multivariate time series data generated by the *Feature Extractor* is passed through the input layer to the hidden layer of LSTM units, which is then passed through the memory blocks in each of the units. Each of this memory blocks contains a set of cells and the input to these cells are controlled by three multiplicative gates namely, *input gate* i_t, *output gate* o_t, and *forget gate* f_t (refer Section 2.2.3.2). These gates determine the flow of information between the cells. Standard activation functions, *sigmoid, sigmoid* and *tanh* are used for each of the input, output and forget gates respectively. The training loss is handled using the *Mean Absolute Error (MAE)* loss function, which calculates the absolute error between the predicted value from the output neurons and the expected forecast from the training set. This loss is further propagated back to the LSTM units in the hidden layer. The convergence in training is obtained using the Adam optimization algorithm [65].

The trained LSTM network is further evaluated by the *Model Evaluator* component using the test data generated by the feature extractor.

The *Model Evaluator* (component 4) checks the accuracy of the model on the test data. It performs this using the test data set to forecast the energy values and compare it with the actual values. In the case of lower accuracy, the model evaluator retrains the LSTM network by tuning the network parameters such as modifying the number of hidden layers, training epochs, etc. The final trained LSTM model is ingested to the *Model Repository* component in the *Knowledge* base of the managing system.

4.2.3 Knowledge

Going back to figure 4.2, the *Knowledge* acts as central storage for different types of knowledge required by various components of the managing system for performing the adaptations. It stores three types of information:

(i) *QoS Goals*, consists of a set of energy consumption goals/requirements (or QoS goals in general) that the system needs to meet. The energy requirements define the different energy thresholds by specifying the system's acceptable energy consumption for a given time interval. This includes defining a time interval, t and a threshold set, T consisting of two energy consumption thresholds for the specified interval where, i) Max Energy (M_E), denotes the maximum total energy that can be consumed by the sensors for the given interval, t; ii) Low Energy (L_E), is the total energy limit below which the sensors operate in high-energy efficiency mode, but this might have an impact on the overall service accuracy. These thresholds are defined by the respective stakeholders and stored in the form of a text file.

(ii) *ML Models*, is a repository consisting of the ML models generated by the MLE. As explained in Section 4.2.2, the MLE is executed at periodic intervals to generate ML models. Each such ML model generated by the MLE is versioned and stored in this component. This versioning is performed to ensure the possibility of rollbacks in case of any issue with the model in use. The *Analyze* activity further uses these models for generating energy consumption forecasts during the system's operational phase.

(iii) *Configurations*, consists of the configurations of different components of the IoT system and the configurations of the managing system. These include configurations such as the frequency of data acquisition/transfer for various sensor components and the *decision period*, d. The decision period denotes the interval for generating the forecasts. For this, the approach takes inspiration from Model Predictive Control (MPC) [151], where the horizon is divided into multiple decision periods. The approach performs the forecasts of the expected energy consumption at every decision period for the given horizon, H. These configurations are stored in the form of a configuration file.

4.2.4 Analyze Activity

The *Analyze* activity is responsible for gathering the real-time energy consumption data and forecast the expected energy consumption using the trained model available from the Knowledge activity. During the operational phase, the Analyze activity is responsible for processing the real-time QoS data gathered by the monitoring activity to generate energy consumption forecasts. It consists of two main components:

i) The Data Processor component fetches the value of d specified from the Knowledge activity. It is then responsible for performing the temporal aggregation (as mentioned in Section 4.2.2.1) of the real-time energy consumption data obtained from the *Monitor* activity. This is achieved by continuously aggregating the real-time data into a smaller time series until the number of observations, n, becomes equal to the lag value, l, as required by the ML model. It further sends this processed data to the *Predictor* component at the lapse of every decision period, d.

ii) The QoS Predictor component further passes the processed data available from the data processor component through the latest trained LSTM model available from the ML Models repository to forecast the energy consumption of each of the components in the IoT system for the next H steps. These forecasted values are then passed to the *Decision Maker* component in the *Execute* activity.

For example, if $H = 15\ minutes$, $l = 10\ minutes$ and $d = 5\ minutes$ then the data processor component will keep aggregating the last 10 minute data, and every 5 minute, it will pass this aggregated data to the predictor component which further forecasts the expected energy consumption of the components for the next 15 minutes.

4.2.5 Plan activity

The primary role of the *Plan* activity is to select an adaptation plan based on the forecast generated by the *energy predictor* of the *Analyze* activity. This is accomplished with the help of a *Decision Maker* component. It implements the algorithm to identify the best adaptation plan. In reality, this can be achieved using different methods such as by using model-checking techniques, classical machine learning techniques, reinforcement learning, or deep learning techniques, etc. Since this is not the major focus of this work, we use a greedy approach for identifying the adaptation plan.

Towards this, the decision-maker first classifies the forecast made by the *energy predictor* into one of the three categories: $\{[0, L_E], [L_E, M_E], [M_E, \infty]\}$, where M_E, L_E denotes the thresholds T as defined in the *QoS Goals*. Based on this, the decision-maker generates an adaptation strategy for reconfiguring the data acquisition frequency of the sensor components. One thing to note is when we reduce the frequency of acquisition, it actually implies the increase in the time for which the sensor has to wait before it sends/acquires data. For example, if a sensor S1 is operating at a high frequency of 10 seconds, it means every 10 seconds it is acquiriång some data, now reduction by 10 implies changing the frequency to 20 (10+10).

The complete algorithm of decision-maker is presented in Algorithm 1. It first uses the threshold T as defined in the QoS Goals. It then uses the set of frequencies, F, which specifies the maximum frequency reduction that can be performed for a sensor component when operating on a particular mode under a threshold in T (lines 1-3). EF represent the set of energy forecasts produced by the *Energy predictor* for each of the component in the architecture, N represents the set of sensor components in the architecture, CF consists of the set of sensors along with their data acquisition frequencies, ef, represents the forecasted total energy consumption and n denotes frequency reset interval.

At any given instant, it first finds the sensor component that is expected to consume the maximum energy from the set of forecasts (line 9). It then uses this to calculate the percentage of frequency that needs to be reduced for each sensor component based on a simple greedy approach (lines 13-16). This is done to ensure that the component which is expected to consume the most energy gets the maximum allowable frequency reduction. In this process, the algorithm also identifies the nodes that are running in a critical mode and adds them to a critical node list, CL (lines 17-19). It then identifies the system's current energy state by comparing the forecasted energy value, ef with the threshold values. If ef falls in the category of $[M_E, \infty]$ or $[L_E, M_E]$, then the frequency of the sensor components is reduced based on the corresponding reduction frequency and reduction percentage values (lines 20-38). If ef falls in the category of $[0, L_E]$, then no action is taken.

At any instant, the frequency is reduced, considering the operational modes of the sensors by using different reduction frequency values for each mode based on the thresholds (e.g., fc_{he}, fn_{he}). This is done to ensure that when a sensor is in critical mode, it might happen that the priority needs to be given more for the service accuracy rather than the

Algorithm 1 Decision Maker Algorithm

```
Require: :
 1: Thresholds $T = \{M_E, L_E\}$
 2: Reduction Frequency $F$ =$\{fn_{me}, fn_{le}, fc_{me}, fc_{le}\}$
 3: Current Frequency $CF = \{f_1, f_2, f_3, ....f_n\}$
 4: Set of energy forecasts $EF = \{e_1, e_2, e_3, ...e_n\}$
 5: NodeList $N = \{n_1, n_2, n_3...n_n\}$                    ▷ set of sensor nodes
 6: procedure DECISION-MAKER($T$,$F$,$EF$,$CF$,$ef$,$n$)
 7:     let $l$ represent the length of $N$
 8:     $i \leftarrow 0$
 9:     let $max_C = max(EF)$ ▷ (Forecast of the component with expected to consume
    the maximum energy)
10:     $CL \leftarrow \{\}$                                       ▷ critical node set
11:     $R \leftarrow []$                               ▷ reduction percentage array
12:     while $i < l$ do
13:         if $max_C = EF(N[i])$ then            ▷ E(N[i]), Energy forecast of N[i]
14:             $R[i] \leftarrow 1$
15:         else
16:             $R[i] \leftarrow (max_C - EF(N[i]))/100$
17:         if $N[i]$ in critical mode then
18:             Add $N[i]$ to $CL$
19:             $i \leftarrow i + 1$
20:     if $ef > M_E$ then                                         ▷ ef: $[M_E, \infty]$
21:         $i \leftarrow 0$
22:         while $i < l$ do
23:             if $N[i]$ in $CL$ then
24:                 $CF[i] \leftarrow CF[i] + (fc_{me} * R[i])$
25:             else
26:                 $CF[i] \leftarrow CF[i] + (fn_{me} * R[i])$
27:                 $i \leftarrow i + 1$
28:     if $ef > L_E$ and $ef <= M_E$ then                          ▷ ef: $[L_E, M_E]$
29:         if $ef$ was in $[L_E, M_E]$ for last $n$ minutes then
30:             reset the frequency to original frequency
31:         else
32:             $i \leftarrow 0$
33:             while $i < l$ do
34:                 if $N[i]$ in $CL$ then
35:                     $CF[i] \leftarrow CF[i] + (fc_{le} * R[i])$
36:                 else
37:                     $CF[i] \leftarrow CF[i] + (fn_{le} * R[i])$
38:                     $i \leftarrow i + 1$
39:     if $ef <= L_E$ then                                           ▷ ef: $[0, L_E]$
40:         if $ef$ was in $[0, L_E]$ for last $n$ minutes then
41:             reset the frequency to original frequency
42:         else
43:             Remain in the same frequency
44:     Returns modified $CF$ ▷ The set of frequency reductions for each component in
    the architecture
```

energy but, it might also happen that the frequency needs to be adjusted so that the system will still deliver the best service accuracy along with longer power.

In the case of $ef > L_E$ and $ef <= M_E$, it first checks if the ef value has remained in the same range for last n minutes. If yes, the frequency of all sensor components is reset to the original frequency. This is done to ensure that once the system reaches a low energy state, it needs to be reset to the old configuration else it might happen that the system consumes lesser energy but delivers a poor service accuracy. If no, the frequency of nodes executing critical mode is reduced by a factor of $fc_{le} * R[i]$ and that of nodes executing critical mode is reduced by a factor of $fn_{le} * R[i]$ (lines 28-38).

The same procedure is repeated to check if the value of ef is less than L_E, which means that ef falls in the category of $[0, L_E]$ (lines 39-43). The final frequency values that need to be assigned to each component in the architecture are then sent as adaptation decisions in the form of configurations to the *Executor* component of the *Execute* activity.

4.2.6 Execute activity

The *Execute* activity is responsible for executing the adaptation and thereby reconfiguring the architecture. It achieves this with the help of an *Executor* component. The adaptation decisions produced by the *Plan* activity consists of the details on the data acquisition frequency for each sensor component in the architecture. These decisions are sent in the form of configuration files. The *Executor* component then performs a simple check to ensure that the acquisition frequency for the different sensor components has been correctly specified in the configurations. This is done by ensuring that the configuration file contains a mapping of frequencies for all the sensor components in the architecture and ensures that the configurations are not malformed. For example, the file can have configurations like "S1: 1300, S2: 500, S3: -100...." where S1, S2, and S3 represent the sensor components, and the values represent the frequency. As we can clearly see, the value of -100 cannot be used. Hence, the adaptation manager performs necessary validation tests to check for negative values, null values, etc. and avoids to perform adaptation for that specific component in case of such errors. Once this check is done, it applies the adaptation on the system by updating the existing configurations of the architecture with the configurations provided by the Plan activity. It is then the responsibility of this *Executor* component to dynamically apply these configurations to the architecture.

The MLE, together with the Analyze and Plan activities, ensures that the architecture of the system is adapted proactively by looking for possible energy fluctuations upfront and by reducing the data acquisition frequency in a smart manner. Thereby making the system efficient with respect to the energy consumed.

4.3 Architecture and Implementation

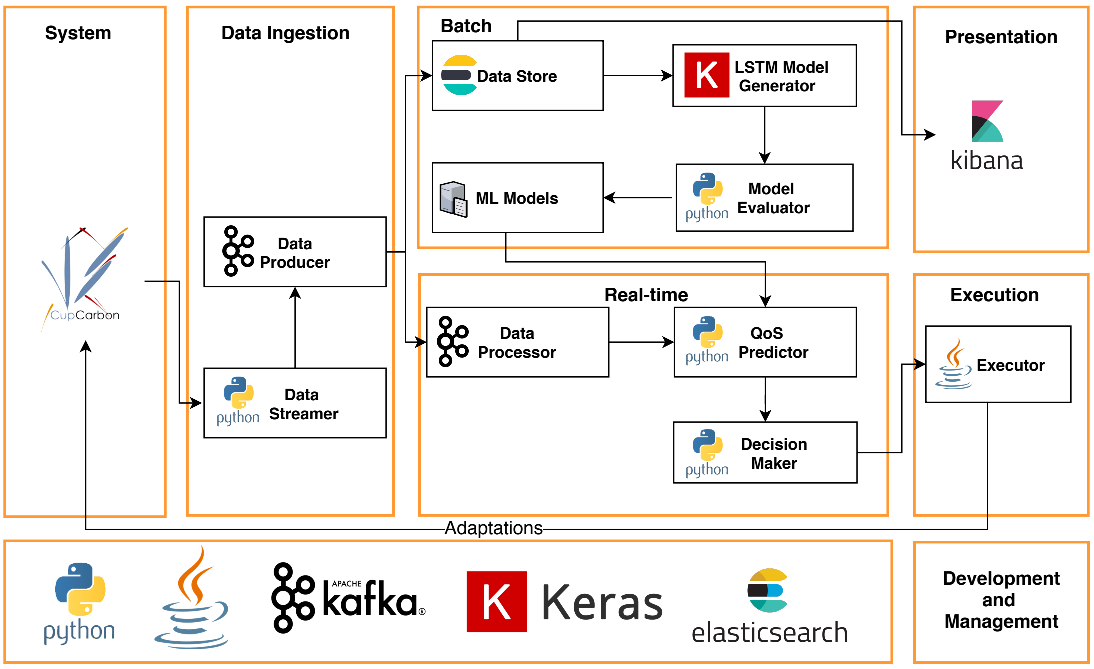


FIGURE 4.5: Data pipeline view of the approach implementation

In this section, we provide an overview of the implementation of the approach and the technology stack used for implementation. We consider the NdR scenario mentioned in Section 4.1 for the explanation of the approach.

Figure 4.5 shows the implementation view of our architecture. It also gives the data pipeline view of the architecture. We use a traditional layered architecture (based on lambda architectural model [152]), with enterprise-grade big data stack for the implementation. We use both java and python for the implementation of the approach. The architecture consists of 6 layers, namely System, Data Ingestion, Batch, Real-time, Execution, and Presentation. While the Data Ingestion layer supports the *Monitor* activity, the *Analyze, Plan, MLE, and Knowledge* is accomplished by the Real-time layer in conjunction with the Batch layer. Finally, the *Execute* part is realized by the execution layer. A detailed description of each of the layers is as follows.

4.3.1 System Layer

This layer consists of the running system. It is responsible for the generation of the simulated/execution data. It uses *CupCarbon*, which allows simulation of any given IoT system.

The first part of the implementation is to model the architecture of the IoT system in CupCarbon [153, 154]. It is regarded as one of the best IoT simulation tools with high practical use especially for energy simulation [155] [156] [157] (for more details refer Section A). The tool allows the creation of wireless sensor nodes with predefined configurations. This results in a total of 22 nodes (as per Figure 2.14 in Section 2.5) which includes 10 sensor components, 5 controller nodes, 5 display nodes, 1 node each for data processing, and database.

We use *senscript*, the scripting language provided by CupCarbon, to implement the sensor logic. As explained in Section 4.2, every sensor component operates in two modes, i) normal mode; ii) critical mode. To achieve this functionality, we use the *delay* feature in Senscript for scheduling the frequency of data acquisition/transfer. Further, CupCarbon supports open-street map-based visualization. This feature is modified to add the map of the city of L'Aquila, and the sensor nodes are placed at different points in the map to emulate the real-setup.

Two sensors, namely, *Parking1 Entrance Mat* and *Parking2 Entrance Mat*, are placed at the entrance of each of the parking lots. They are responsible for counting the cars as they enter the parking lots and send it to the respective controllers (refer to Section 2.5). Further, we use senscript to define the functionality for each of these sensors. This involves defining the logic for the transition between the normal and critical modes. For instance, in parking lot 1, the sensors send data to controllers every 30 seconds in normal mode and 10 seconds in critical mode. If the cars' count in parking lot 1 is more than 20, it is more likely a busy hour, and the sensor moves into a critical mode.

As explained in Section 2.5, we use two parking controllers, one for each of the parking lot. They are responsible for getting the count from the entrance and exit sensors, calculating the parking lots' availability, and sending the information to the display units placed in the parking lots' entrance. They also ingest these data in the database component, which can further be used by the compute component.

For automated venue management, we use the same approach as described above. RFID readers and people counters are placed in the entrance and exits of each of the venues. Following this, the condition to switch between normal and critical mode is defined using senscript. The venue controller component, similar to the parking lot controller, is responsible for collecting the data from the RFID readers at the entrances and the people counters at the exits. This data is then used to provide near real-time statistics on the availability inside the venue. The readers and people counters further sent this data to the concerned display unit.

We then define the hardware configurations for each of the sensor components. These are defined as close as possible to the real system to emulate the real setup. For instance, the hardware for RFID Reader at the venue entrances follows the Wifi standard, and the radio radius is 20 meters (the controller is placed accordingly). The sensor uses Texas Instruments ChipCon 2420RF transceiver for communication, and it uses batteries with up to 19159 joules capacity. Detailed specifications of each of the sensors can be found here [1].

We use .json format for defining the energy requirements as needed for the Knowledge activity. We defined the threshold limits on the energy consumption data. Subsequently, the architecture is simulated using CupCarbon to generate the energy consumption logs. These are .csv files, and it contains the energy consumed by the nodes during idle time, communication, and processing. The other components in the pipeline can further leverage this for forecasts and adaptation.

4.3.2 Data Ingestion Layer

This layer forms the second layer of our architecture. It is responsible for processing the events generated during the simulation of CupCarbon. An event represents the creation of an energy consumption log (or any QoS log in general) for a given instant of time. This layer is responsible for the processing of real-time events in the form of energy logs. This layer acts as an aid to the real-time data monitoring process. It consists of two components: i) Data Streamer, which is responsible for streaming real-time energy logs, and ii) Data Producer, which is implemented using Apache Kafka [158].

For the predictions to happen in near real-time, the data needs to be processed with high throughput and lower latency. This essentially means the data as it is being generated needs to be analyzed and processed in near-real-time. For this purpose, we use Apache Kafka, which is specifically designed for distributed log processing [158]. It has been shown to be very effective in building real-time data ingestion pipelines [159]. Other enterprise-grade tools like Apache Flume could be used for this purpose as well.

The data streamer's role is to stream the energy consumption logs produced by CupCarbon in real-time to the Data Producer. It acts as the bridge between CupCarbon and

[1]https://tinyurl.com/yco2yq4r

Kafka. We developed an off-the-shelf component using Python for achieving the same. Since the logs produced by CupCarbon are .csv files, we created a CSV streamer that streams the log line by line to a topic in Kafka.

The *Data Producer* then streams this real-time energy consumption logs to the Data Processor component in the *Real-time* layer as well ingests them to the *D*ata Store component in the *Batch* layer. In this way, it basically acts as the Data Store of the *Monitor activity* of the managing system as defined in Section 4.2.1.

4.3.3 Batch Layer

This layer forms the third layer of our architecture. As the name suggests, it is responsible for performing batch processing and hence directly maps to the activities of the *MLE*. The *Data store* component (maps to the *Data Store* component of MLE in 4.2 stores the energy consumption data received from the Kafka producer. It is realized using Elasticsearch [160]. Elasticsearch is widely known for its use as a data store for time-series data. Hence, we use it for storing the time-series batch data that is obtained from the Kafka producer. The *LSTM Model Generator* further uses these data for building the forecast model. It combines the functionalities of the *Feature Extractor* and the *LSTM* component depicted in 4.3. We use Keras with Tensorflow [161] backend for implementing the LSTM model [162]. The generated model is further evaluated by the *Model Evaluator* component implemented using Python. On successful evaluation, the model is ingested into the model store component that is implemented as a file system-based storage for storing the forecast models generated. This process is repeated every 4 hours to continuously improve the model.

4.3.4 Real-time Layer

This layer forms the fourth layer of our implementation architecture. This layer's responsibility is to perform energy forecasts and generate dynamic architectural adaptation decisions based on the real-time data received from the *Data Ingestion Layer*. It achieves this by using three components: i) the Data Processor implemented using Apache Kafka [158]; ii) QoS Predictor implemented using Python, and iii) Decision Maker implemented in Python.

The *Data Processor* is an Apache Kafka consumer that is used for the batch processing of real-time energy logs. It is set to consume the message from the topic ”sensor” with a batch interval of 10 seconds. It first applies a simple pre-processing like splitting the comma separated data to generate energy consumption data for every component. It then converts the data to a time-series data and forwarded to the persistence layer for storage. The data is also appended to a multi-dimensional array. Based on the lag value, l, the array is aggregated for 1-minute intervals to generate the past l minutes aggregated data. The aggregated data is sent to the *QoS Predictor* Component at every decision period, d.

The *QoS Predictor* component uses the latest trained models available from the *Model Store* to generate energy forecasts for every component for the horizon, H. This is then passed to the Decision Maker component. The Data Processor component, together with the *QoS Predictor* in the operational phase, accomplishes the *Analyze activity* of the managing system.

The *Decision Maker* component implements the *Plan activity* of the managing system. It uses the forecast of the individual component made by the LSTM model along with thresholds, T, reduction frequencies, F, and current execution frequencies of the components, CF as inputs to the decision-maker algorithm (Algorithm 1) as defined in Section 4.2.5 for dynamically generating the re-configurations for every component in the architecture. The thresholds and reduction frequencies are defined by considering the total energy available in batteries and the operational constraints we have from the use case.

The generated decisions in the form of a .json are then sent to the *Executor* in the *Execution Layer* for executing the adaptation.

4.3.5 Execution Layer

This layer forms the fifth layer of our architecture. This layer is responsible for executing the adaptation. For performing adaptation, we use an off-the-shelf component, *Executor*, which we have developed using Java. It receives adaptation decisions in the form of dynamic architectural re-configurations that needs to be performed from the Decision Maker component of the *Real-time layer*. The role of this component then is to execute the adaptation by communicating the decisions to the *System* layer and dynamically modifying the configurations of the running sensor nodes in the CupCarbon. Thereby, the dynamical adaptation of the running system architecture is achieved. In order to perform this, we modified the source code of CupCarbon. CupCarbon allows users to write custom functions to alter the working of the senscript. We developed a Java program that enables CupCarbon to read the data acquisition frequency from an external configuration file. Every time an adaption needs to be applied, this component ensures that CupCarbon reads the updated configurations as generated by the decision-maker algorithm (Algorithm 1).

4.3.6 Presentation Layer

This layer is responsible for providing visualization of the energy consumption data. This visualization can be leveraged further to improve the configurations of the components in the architecture. We use Kibana [163] for creating a visualization. For further details on the technologies used, we refer the readers to the Appendix A.

4.4 Experimentation and Evaluation

In this section, we describe how we evaluated the approach. First, we describe the data used for the evaluation. Then we evaluate the approach based on its effectiveness and efficiency. Specifically, we evaluate our approach by answering the following research questions:

RQ1.1 How accurate and stable are the energy consumption forecasts made by the approach?

RQ1.2 How much does using this approach save energy as compared to a reactive or non-adaptive approach?

RQ1.3 What is the quality of adaptation performed in terms of the energy consumed by the system with respect to the specified thresholds?

RQ1.4 How efficient is the approach in terms of the number of adaptations performed?

RQ1.5 What is the computation overhead of the overall approach?

4.4.1 Experimentation Setup

For experimentation, We integrated our approach with the NdR system modeled using CupCarbon, and the implementation was done on a High-Performance Computing Cluster consisting of 4 compute nodes. Each of these nodes runs on a Dell R730 CPU with an Intel Xeon Processor comprising 20 cores with CPU 256 Gb of RAM. We used one compute node for running Apache Kafka producer and consumer. Elasticsearch and Kibana were run on the second and third compute node, with the fourth one being used to create and test the ML models. This separation was done so as to mimic the real IoT data pipeline setup. In order to emulate a real-time scenario, we deployed the CupCarbon IoT simulator on a desktop machine running on Intel i5, 2.6-3.2 GHz processor with 16 Gb of RAM. The complete implementation, along with the source code, can be found here [2].

4.4.2 Data Setup

For generating the historical data for the build phase, we simulated the NdR system using the CupCarbon simulator for a period of 30 days. To emulate the case study's real scenario as close as possible, we created a script that generates data for each of the sensor components using intervals of 60 seconds with arrival rates based on a Poisson distribution. The distribution mean values were selected through general observations from the real scenarios of NdR. During the simulation, the sensor data acquisition frequencies were varied randomly. This was done to capture the different types of energy variations that might happen in the real scenario, while the sensors switch between different execution modes. The simulation resulted in an energy consumption log file consisting of the energy consumed by every sensor per second for a period of 2592000 seconds (30 days), amounting to a size of 3.5 Gb. The data were further processed by feature extractor to generate the aggregated one-minute energy data, thus, resulting in a dataset consisting of simulation data for 43200 minutes. This data was then divided into training and testing set in the ratio 7:3, thus, resulting in a training set of 30240 samples and a testing set of 12960 samples. This data was further transformed to supervised learning data as required by the LSTM models, as discussed in Section 4.2.2 for building the forecast models.

4.4.3 Evaluation Candidates

We used the baselines and parameters as recommended by [149] for evaluating the forecast's effectiveness. The evaluation candidates are as follows:

1. *Naive 1*: The basic naive forecasting approach which uses the value at time t for forecast at $t + 1$.

2. *Naive S*: In this approach, the forecasts at time, t is equal to the last known observation of the same seasonal period.

3. *Naive 2*: This approach is similar to naive 1 but applied on a seasonally adjusted data.

4. *SES*: The approach uses Seasonal Exponential Smoothing [164], where the forecasts at time t are calculated based on the equations as expressed in [165].

[2]https://github.com/karthikv1392/PIE-ML

5. *Holt*: The approach uses a traditional holt-winters additive method [166], which is an extension of SES for time series forecasting.

6. *Damped*: This approach is similar to Holt, but the trend component is damped, which allows it to flatten over time [167].

7. *Comb*: This approach uses a simple arithmetic average of the forecasts made by SES, Holt, and Damped for generating forecasts.

8. *ARIMA*: The approach uses traditional ARIMA method [71] implemented with coefficients $p = 2, q = 1, d = 1$ (based on the standard defined in [149]) for forecasting.

9. *MLP*: The approach uses a simple Multi-Layer Perceptron with a single hidden layer implemented using the Keras framework for generating forecasts [168]. Adam optimizer is used for optimization.

10. *LSTM*: The approach we used for forecasting energy consumption.

For further evaluating the effectiveness in terms of energy consumed, the following baselines were used:

1. *No Adap*: Approach without any adaptation.

2. *Reactive*: Approach that performs adaptation using the decision maker algorithm (Algorithm 1) in a reactive manner with thresholds T, $M_E = 1.75$ and $L_E = 1.12$.

3. *Proactive_5*: Approach that performs adaptation using the decision maker algorithm in a proactive manner with a horizon, $H = 5$ minutes, $M_E = 5.89$ and $L_E = 5.2$.

4. *Proactive_10*: Approach that performs adaptation using the decision maker algorithm in a proactive manner with a horizon, $H = 10$ minutes, $M_E = 11.87$ and $L_E = 10.62$.

5. *Proactive_15*:Approach that performs adaptation using the decision maker algorithm in a proactive manner with a horizon, $H = 15$ minutes, $M_E = 17.86$ and $L_E = 16.88$.

6. *Proactive_30*: Approach that performs adaptation using the decision maker algorithm in a proactive manner with a horizon, $H = 30$ minutes, $M_E = 35.9$ and $L_E = 34.36$.

The values for M_E and L_E were selected by analyzing the maximum, median, and minimum total energy consumed by the sensors every H minutes (except for Reactive, where this calculation is done based on the total energy consumed *every minute*) in a regular setup without using any adaptation techniques. M_E was set equal to the median, whereas L_E is set to the average of median and minimum.

4.4.4 Evaluation Metrics

To measure the effectiveness of the forecasts we use the following metrics which are considered as the standard metrics for evaluating forecasts in [149]:

1. *RMSE*: The Root Mean Square Error value for a dataset with n samples is given by the formula :

$$RMSE = \sqrt{\frac{1}{n}\sum_{n}^{i=1}\left(p_i - y_i\right)^2} \tag{4.4}$$

Where p_i represents the predicted value and y_i represents the actual value. It is a good estimate of the deviation of the predicted value with respect to the actual value. The larger the value, the higher, are the prediction outliers.

2. *sMAPE*: The Symmetric Mean Absolute Percentage Error for a dataset with n samples is given by the formula:

$$sMAPE = \frac{100\%}{n}\sum_{i=1}^{n}\frac{|p_i - y_i|}{|y_i| + |p_i|} \tag{4.5}$$

It gives an estimate on the overall percentage of error in the forecasts.

3. *MASE*: Mean Absolute Scaled Error for a dataset with n samples is given by the formula:

$$MASE = \frac{1/H\sum_n |y_i| - |f_i|}{\frac{1}{n-1}\sum_{i=1}^{n}|y_i - y_{i-1}|} \tag{4.6}$$

It penalizes errors in large forecasts and small forecasts equally and is not dependant on the scale of the data. It also gives a measure of how better is the forecast in comparison with simple naive method.

4.4.5 Results

RQ1.1. How accurate and stable are the energy consumption forecasts made by the approach ?

We evaluated the forecast accuracy of our approach using the ten different evaluation candidates, as explained above. For evaluating statistical approaches except for ARIMA (1 to 7 of Section 4.4.3), the *forecastX* [169] package of Python was used and for evaluating ARIMA, the *statsmodels* [170] package was used.

To evaluate the accuracy of prediction, a short-term forecast horizon, H of 5 minutes was used. (This was selected as most approaches may produce reasonable accuracy with short-term forecasts. Hence, it's important to test if our approach using LSTM can

Approach	**RMSE**	**sMAPE**	**MASE**
Naive1	0.95	11.73	1
Naive2	0.83	10.16	0.4
NaiveS	0.75	9.24	0.37
Holt-Winters	1.61	20.06	0.79
Damped	1.13	13.82	0.55
SES	0.82	9.88	0.39
Comb	1.13	13.9	0.55
ARIMA	0.99	12.04	0.48
MLP	5.41	138.24	4.72
Our Approach (LSTM)	0.27	1.57	0.17

Table 4.1: Forecast accuracy of different approaches for a horizon value of 5 minutes

provide higher accuracy than state-of-the-art approaches). Each of the approaches was then used to generate a 5-minute forecast for every observation, o on the testing set by considering the past observations. For building the LSTM networks with Keras, a lag value of l of 5 minutes was used. Further, we used a batch size of 31. This value was obtained after experimenting with different batch size values to understand the best fit. Moreover, keeping the batch size low enables the algorithm to more effectively understand the relationship among the input dataset. We used the standard adam optimizer offered by Keras for performing optimization, as explained in Section 4.2.2.2. The model was built with 110 neurons (22 components * lag/horizon of 5 minutes) in the input and output layers (as defined in Section 4.2.2) and 1 hidden layer with 294 neurons. The model was fit in 120 iterations. Detailed experiments with different values for iterations, hidden layers, and batch size can be found here [3].

Each of the approaches generated the forecast for the total energy consumed by the sensor components in the system for a period of 5 minutes to form the forecast vector, F_o for each of the approaches. However, LSTM model being trained on multi-variate data generated energy forecasts for each of the n components of the architecture. This was then summed up to generate the total energy forecast of the system, F_o based on the o^{th} observation as:

$$TotalEnergyForecast, F_o = \sum_{i=1}^{n}\sum_{j=1}^{H} fe_{ij} \tag{4.7}$$

where, fe_{ij} represent the energy of the i^{th} component in the j^{th} minute forecast. This was performed for every observation in the testing set to generate a forecast vector, F.

For computing the actual energy vector, A for each of the approaches, we followed the same process, but instead of using forecasts provided by the approaches, we computed

[3]https://tinyurl.com/ya9ttl4q

Approach	**H = 10 minutes**			**H = 15 minutes**			**H = 30 minutes**		
	RMSE	**sMAPE**	**MASE**	**RMSE**	**sMAPE**	**MASE**	**RMSE**	**sMAPE**	**MASE**
Naive 1	2.04	12.59	1	3.14	12.89	1	6.47	13.28	0.99
Naive 2	1.39	8.51	0.68	1.95	7.9	0.61	3.76	7.46	0.56
Naive S	1.18	7.13	0.57	1.51	6.04	0.47	2.48	4.52	0.33
Holt-Winters	3.99	25.3	1.96	7.27	31.55	2.34	22.92	57.06	3.58
Damped	2.09	12.8	1.02	3.01	12.33	0.96	5.86	12.02	0.91
SES	1.4	8.5	0.67	1.94	7.86	0.61	3.58	7.23	0.55
Comb	2.36	14.56	1.15	3.85	15.88	1.24	10.26	21.53	1.6
ARIMA	1.81	11.06	0.88	2.64	10.73	1.28	5.05	10.28	0.78
MLP	12.48	180.19	7.59	15.74	132.43	13.77	33.03	143.83	6.4
LSTM	0.67	2.0	0.32	0.7	1.3	0.19	0.81	0.61	0.19

TABLE 4.2: Forecast stability of approaches across different horizon values

the actual total energy of the system after 5 minutes based on the current observation,O

This was done as follows :

$$TotalEnergyActual, A_o = \sum_{i=1}^{n}\sum_{j=1}^{H} e_{ij} \tag{4.8}$$

where, e_{ij} represent the actual energy consumed by the i^{th} component in the j^{th} minute. We then used these to compute the different evaluation metrics. Table 4.1 shows the results of evaluation among different approaches. We can clearly see that the use of LSTM offers the best forecast accuracy in terms of the three measures. It has the lowest values for RMSE, sMAPE, and MASE. For instance, the RMSE offered by LSTM (0.27) is about 69% and 73% better than one offered by SES and ARIMA. sMAPE is evaluated out of 200%, and we can see that sMAPE of LSTM is just 1.57%, thereby producing the least percentage of errors in forecasts. The MASE score further denotes that LSTM produces 93% better forecasts compared to the simple naive method.

Figure 4.6 shows the plot between the actual vector (red) and the forecast vector (black) for 100 samples (for ease of representation). We can clearly see that the prediction made by the LSTM model is able to clearly follow the curve of actual energy. There are periodic intervals where the actual values show a trend to go towards 8.5, where the LSTM also goes down. This is due to the fact that LSTM being a deep neural net, has the ability to identify the non-linear relationships for energy consumption that may exist between the different components of the system. Further, being a class of recurrent neural nets, it also posses the ability to memorize and reproduce this information. Hence, unlike other standard methods, it does not depend only on the seasonality, trend, or any other statistical information of the observed time-series.

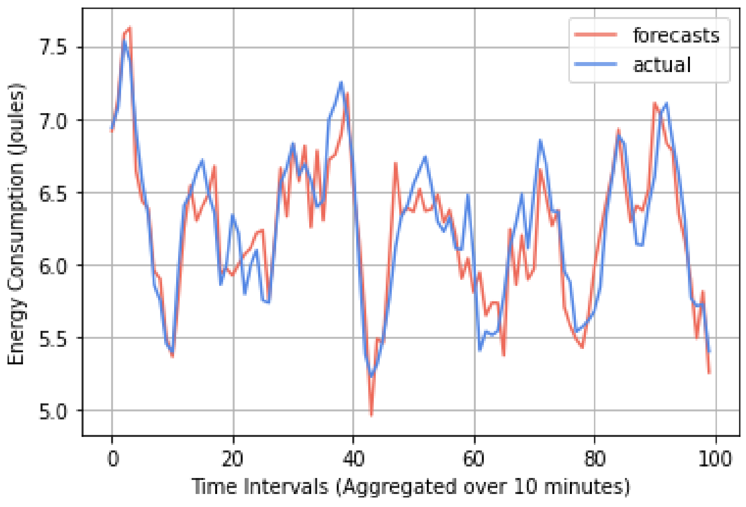
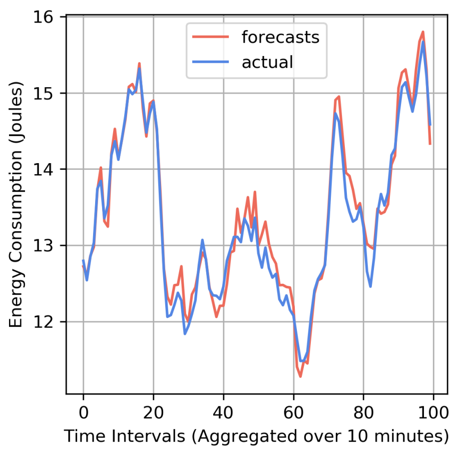


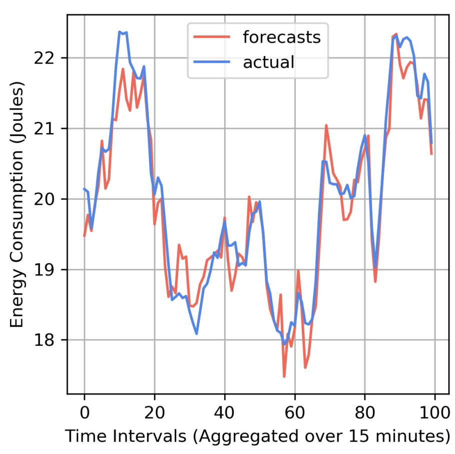
FIGURE 4.6: LSTM Forecasts on historical dataset

One of the key attributes of a good proactive approach is that it should have the ability to provide accurate forecasts for different time horizons. Since a given system/environment might demand forecast of the different horizons, this leads to the test of forecast stability among the approaches. To measure the forecast stability offered by the approaches, we used them to generate forecasts for three different forecast horizons: i) a short-term horizon of *10 minutes*; ii) a medium-term horizon of *15 minutes*; iii) long-term horizon of *30 minutes*. Then we repeated the same procedure as above to measure the evaluation metrics for each of the forecast horizon[4]. Table 4.2 shows the values obtained during the evaluation of stability. LSTM can produce long-term forecasts, the forecast accuracy of LSTM is nearly consistent across varying time horizons. It is also notable that the RMSE values of other approaches such as ARIMA, SES, NaiveS decrease by a large margin with the increase in time horizon. For instance, the RMSE offered by ARIMA for the long-term forecast (5.05) is 174% and 35% higher with respect to the RMSE offered for the short-term forecast (1.81). However, in the case of LSTM, RSME offered for the long-term forecast (0.81) is just 18% and 5% higher than the RMSE for the short-term forecast (0.67). The values of other evaluation matrices for LSTM are also the least and the most consistent across the time horizons.

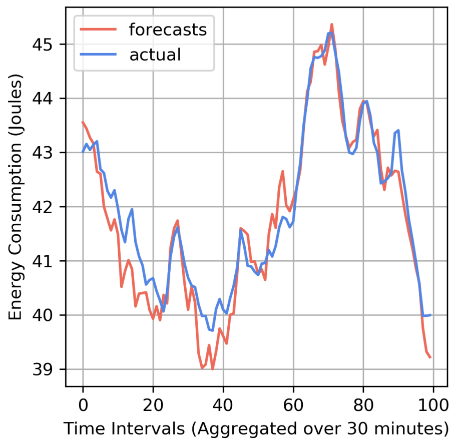
Figure 4.7 shows the RMSE plot of actual energy vs. forecasted energy consumption for each of the chosen horizons. It demonstrates the ability of LSTM networks to provide accurate forecasts across varying horizons. This is mainly due to LSTM networks' ability to learn long-term dependencies, unlike traditional statistical methods.



(A) LSTM forecasts with horizon 10



(B) LSTM forecasts with horizon 15



(C) LSTM forecasts with horizon 30

FIGURE 4.7: LSTM Energy Consumption Forecasts for different horizon intervals

RQ1.2. How much does using our approach save energy as compared to a reactive or non-adaptive approach ?

We measured the energy efficiency by simulating the system using each of the six baselines mentioned in Section 4.4.3. The simulation was carried out for a period of 1 day (1440 minutes) so as to emulate the real scenario of NdR (NdR is a one day event, as presented in Section 2.5). Every sensor components were given an initial energy of 19160.0 Joules which is the default setting of CupCarbon. Further, to evaluate the role of different decision period intervals, d while using the proactive approaches, we used three different decision period intervals for each of the proactive approaches namely, *short* (s_d), *medium* (m_d) *and large* (l_d). While short denotes, a decision interval very close to the time horizon (s_d = 1, 3, 5 and 5 for H = 5, 10, 15, and 30 respectively), medium denotes a middle value in between s_d and H (m_d = 3, 5, 10 and 15 for H =

Approach	**Energy Consumed (Joules)**		
	s_d	m_d	l_d
NoAdap	1712.42	-	-
Reactive	1557.67	-	-
Proactive_5	1504.16	1497.62	1451.43
Proactive_10	1479.44	1462.71	1372.18
Proactive_15	1492.06	1478.03	1394.69
Proactive_30	1553.34	1505.88	1406.75

TABLE 4.3: Total Energy Consumed by the approaches while using different decision periods

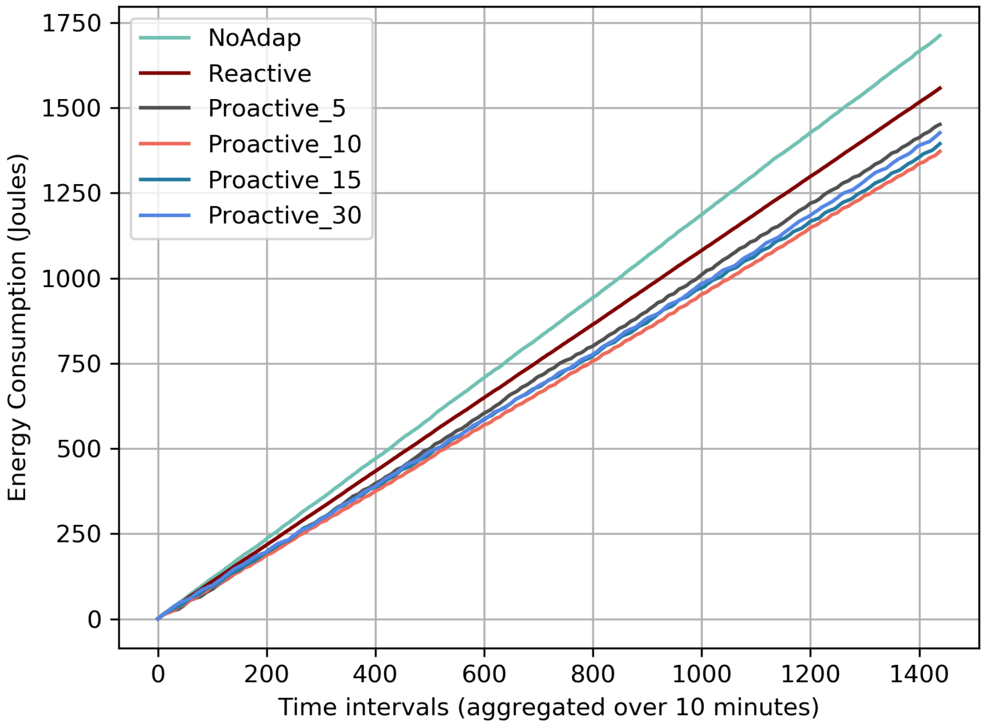


FIGURE 4.8: Cumulative Energy Consumed by different approaches with decision period, l_d

5, 10, 15, and 30 respectively) and large denotes a decision interval $l_d = H$ (m_d = 5, 10, 15 and 30 for H = 5, 10, 15, and 30 respectively). The results obtained during the system's simulation while using each of the approach (including the ones obtained using the three different decision periods) are presented in Table 4.3.

As we can observe from the table, the proactive approach performs better than the reactive approach. However, the time horizon and decision period used have a big impact on the total energy consumption of the sensors while using each of the proactive approaches. In particular, the energy consumed by *Proactive_5* and *Proactive_30* using decision period, s_d is very close to the one consumed by *Reactive approach.* This is because, when using a short decision period, the LSTM networks don't get enough observations to understand the impact of the decision made at the previous decision period, hence ending up performing sub-optimal adaptations. However, as we increase the decision periods, we can see that the energy saved also increases. The cumulative plot of energy consumed by the system while using each of the different approaches is represented in Figure 4.8. We can also observe that as time progresses, the gap between the approaches starts increasing. Towards the end of 24 hours, the least energy consumption is offered by *Proactive_10* with a long decision period, l_d (1372.18 Joules), which is almost 20% and 13% less than the one offered by *NoAdap* and *Reactive* respectively. Moreover it is clear from the Table 4.3 and Figure 4.8, the energy consumed by Proactive_15 and Proactive_30 (to some extent) with decision period l_d is very close to the one offered by Proactive_10 with decision period l_d. This is because there is a great change in the energy consumption rate due to operational modes changes. This implies that it is important to consider a prediction horizon with a decision period that allows the proactive approach in analyzing the impact of the adaptation made during the past decision period. This further enables the approach to make optimal adaptations during the given decision period. Hence, long forecast horizons with large decision periods or even short decision periods do not offer the best results. However, suppose this was a system where the rate of energy consumption change (or any QoS in general) is not high. In that case, even the short-term horizon with short/medium decision periods may produce optimal adaptations.

RQ1.3. What is the quality of adaptation performed in terms of the energy consumed by the system with respect to the specified thresholds ?

The objective of this question is to understand how effective is the adaptation performed by the approach in-terms of the total energy consumed by the sensors per time interval with respect to the threshold, M_E. To measure this, we did the following:

First, we calculated the total energy consumed by the sensors per minute using each approach. We analyzed the effectiveness of approaches to keep the energy consumption under the threshold, $M_E = 1.75$ per minute (The max energy threshold for one minute as defined for *Reactive* in Section 4.4.3). Figure 4.9 shows the box plot of the total energy consumed by the different approaches per minute. pro_h5, pro_h10, pro_h15 and pro_h30 denotes the proactive approaches with decision periods, l_d and horizons 5, 10, 15 and 30 respectively. The average energy consumed by pro_10 per minute is around 0.95, which is about 14% percent better than its reactive counterpart. There are some instances where the energy consumed by pro_10 goes as high as approximately 1.80, which is higher than the threshold hp for one minute (as defined for the reactive baseline). This is true for other proactive approaches, as well. However, as we can observe from the plot, most of the energy values in proactive approaches lie in the lower range of $[0.75, 1.20]$ with pro_h5 consuming the most pro_h10 consuming the least on average.

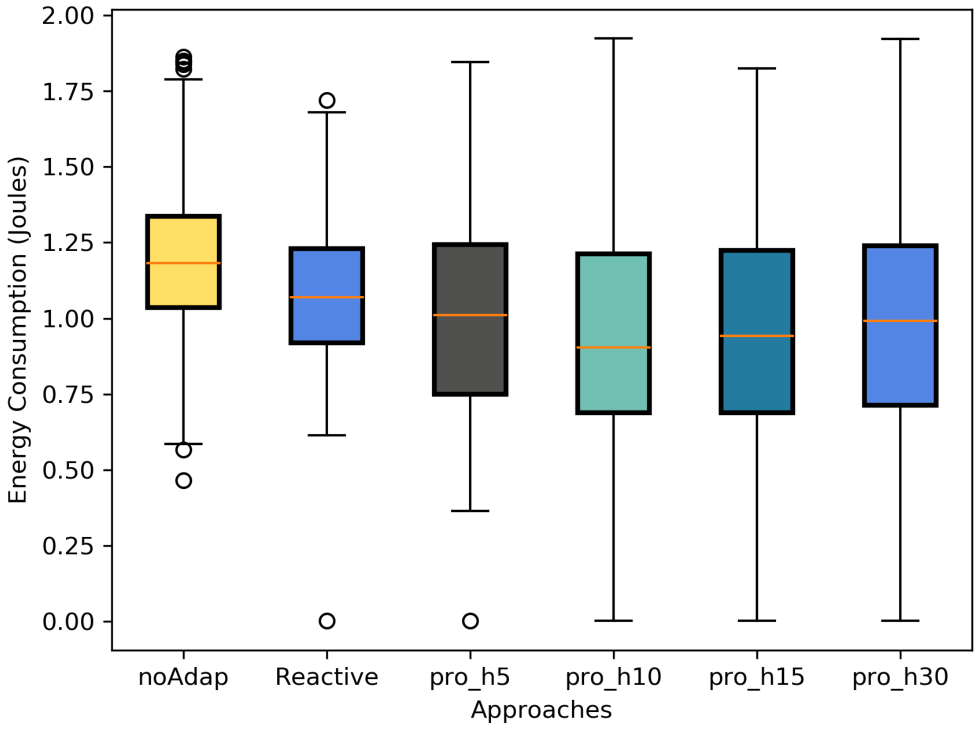


FIGURE 4.9: Box plot depicting the total energy consumed by the sensors per minute

Second, we measured the total energy consumed within the proactive approaches with different horizons when using different decision period intervals, d. This was done to measure the impact of using different decision periods on the adaptation's effectiveness to maintain the energy consumption under the threshold, M_E.

The results are presented in Figure 4.10 in the form of a box plot (y-axis denotes the total energy consumed for the given horizon interval, H. For instance, when H=5, the corresponding y-axis denotes the total energy consumed in 5 minutes). As expected, the approaches, while using a short decision period, s_d, on average, consume the most amount of energy. However, while using the decision period, s_d, the variation in energy consumed is very less. This is because, during a short decision period, adaptations are performed at more frequent intervals. This does not allow the approach to analyze the impact produced by every adaptation, thereby resulting in sub-optimal adaptations. Due to this reason, we can see that, as the decision period changes from s_d to m_d and further to l_d, the variation or the range in energy consumption increases at the same

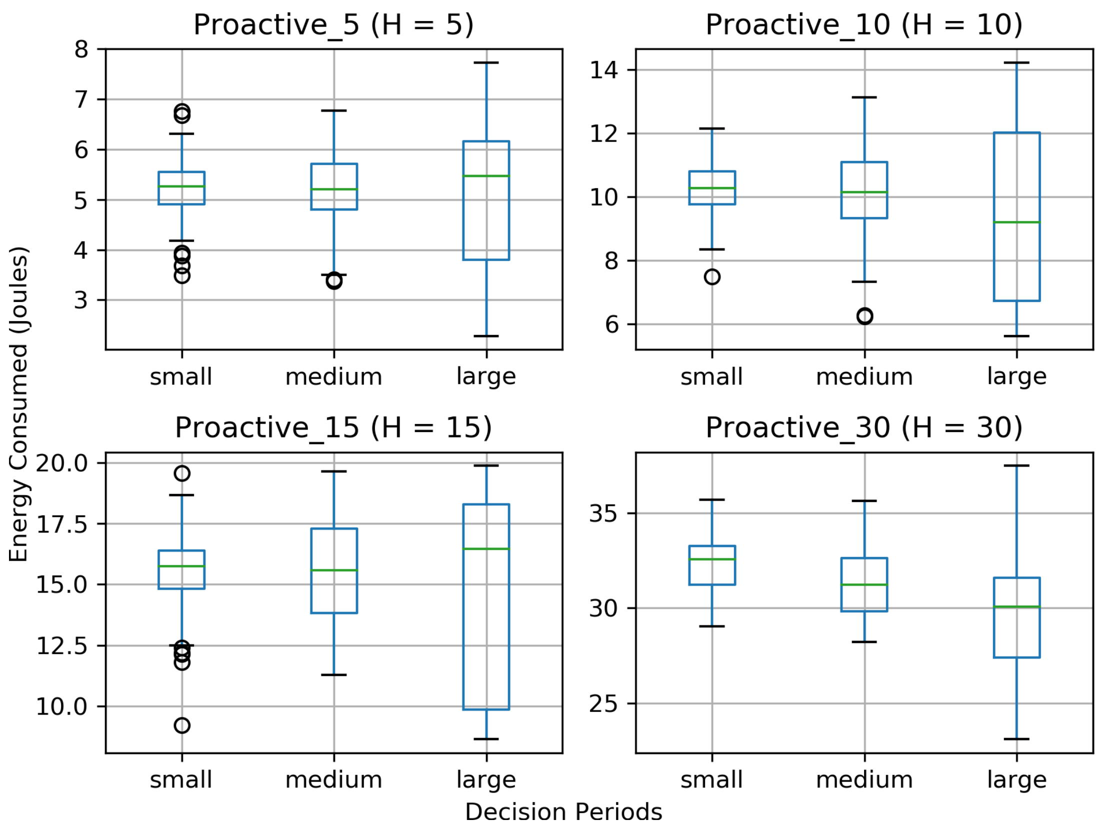


FIGURE 4.10: Box plot depicting the total energy consumed by the sensors using different decision periods per H minutes

time, the average energy consumption decreases. The average offered by proactive_10 (9.5) is almost 20% less than the corresponding M_E for H=10 (11.87). This value is the highest compared with other approaches, and the median value is also more balanced in the case of proactive_10.

Hence, the best energy effectiveness in terms of the total energy consumed and the adaptation quality in terms of adherence to the threshold (M_E) is offered by the approach proactive_10 when using the decision period, $l_d = 10$ minutes.

RQ1.4. How efficient is the approach in terms of the number of adaptations performed ?

For evaluating the adaptation efficiency in terms of the number of adaptations performed, we computed the total number of adaptations applied on the system while using each of the approaches. This was done by parsing the adaptation log files and extracting the information on the total number of adaptations performed for the total period of one day. However, each approach uses different decision periods, and due to this, the number of adaptations performed needs to be measured with respect to the total number

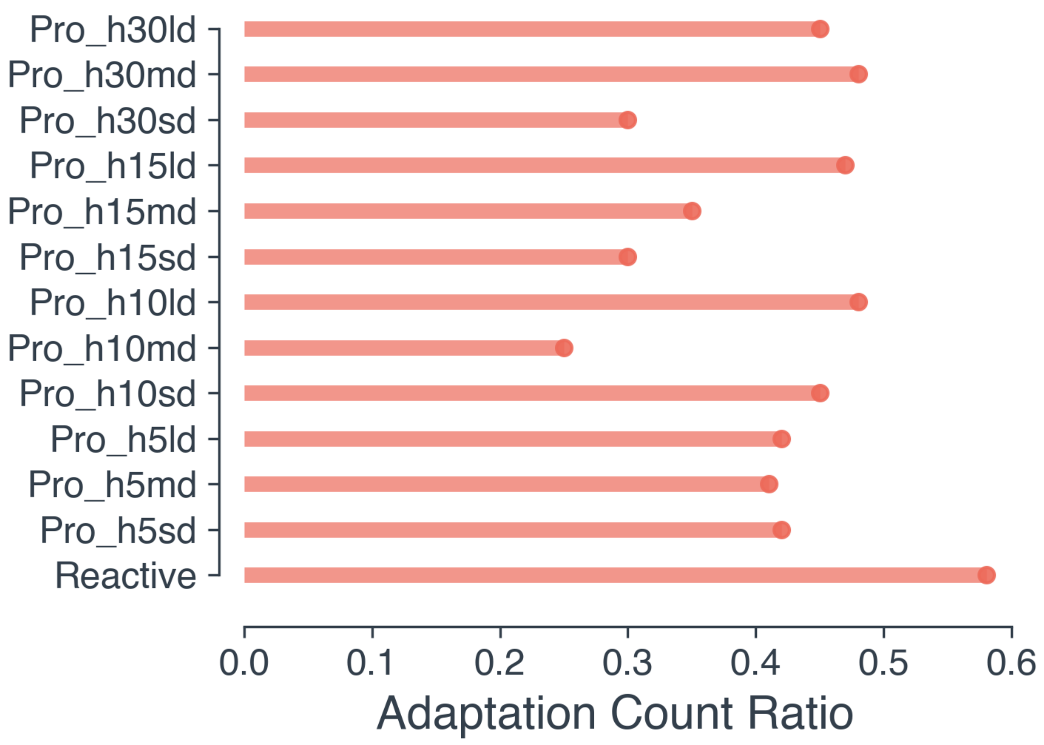


FIGURE 4.11: Number of adaptations performed by each approach

of adaptation intervals. We define this as the adaptation count ratio A_r where,

$$A_r = \frac{\#\, adaptations\, performed}{Total\ number\ of\ possible\ adaptations} \tag{4.9}$$

The results are represented in Figure 4.11. The Reactive approach ends up performing the most number of adaptations, 752 out of 1440 possible adaptation intervals (1440 minutes), and hence, it has the highest value of A_r. On the other hand, we can see that the proactive approaches, except for a few, perform fewer adaptations in general and have a low A_r. The most efficient approach in terms of A_r is Proactive_10 with decision period m_d. However, we have observed from our previous analysis that this does not provide the best effectiveness. Even though Proactive_10 is not highly efficient with respect to other proactive approaches in terms of the number of adaptations, it still provides the best effectiveness to efficiency ratio considering all the other approaches. Thus, we can observe that even though LSTM provides good accuracy for other time horizons, the decision period's choice has a big impact on the effectiveness and efficiency of a proactive adaptation mechanism.

RQ1.5. What is the computation overhead of the overall approach ?

To evaluate adaptation efficiency in computation overhead, we clocked the average time required for our approach to execute an adaptation. The results show that, on average, our approach takes 0.20 seconds for performing the whole process. In this, the major time is consumed for network communications. The speed can be primarily attributed to the usage of trained models and efficient data pipeline, which uses some enterprise-grade big data stack. The training of LSTM with H=10 takes, on average, around 120 minutes. This time changes depending on the time horizon for a fixed number of iterations, neurons, and the depth of the LSTM used. However, this does not impact the real-time process as only the trained models are used for performing analysis and adaptation in real-time.

4.5 Discussion

In this section, we first describe the lessons learned from the evaluation of the approach. We further present some of the threats to the validation of our approach.

4.5.1 Lessons Learned

Accurate QoS Forecasts: One of the crucial challenges in applying proactive adaptation mechanisms, as stated in Section 1.2, has been to generate QoS forecasts with good accuracy [31]. Towards this, our approach exploited the temporal dimension property of QoS data to treat the problem of forecasting QoS as a time-series forecasting problem. However, as can be seen from Table 4.1, traditional time-series forecasting methods cannot be used effectively due to the high variability of QoS, especially in the case of systems like IoT. Moreover, as can be seen from Table 4.2, existing techniques do not guarantee consistent accuracy across different prediction horizons. This is one of the key characteristics required for a proactive approach. Sometimes, the adaptation process can be time-consuming (For example, the use of model-checking techniques on systems with high variability), which means the approach should have the ability to perform effective forecasts across different prediction intervals. Our results on the energy consumption dataset show that high forecast accuracy can be achieved using deep neural networks like LSTM for prediction intervals ranging from 5-time steps to even 30-time steps. Moreover, sudden mode changes cause an increase/decrease in energy consumption. To counter this variation, the MLE is executed periodically to update the models, ensuring that it continuously evolves.

Learning Enabled MAPE-K: Another main challenge in using ML techniques for self-adaptation in general concerns the time required for learning. We address this issue in our approach by extending the traditional MAPE-K loop with a machine learning engine component. Further to support the learning time required for training the learning algorithm, the machine learning engine is executed only at periodic intervals to overcome the issues associated with concept drift [113]. Further, the trained models are made available in the Knowledge repository to Analyze activity to support real-time analysis and predictions.

Impact of Adaptation Intervals on Proactiveness Another challenge in using proactive techniques, as mentioned at the beginning of this chapter, is to identify the best adaptation interval and forecast interval for adaptation. The results from figures 4.10 and 4.9 shows that the best effectiveness in terms of the energy saved is obtained when using intermediatory adaptation intervals and medium-term forecast intervals. When using long-term adaptation intervals, the approach has to wait for a longer duration to identify possible goal violations. During this time, many violations would have happened, especially in highly dynamic systems like IoT. This can be clearly seen in figure 4.10. On the other hand, when using short term adaptation intervals, the approach does not get enough time to capture the change triggered by the previous adaptation. Similar is the case with using short-term and long-term forecast intervals. Hence an ideal choice, as we can observe from the results, is to use medium-term (not too long and not too short with respect to the time of expected QoS violation) adaptation interval with medium-term prediction horizon.

Overall effectiveness and efficiency of Proactive technique Most of the existing self-adaptive mechanisms are reactive in nature [31]. Although they are effective, those techniques trigger adaptation when some QoS violation happens, which brings down the overall effectiveness. It can be clearly seen in the results presented above how a proactive technique can save more energy as opposed to its reactive counterpart. The energy saved by all the proactive approaches (with different prediction horizon and adaption intervals) is higher than the reactive approach. These results clearly indicate the overall effectiveness of a proactive approach for performing self-adaptation, especially in dynamic systems like IoT.

Near-Real-Time Adaptations One of the characteristics of IoT systems is the amount of data produced and the speed at which this data is produced. With the help of enterprise-grade big data stack, we can see that our approach can analyze and perform adaptation in almost 0.20 seconds. This means that the adaptation process by itself will not cause interruptions to the general functionality of the system and the overall approach being proactive ensures that the system never reaches an undesired state. The use of a message broker like Apache Kafka ensures that most of the issues related to reliability, latency, or data availability can be easily handled.

4.5.2 Threats to Validity

1. External Validity: Scalability of our Approach While we have evaluated our approach on a portion of the NdR case study with 22 components, we believe that our approach can be easily extended to more complex systems with a larger number of components. The layered architecture that we have used for the implementation supports horizontal scalability in terms of the number of architecture components. It also supports vertical scalability in terms of the amount of data generated. This will mean just modifying the configurations of the Kafka (to run in streaming context) or adding a batch streaming component like Apache Spark [171]. This is also true with the MLE. The ML techniques we have used can be applied easily on larger datasets. It will be just about fine-tuning the neural network parameters during the model building process.

2. External Validity: Generality of our Approach Although our approach has been applied to a specific case study, the techniques used by our approach can be replicated on any case study/use-case. Such replication is possible as long as we can extract/obtain the energy consumption data either through simulation or from running instance. This same approach can also be used for any other QoS metric such as data traffic, response time, throughput, etc. since any of these data can be modeled as time-series data. Further, the Decision Maker uses the threshold settings and frequency reduction limits provided by the stakeholders, and this can be easily modified depending on the case study. Moreover, our approaches use generic machine-learning techniques that do not use the NdR case study's specific properties or characteristics. We further prove this aspect of the approach later in Chapter 7, where we apply the same approach to accurately forecast response time of microservices.

3. Internal Validity: Incorrect Forecast Any ML process suffers from the problems of accuracy. The model might forecast high energy consumption, whereas the system would not have entered such a state (as we have seen in the case of Proactive_5 approach). We understand this issue, and to avoid this, we suggest using horizons that are not too short and not too long, along with decision periods that allow LSTM to make accurate forecasts. Moreover, to improve our model continuously with new data, we keep performing the model building process mentioned in Section 4.2.2 at regular intervals, thus ensuring that the model keeps improving with new data.

4. Construct Validity: Use of Simulated System for the Experiment Even though we simulated the NdR IoT system, the simulation is performed using a state-of-the-art IoT simulator (CupCarbon [153]), especially for simulating the energy consumption of sensor nodes. Further, we modified the software, hardware, and spatial configurations of the sensor components within the simulation tool to be compliant with the real scenario. This included creating custom sensor scripts, setting the communication protocols, communication range, placement of sensor nodes in the respective (x,y) coordinates, setting the elevation of sensor nodes, etc. All these were done to ensure that the performed simulation respects the configurations of the envisioned real system. Moreover, with regard to the data used for simulating the sensor components, we used the sensor data distribution (which includes the number of people arriving at venues, movement of the car to/from parking lots, etc.) based on real observations from the NdR event.

5. Construct Validity: Energy savings vs. adaptation cost We are performing adaptation based on the ML model's forecast, which is then used for performing decision-making and adaptation. This adaptation performed can have a cost associated in terms of the energy consumed for performing the adaptation. While this is true, we need to consider this cost with respect to the benefit of the gain in the lifetime of the sensor nodes. Moreover, since prediction and decision making components need not be deployed in the edge node, they can be placed in the cloud/server used to execute the IoT system, thereby reducing the adaptation cost in terms of energy consumption. However, this will be relatively less when compared to the total energy savings. To further save the loss incurred in communicating the energy consumption data, the edge nodes can append the energy value to the normal data message (for example, 11.0#100.75J where 11.0 denotes the sensor data and 100.75 denotes the energy remaining in joules).

To summarize, in this chapter, we addressed *RQ1* through a proactive approach that extends the traditional MAPE-K to support self-adaptation in IoT systems using ML techniques. It achieves this by leveraging the QoS data in particular energy consumption. We also provided a complete implementation of the approach using a layered architecture with enterprise-grade big data stack. We performed extensive evaluations of our approach by applying it to a simulated version of the NdR case study. The results proved that our approach's deep neural network (LSTM) outperforms traditional time-series forecasting models. It provides highly accurate energy consumption forecasts for prediction horizons ranging from short intervals of 5 time-steps to even long-intervals of up to 30 time-steps. Further, we also evaluated the effect of different forecast and adaptation intervals on the overall adaptation effectiveness. We also proved how a proactive adaptation mechanism could provide better effectiveness in terms of energy consumption and efficiency in terms of the number of adaptations than a reactive counterpart.

While this chapter focused on the Analyze activity of the MAPE-K and used a simple algorithm for the Plan phase, in the next chapter, we describe how ML can be also used in the Plan activity to enable architectures to learn and improve. We also describe how such an approach can handle multiple QoS parameters.

Chapter 5

Learning to Adapt and Adapting to Learn

In the previous chapter, we saw how we could use machine learning techniques like deep neural networks to perform effective and efficient proactive adaptations. We further saw how such a process improves the overall energy efficiency of an IoT system.

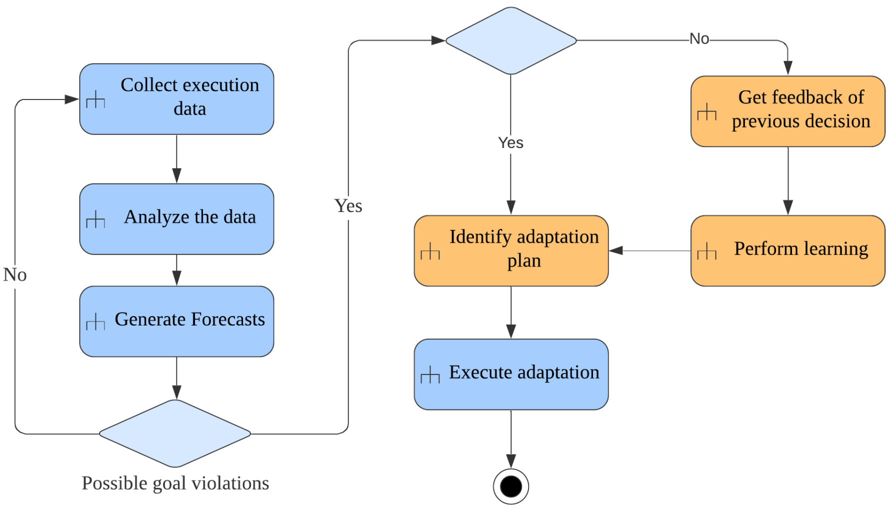


FIGURE 5.1: Learning-driven adaptation

Although the traditional proactive adaptation process works better than the reactive approach and provides better QoS guarantees, one main issue exists. The performed adaptation does not necessarily improve the architecture; it may temporally overcome an impending failure while not preventing the system from entering the same state in the future. In other terms, the architecture adapts to the current context, but it does not learn how to react to new families of the same contexts (refer Section 1.2).

Towards this direction, in this chapter, we extend the proactive adaptation approach with learning capabilities. Figure 5.1 shows the overall process flow such an adaptation process. As seen from the figure, different from a traditional proactive adaptation process (which we have seen in our previous chapter), once a possible goal violation is identified, another condition is checked to see if the approach runs for the first time. If yes, then an adaptation plan is determined, and the adaptation is executed. But, if it is not the first execution of the approach, then it obtains the feedback of the adaptation made at the previous step, learns from it, and then comes up with a new adaptation plan which is then executed. So as we can see with this process flow, the approach will have the ability to continuously learn from every adaptation performed. Moreover, the adaptation will be performed based on the learning obtained in the form of feedbacks. In this manner, the system will learn how to adapt and further learn with each such adaptation. Such an approach is used in this chapter to address the RQ2 *How can machine learning be used to continuously improve the adaptation process?*

The overall approach is based on the MAPE-K loop. It extends the previous chapter's approach by adding *a reinforcement learning-based decision-maker in the Plan activity of the MAPE-K loop.* The approach thereby uses a combination of deep neural networks and Reinforcement Learning (RL) techniques to enable proactive self-adaptation driven by continuous learning and vice versa. This approach is further applied to an IoT system for continuously monitoring and forecasting QoS data, particularly the energy consumption and data traffic data, and additionally perform adaptations using the different self-adaptation patterns available for IoT systems. More specifically, the approach offers the following advantages:

1. Continuously monitors the QoS parameters, particularly the energy and data traffic of the given IoT system. It then uses LSTM networks to build forecast models for forecasting each of the QoS parameters.

2. The approach performs adaptation based on the generated forecasts. It allows stakeholders to specify the energy and data traffic constraints the system needs to meet. It then uses a RL algorithm to select the best adaptation patterns based on contexts using RL techniques.

3. Uses the generated forecasts as feedback mechanisms to measure the effectiveness of a selected decision.

4. Continuously performs the loop of forecasts, selection, adaptation, and learning (through feedbacks) to improve the decision-making process.

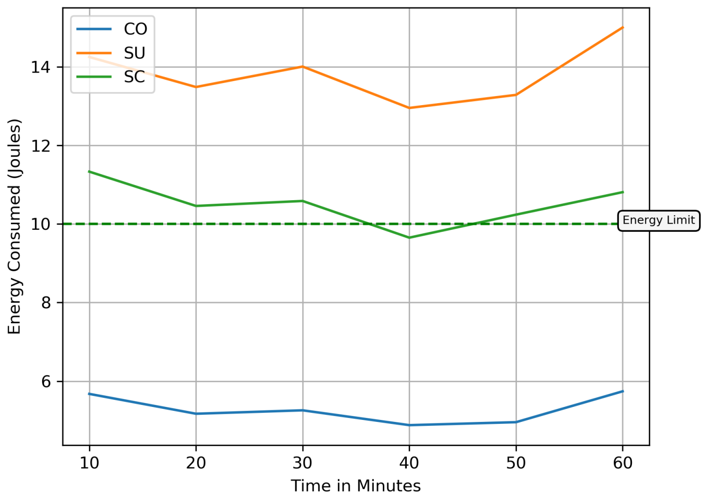
5.1 Motivating Scenarios

This section presents the motivation behind the work described in this chapter through the NdR case study described in Section 2.5.

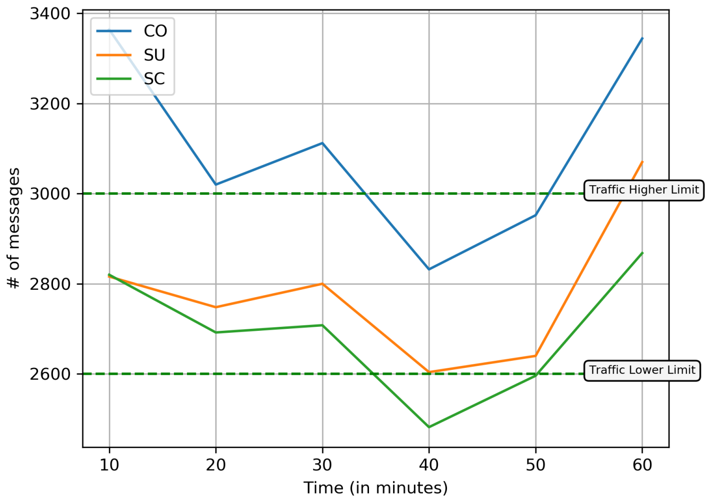
Another significant QoS challenges in IoT systems, similar to ensuring energy efficiency concerns with minimizing the system's overall data traffic [172]. The impact of data traffic is such that the global carbon footprint is expected to increase with an exponential increase in ICT data traffic [173]. Considering the increasing use of IoT devices, there will be a tremendous increase in data traffic generated and energy consumed by these systems. One re-configuration method to reduce data traffic and energy consumption, as we have seen in the previous chapter, consists of reducing the sensor data acquisition frequency. However, such a solution may affect service accuracy since it causes a drastic reduction in data traffic. Another method consists of reducing the data traffic by using decentralized architectures, but this might increase the sensors' energy consumption. This mandates the use of structural, architectural adaptation techniques that, *while offering the desired energy efficiency, maintain data traffic between the required performance thresholds.* For this purpose, we rely on three different self-adaptation patterns for architecting IoT systems proposed in literature (refer Section 2.1.5) [56].

5.1.1 The case of NdR: Handling tradeoff's

As we have seen in the previous chapter, One of the critical constraints we have in the NdR case study is to ensure that the sensors can operate in an energy-efficient manner. This is because they are battery powered and they can run only for half a day with a normal battery setup. Even though charging can be performed, the charge could go down, and plugging in a new sensor (or charging the existing one) during peak hours can be costly. Moreover, we also need to ensure that the system's overall data traffic is optimized since high data traffic can affect performance. In contrast, little data traffic might cause issues with service accuracy. One of the easiest solution to solve this problem is to use one of the self-adaptation patterns. However, each pattern has its advantages and disadvantages, especially regarding energy savings and the amount of data traffic generated. Figure 5.2 illustrates such a scenario. It shows a sample simulation of the envisioned system using each of the three patterns for a one-hour duration. The dotted lines in Figure 5.2A) represent the maximum total energy that can be consumed for 10 minutes. On the other hand, the dotted lines in Figure 5.2B represent the maximum/minimum limit above/below, which leads to congestion/service accuracy issues.



(A) Energy consumption



(B) Data traffic

FIGURE 5.2: One hour simulation of NdR architecture

We can see that the CO pattern (lower line in Figure 5.2A) ensures that energy consumption stays within the limit. However, the same pattern does not guarantee a stable performance as the data traffic of the system goes above the threshold (topmost line in Figure 5.2B)). This behavior is because CO, being semi-decentralized, has an additional controller component in between; hence, the sensors' total energy will be less, whereas the data traffic in the system will be high. The same is the case with SU and SC patterns: they initially perform reasonably well in terms of data traffic, but towards the middle, we can see that they go below the minimum threshold, causing potential service accuracy issues. Thus, those that work well with energy do not work with traffic and vice versa.

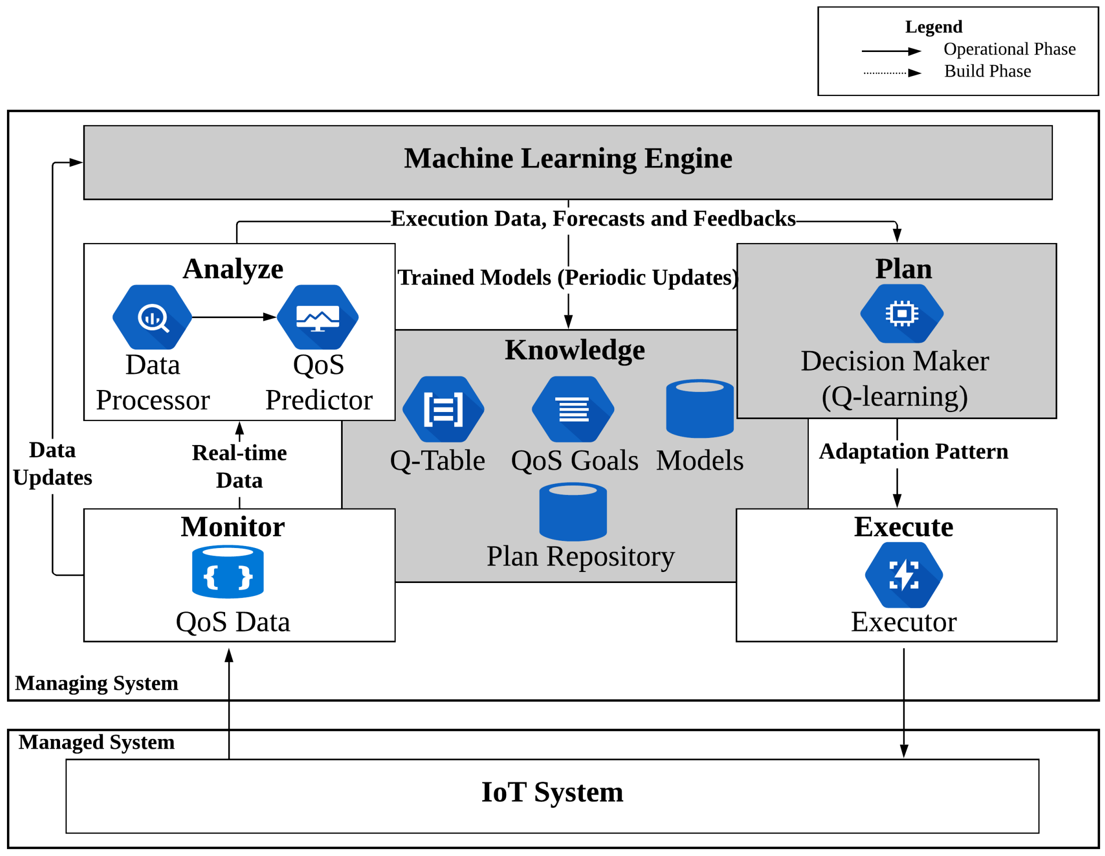


FIGURE 5.3: Approach Overview

Henceforth, an ideal IoT system should have the ability to proactively predict the consumption and intelligently use different self-adaptation patterns based on the context to keep the energy and data traffic consumption within limits. Moreover, it may happen that a particular pattern selected behaves indifferently (due to sudden mode changes/environmental uncertainties). Thus, the system should have the ability to learn from these situations and improve the decision-making process.

5.2 Approach Overview

In this section, we first present an overview of the approach. We then provide details on how it uses different machine learning techniques and the different self-adaptation patterns to proactively and autonomously adapt the software architecture, thus ensuring that the architecture learns and improves over a period of time.

Figure 5.3 shows the overview of the approach. It is based on the MAPE-K loop, as we have seen in the previous chapter. However, while the last chapter focused more on the *Analyze* activity of the MAPE-K, in this chapter, we shift the focus to the *Plan* activity of the MAPE-K (highlighted using grey boxes in Figure 5.3). Further, we elaborate on how the *Machine Learning Engine* (MLE) along with the *Analyze* and *Knowledge* activity enables the architecture to adapt and learn with every adaptation performed continuously.

The *Managed System* represents the IoT system, which has been implemented based on defined software architecture. In this approach, we model the architecture such that it will be able to switch between multiple patterns depending on the decision produced by the plan activity. During modeling, the operational modes of sensor components need to be defined. We also define the conditions for transitions between the different modes during this phase.

The *Managing System* consists of the various activities which use a combination of deep neural networks and RL techniques to perform adaptations. It is responsible for analyzing the data produced by the managed system, forecasting the expected QoS (*Analyze Activity*), and generating actionable insights in the form of adaptation decisions (*Plan Activity*). The decisions generated by are then used for executing the adaptation (*Execute Activity*). This learning and adaptation loop keeps continuing, thereby enabling the system to become efficient with respect to the energy and data traffic thresholds.

5.3 Learning-driven Adaptation Approach

In this section, we extend the proactive approach (presented in Chapter 4). This further enables the architectures to adapt and learn continuously from each adaptation performed. We particularly focus on how the approach i) stores the knowledge, ii) performs learning, and iii) executes the adaptation.

5.3.1 Knowledge

As explained in the previous chapter, the *Knowledge* acts as central storage for different types of knowledge required by different components of the managing system for performing the adaptations. It stores four different types of information:

i) QoS Goals: It consists of the energy and data traffic goals that need to be satisfied. These goals define the different energy and data traffic thresholds by specifying the acceptable energy consumed and data traffic generated by the system for a given time interval. The respective stakeholders define the thresholds through a configuration file. This process involves defining four variables: i) high energy (he), the maximum energy that can be consumed by the sensors every 10 minutes; ii) low energy (le), the threshold below which the sensors save the maximum energy; iii)high data traffic (hd), the maximum allowed traffic to avoid congestion; iv) low data traffic (ld), the minimum traffic required to maintain service accuracy.

ii) ML Models: It consists of the different trained ML models, which are further used by the *Analyze* activity to forecasts the expected QoS. In this chapter, we extend the ML Models store to support storage and versioning of different types for forecast models: i) forecast model for energy consumption ii) forecast model for data traffic.

iii) Plan Repository: The plan repository stores the different adaptation plans, in this case, the different adaptation patterns, and further stores the configurations for each pattern. The *Executor* component uses this for executing the adaptation.

iv) Q-table: It is a lookup table generated and used by the *Decision Maker* component to identify the best adaptation pattern based on a given scenario. It enables the Decision Maker to converge towards the selection of better adaptation decisions with time.

5.3.2 The Machine Learning Pipeline

Figure 5.4 shows the overall machine learning pipeline view of the approach (It is the same as the one presented in Chapter 4, we have represented this pipeline to give an overview of the overall machine learning flow). It represents the *Analysis* and *Planning* phase of the MAPE-K loop. As we have seen in the previous chapter, the overall process can be divided into two phases, *Build Phase* and *Operational Phase*. The former refers to the process of building machine learning models for forecasting QoS parameters. The latter refers to the run-time phase, where the machine learning models are used to forecast the energy consumption and data traffic based on the run-time data and select the best adaptation pattern for architectural adaptation. When running the *Operational Phase*, the *Build Phase* is run in parallel (at regular intervals) to ensure that the machine learning model is up-to-date with new patterns in the data. Further, we provide details on each of the components of the machine learning pipeline.

5.3.2.1 Data Store and Feature Extractor

The *Data Store* consists of logs containing information on every interaction that happens within and between the sensors at every instant of time. The intervals of time may not necessarily be uniform. Rather it is based on the fact that whenever there is any interaction within or between the components during the simulation/execution, the energy consumed for corresponding interaction is logged. The information of the interaction can be used to calculate the data traffic by computing the number of messages exchanged. These logs are then passed to the *Feature Extractor* for further processing.

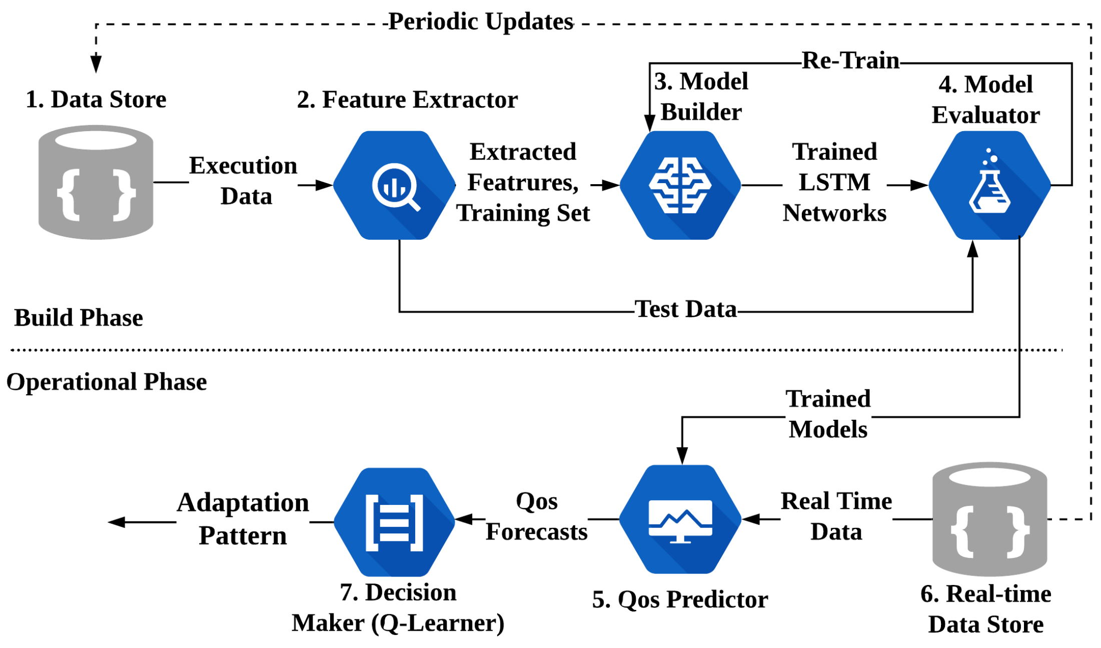


Figure 5.4: Machine Learning Pipeline of the Approach

The *Feature Extractor* (component 2) first splits the data in the *Datastore* into two datasets, namely energy data and traffic data. Energy data consists of the amount of energy consumed for every component in different instants of time. Traffic data consists of the total data traffic generated in the system for different instants of time. Each of these data sets is further processed using four steps, as explained in Chapter 4: i) Time series Modeling, ii) Normalization, iii) Conversion to Supervised learning problem, and the iv) Train and Test data generation.

Finally, each of these datasets is divided in the ratio 7:3, where 70% of data becomes the training set, and the rest 30 % forms the testing set. The training sets are passed to the *Model Builder*, and the testing sets are passed to the *Model Evaluator*.

5.3.2.2 Model Builder and Model Evaluator

The Model Builder (component 3) as explained in Chapter 4, forms one of the key components of the approach as it is responsible for building the forecasting models. It uses LSTM networks and creates two forecasting models, one for forecasting the total energy consumption of the sensor components in the architecture and the other for forecasting the system's total data traffic.

In our previous chapter (Chapter 4, Section 4.2.2.2), we have shown how LSTM networks can forecast energy consumption of the sensor components. We have also demonstrated why LSTM networks are more effective than traditional models like ARIMA, SES, etc.

This chapter uses the same approach to create the LSTM networks but for both energy consumption and data traffic forecasting.

The LSTM networks are trained using the training sets to generate forecast models. The *Model Evaluator* checks the accuracy of the models on the respective test datasets. This is performed by using the test data sets to forecast the energy and data traffic values and comparing it with the actual values. In the case of lower accuracy, retraining is performed by modifying the network parameters, and the trained models together form the *QoS Predictor (Component 5).*

5.3.2.3 Real Time Data Store and QoS Predictor

During the operational phase, the data generated during the execution/simulation is ingested into the real-time data store (component 6 in Figure 5.4, represents the QoS Data in Monitor activity, Figure 5.3), and this is immediately sent to the QoS predictor (component 5 Figure 5.4 represents the QoS predictor in Analyze activity, Figure 5.3). It keeps aggregating the real-time data until the number of observations, n, becomes equal to the lag, l (the number of observations required for the model to forecast). It then uses the trained LSTM networks to forecast the energy consumption of the sensors and the data traffic of the system for the next H time steps.

5.3.2.4 Decision Maker (Q-learning)

The *Decision Maker* component (component 7 in Figure 5.4, represents the Decision Maker in Plan activity, Figure 5.3) is responsible for selecting the best adaptation pattern based on the forecasts provided by the *QoS predictor*. It uses Q-learning, a reinforcement learning-based technique [61, 73] to perform the decision making. Q-learning is a widely used method for decision-making scenarios due to their ability to come up with optimal decisions through a model-free learning approach. For further details on Q-learning refer to Section 2.2.4.

The problem of selecting the best adaptation strategy can be converted to a famous "Robot in the Grid World" problem [174], where the running software architecture can be considered as a robot that has to navigate through a grid where the goal is to reach to a position in the grid which enables the architecture to become efficient with respect to energy and data traffic thresholds as defined in *QoS Goals* (Figure 5.3). Figure 5.5 shows how an adaptation decision making problem can be modeled as a grid problem. SA denotes the current software architecture of the system. Each grid represents the combination of thresholds, as defined in 5.3.1. For instance, the grid $HEHD$ represents the case when the system is consuming high energy and high data traffic. The objective of the *Decision Maker* component at every point will be then to select the best strategy in the form of patterns such that, given the system's position, the architecture SA will eventually move towards the required goal.

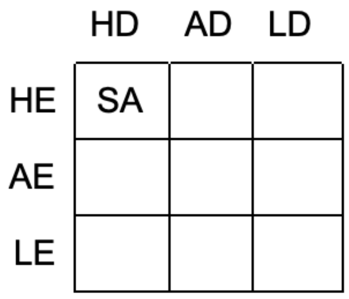


FIGURE 5.5: Grid problem representation of the decision making problem

Figure 5.6 represents the mapping of the decision making problem into a q-learning problem (refer Section 2.2.4). The Decision maker represents the *Agent* and the running IoT system represents the *Environment*. First part of Q-learning then, is to divide

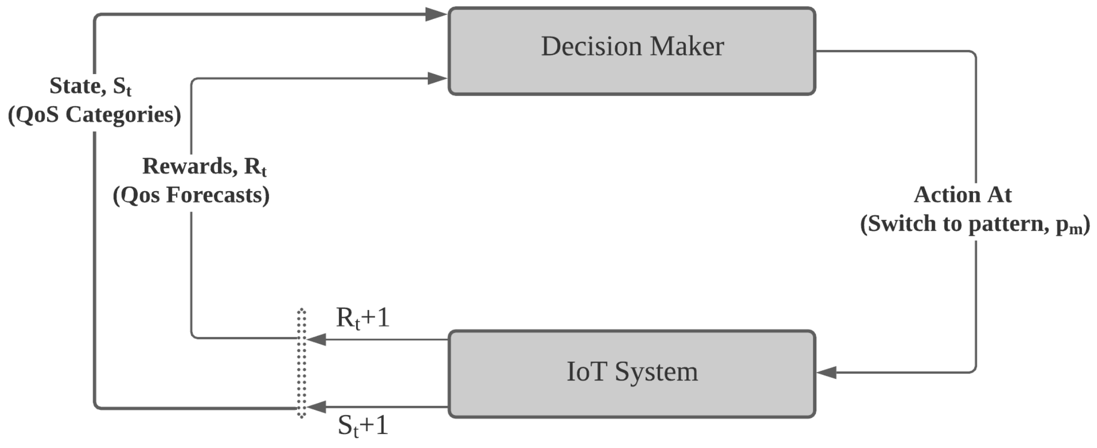


FIGURE 5.6: Decision Making through Q-learning

the problem into set of States, S Actions, A and Rewards, R. The forecasts of the *QoS predictor* can be classified into different categories based on the values of forecasts with respect to QoS thresholds. The state space represents a set that contains all the categories based on the thresholds combined with each pattern, p available. For example if there are three patterns $\{p_1, p_2, p_3\}$, and QoS thresholds = $\{he, hd, le, ld\}$ then state set can be: $S = \{he\_hd\_p1, he\_hd\_p2...., le\_ld\_p2\}$ Thereby, resulting in total of 27 states (3 patterns and 9 combination of QoS thresholds). The action space on the other hand, consists of the possible actions which allow the architecture to move in the grid. Let $P = \{p1, p2, ...pm\}$ represent the set of available patterns then, action space, A can be defined as $A = \{0, 1, ...m\}$ where any $i \in A$ represents switching to the pattern $p_i \in P$.

Rewards, R consists of a set of integer values which corresponds to every state, s_i in S. It represents the reward for moving to a particular state.

S and A together form an $N \times M$ *Q-table Matrix* or Q-table where *N represents the states* and *M, the actions*. Each value inside Q-table corresponds to a (state s,action a) pair and it indicates the relevance of taking an action a from state s. At a given instant of time, t, the energy and the data traffic forecast produced by the *QoS Predictor* along with the current pattern in use, is used to determine the state, action inside the Q-table.

At every instant of time, t, based on the Q-table, the *Decision Maker* performs an action, A_t by switching to one of the patterns, P (Figure 5.6). Such a switch moves the state of the IoT system from one to another, and this is determined by the QoS categories, as explained above. Further, for every action, a_t performed at a time, t, the Decision Maker obtains feedback in the form of Rewards, R, and this is assigned based on the QoS forecasts obtained at a time, $t+1$. This process allows the decision maker to improve the overall decision making process continuously.

The complete algorithm of *Decision Maker* is presented in algorithm 2. It first takes as inputs the states and action, S and A as mentioned above. It then uses the set of thresholds (line 3 and 4), T defined by the stakeholder in the form of QoS Goals, the forecast set, F produced by the QoS predictor; the Reward list, R, which contains a set of rewards corresponding to each state; and a pattern mapp_{map} which consists of two-way mapping between an action, a and a pattern, p. The algorithm also takes as inputs two parameters α and γ where $0 \leq \alpha \leq 1$ represents the learning rate, which represents the importance given to the learned observation at each step, and $0 \leq \gamma \leq 1$ represents the discount factor, which denotes the weight given to the next action.

It then finds the current state of the system (lines7-9), s based on the forecasts F, and thresholds, T. It first identifies the category, C of the forecast. This is then combined with the current pattern, p to obtain the state s as C_P. This is followed by identifying the reward, r, to be assigned for that state from the reward list, R (line 10). Following this, the action, a corresponding to the pattern, is identified from the pattern map, P_{map}. The algorithm then selects the (state, action) pair from the Q-table with the maximum Q-value. This is then assigned as the next state and action, $((s', a'))$ (line 12). The Q-table is then updated using the Q-Function:

$$(s_t, a_t) = (1 - \alpha) * Q(s_t, a_t) + \alpha * (r_t + \gamma * max(Q(s_t + 1, a))) \tag{5.1}$$

Following this, the pattern p corresponding to the action is selected (lines 14-15). One of the crucial parts of Q-learning is the effective use of rewards. By assigning negative rewards (penalties) to states that violate thresholds and positive rewards to states within

Algorithm 2 Decision Maker Algorithm

```
Require: :
1: States $S = \{s_1, s_2, s_3, ...., s_n\}$
2: Actions $A = \{0, 1, 2....m\}$   ▷ represents the switch to each $m$ patterns
3: Thresholds $T = \{he, le, hd, ld, ...\}$   ▷ Thresholds as per QoS requirements
4: Forecasts $F = \{ef, df\}$   ▷ Forecasts of energy and data traffic
5: Rewards $R = \{r_1, r_2, ..., r_n\}$   ▷ Reward for each of the state
6: Pattern map $P_{map} = \{p1 : 1, p2 : 2, p3 : 3....pm : m\}$   ▷ two way mapping between
   patterns and actions
7: procedure DECISION-MAKER($S$,$A$,$T$,$F$,$R$,$p$,$P$,$\alpha$,$\gamma$ )   ▷ Find the state of the system
   from the forecasts
8:     $C \leftarrow identify\_category(F, T)$   ▷ Get Category from Forecasts
9:     $s \leftarrow (C, p)$   ▷ combine category and p to form the state
10:    $r \leftarrow R[s]$   ▷ Reward for attaining the state, $s$
11:    $a \leftarrow P_{map}[p]$
12:    $(s', a') \leftarrow argmax_a Q(s, a)$
13:    $Q'(s, a) = (1 - \alpha) * Q(s, a) + \alpha * (r + \gamma * max(Q(s', a)))$
14:    $a \leftarrow a'$   ▷ The action to reach that state
15:    $p \leftarrow P_{map}[a]$
16:    return $p$
```

thresholds, Q-learning ensures that any decision leading to higher energy/higher data traffic state will get a high penalty in the form of a negative reward. Hence as time progresses, the Q-table will be filled with values where, given any state, the algorithm will be able to select the best adaptation pattern, thus ignoring the patterns that can lead to high penalties.

5.3.3 Execute

The Execute activity is responsible for executing the adaptation and thereby adapting the architecture. It achieves this with the help of an *Executor* component. The adaptation decisions produced by the *MLE* basically contains the pattern that has to be applied. These decisions are sent from the MLE in the form of configuration files. The adaptation manager then applies the adaptation by dynamically updating the existing architecture based on the pattern, P given by the Decision Maker component.

The use of machine learning techniques allows the approach to foresee the possible energy consumption and proactively perform automatic adaptations of the architecture, thus enabling the IoT system to learn, improve the architecture and become efficient with respect to the energy and data traffic constraints.

5.4 Architecture and Implementation

In this section, we provide an overview of the technology stack used for the implementation. As in Chapter 4, we consider the case study mentioned in Section 2.5 for the implementation of the approach.

We use the same layered architecture with enterprise-grade big data stack, as explained in the previous chapter (Figure 4.5) for the implementation of the approach. We use both Java and Python for the implementation. The development of the NdR case study and corresponding simulation is performed using the CupCarbon simulator. We use a *.json* file for defining the energy and data traffic constraints (Thresholds, T). The source code of CupCarbon is also modified to support dynamic pattern switch in run-time based on the decision produced by the *Real-time* layer. The Decision Maker component (Figure 4.5) is modified using native python to implement the algorithm 2. Further, the Executor component (Figure 4.5) is implemented using core Java to dynamically trigger the pattern change in CupCarbon based on the decision produced by the Decision Maker component. The approach automatically executes the batch phase every 3 hours to re-train the machine learning models for forecasting data traffic and energy consumption.

5.5 Experimentation and Evaluation

In this section, we describe how we evaluated the approach. First, we describe the data used for the evaluation of the approach. Then we evaluate the approach based on its effectiveness and efficiency. The *effectiveness* is measured by answering the following questions :

RQ2.1. How accurate and stable are the forecasts made by LSTM Networks?

RQ2.2. How much energy and data traffic is saved by the approach as compared to each individual pattern?

RQ2.3. How much does the adaptation quality improve with time?

The *efficiency* of the approach is measured by answering the following questions :

RQ2.4. How many adaptations are performed by the approach?

RQ2.5. How much time does the approach takes to perform adaptation?

Datasets	**Simulation Time (sec)**	**REL Size (MB)**	**AEL Size (MB)**	**RDFL Size (MB)**	**ADFL Size (MB)**
CO	2,59,200	704.7	2	702.1	0.10
SU	2,59,200	497.9	2.1	520.2	0.13
SC	2,59,200	469.9	2.1	544.2	0.14
Adap	8,64,000	1771.1	5,8	1720.3	0.34

Table 5.1: Summary of Datasets (REL : Raw Energy Logs, AEL : Aggregated Energy Logs, RDFL : Raw Data Traffic Logs, ADFL : Aggregated Data traffic Logs)

5.5.1 Experimental Setup

For experimentation, we implemented the approach on a high performance computing cluster consisting of 4 compute nodes (same setup as the one used in Chapter 4). Each of these nodes runs on a Dell R730 CPU with an Intel Xeon Processor comprising 20 cores with CPU 256 Gb of RAM. We used one compute node to run Apache Kafka. Elasticsearch and Kibana were run on the second and third compute node with the fourth one being used for creating and testing machine learning models. This separation was done so as to mimic the real IoT data pipeline setup. In order to emulate a real-time scenario, we deployed the CupCarbon IoT simulator on a desktop machine running on Intel i5, 2.6-3.2 GHz processor with 16 Gb of RAM.

5.5.2 Data Setup

We simulate the architecture using the CupCarbon simulator for a period of 10 days. The sensor data were generated based on a Poisson distribution with mean values selected based on the case study's real observations (similar to what was done in Chapter 4). During the simulation, we randomly vary the patterns every 10 minutes. This is performed automatically by generating random numbers between 0 and 2 and then using this to select a pattern. This is done to capture the different possible variations in the data in a real scenario. The same process also generates execution logs (to extract traffic-related information). These are further processed by the approach to generate aggregated data sets, which are divided into the ratio 7:3, resulting in a training set of 10,080 samples and a testing set of 4,320 samples. To test the stability of the forecasts made by the approach, we created three sets of datasets by simulating the architecture using each of the three patterns for three days. These datasets were then used as validation datasets. Table 5.1 shows the summary of the validation datasets as well as the adaptation data set (Adap). These datasets represent distinct scenarios that help us to evaluate better the stability of the forecast made by our approach using the LSTM models. The complete implementation, along with the source code and datasets, can be found here. [1].

5.5.3 Results

RQ2.1. How accurate and stable are the forecasts made by the LSTM Networks?

We use a forecast horizon, H of 10, and lag value, l of 10, which means that the models will forecast the energy/data traffic for the next 10 minutes based on the last 10-minute values. These are selected considering the results obtained from the experiments in Chapter 4 and considering the overfitting and accuracy issues. We use the Mean Absolute Error (MAE) loss function and the efficient Adam version of stochastic gradient descent [65] for optimization of the LSTM networks.

We use the Normalized Root Mean Square Error (NRMSE) of the predictions on the test sets for measuring the forecast accuracy. This is done to normalize the RMSE values to a common range, as there are different types of test datasets. Normalized RMSE, NRMSE for a dataset, Y with n samples is given by the formula:

$$NRMSE = \frac{RMSE(Y)}{\sigma(Y)};\ RMSE = \sqrt{\frac{1}{n}\Sigma_{i=1}^{n}\Big(p_i - y_i\Big)^2}$$

Where, $\sigma(y)$ represents the variance, p_i the predicted value and y_i represent the actual value. Since we have two models, one for forecasting the energy consumption and the second one for forecasting data traffic, we run the accuracy tests on both datasets.

1. Accuracy of energy forecasts: We build the energy forecast model using one hidden layer consisting of 110 neurons and 220 neurons (22 components * 10-minute forecast for each, refer Section 4.2.2.2)) in the input and output layers. These numbers were selected through experimentation and based on principles listed in [175]. The model is fit in 150 iterations. This model is then used to generate a 10-minute forecast for every observation, o on the testing set by considering the set $o - 10$ observations. The model generates energy forecasts for each of the n components of the architecture. This is then summed up to develop the total energy forecast of the sensor components in the system. This is performed for every observation in the testing set to generate a forecast vector, F. For computing the actual energy vector, A, we follow the same process, but instead of using forecasts provided by the LSTM network, we compute the actual total energy of the system after 10 minutes based on the current observation, O We then computed the NRMSE between the actual vector, A, and forecast vector, F. The

[1]https://github.com/karthikv1392/IoTAdaptaiton_ML

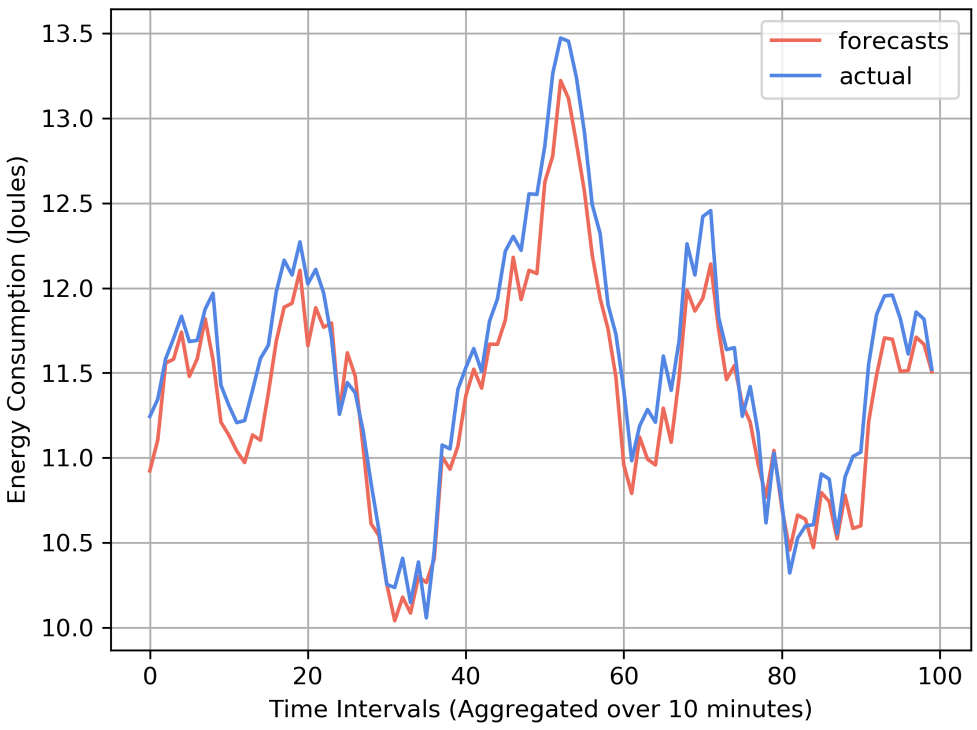


FIGURE 5.7: LSTM Energy Forecasts

NRMSE calculation for the LSTM model resulted in a value of 0.846, which is really good as the value indicates the NRMSE over a sample size of 4,320. Figure 5.7 shows the plot of the actual versus forecast vector. We can clearly see that the LSTM model's prediction is almost able to follow the curve of actual energy.

2. Accuracy of Data Traffic Forecasts: For creating the data traffic LSTM model, we follow the same process as explained above by using the data traffic dataset. Since it is a univariate data, 10 neurons (10-minute forecast) are used in both input and output layers. The model was fit in 100 iterations. The model is then applied on the testing set to generate the forecast vector, F, as explained above, except the fact that this is a univariate data, and the tenth-minute forecast is calculated for the overall system. We then used the actual vector, A, from the testing set to compute the NRMSE. The calculation resulted in a value of 0.90. Figure 5.8 shows the plot of forecasts vs actual vector. In this case, also, we can observe that the model is able to provide forecasts that nearly follow the actual values.

3. Stability of forecasts: Each of these trained models is then applied to the validation sets to test the stability of the forecasts made by the respective models. The results are represented in table 5.2. The model is trained using the *Adap* dataset. We can clearly see that the accuracy level of forecasts of both energy and data traffic is mostly

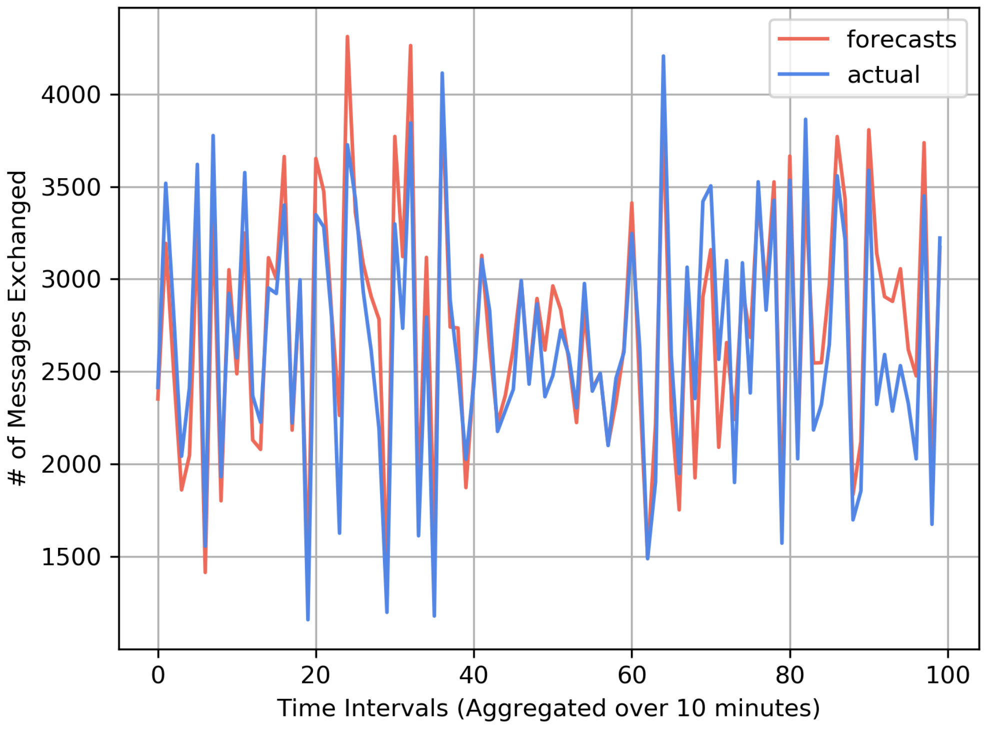


Figure 5.8: LSTM Data Traffic Forecasts

consistent across the datasets The consistent stability is mainly because the training

Dataset	NRMSE Energy	NRMSE Data
Adap	0.87	0.90
CO	0.88	0.75
SU	0.66	1.29
SC	0.50	0.76

Table 5.2: NRMSE of the machine learning models on the validation and adaptation datasets

dataset considers the different possible variations which allow the LSTM's to provide accurate and stable forecasts. This is very important as the adaptation is performed based on the LSTM networks' forecasts, and inconsistency in accuracy and stability can make decision-making erroneous.

RQ2.2. How much energy and data traffic is saved by the approach as compared to each individual pattern?

In order to calculate the energy and data efficiency of our approach, we first calculated the energy consumption and data traffic of the system while using each of the three patterns (Approaches CO, SU, SC). This was performed by simulating the system with

Approach	**Energy (Joules)**	**Data Traffic (# messages)**	**Max DE**	**Min DE**	**Max EE**
CO	752.43	437885	90	0	0
SU	1995.85	382896	5	20	143
SC	1498.57	398393	14	8	104
Our Approach	1051.38	398294	18	7	8

Table 5.3: Energy Consumption and Data traffic of different approaches (DE : Data traffic Exceedance, EE : Energy Exceedance

Time (H)	**DCO**	**ECO**	**DSU**	**ESU**	**DSC**	**ESC**	**DR**	**ER**	**# adap**
4	72473	123.61	63156	326.69	65959	244.92	65684	167.62	10
8	145130	247.65	126624	656.22	132081	492.12	132239	351.99	20
12	218513	373.00	191058	988.14	198939	741.41	199212	531.40	13
16	293133	500.41	256587	1326.35	266810	996.20	267202	708.82	12
20	367397	627.00	321036	1662.16	334248	1247.13	334468	881.65	10
24	437885	752.43	382896	1995.85	398393	1488.57	398294	1045.96	14

Table 5.4: Adaptation Effectiveness over time (DCO : Data Traffic using CO, ECO : Total Energy of Sensors while using CO, DR : Data Traffic using our approach, ER : Energy consumption while using our approach, #adaptations performed by our approach)

the respective pattern using CupCarbon. The simulation was carried out for a period of 1 day to emulate our case study's real scenario. Every sensor component was assigned initial energy of 19160.0 Joules, which is the default setting of CupCarbon. We then integrated our approach with the existing system as explained in Section 5.4 and the simulation was performed. We use learning rate $\alpha = 0.02$ and discount factor, $\gamma = 0.2$. We use the thresholds ($T = \{he = 10.0, le = 6.0, hd = 3000, ld = 2500\}$). These thresholds are defined by considering the operational constraints we have from the case study.

Table 5.3 shows the total energy consumed by the sensors and total data traffic of the system while using each of the approaches. It also shows the number of times each of the approaches exceeded the maximum and minimum thresholds. Although CO consumes the least energy of 747 Joules and does not exceed the maximum energy limit, the data traffic amounts to 437885 messages, and it exceeds Max DE around 90 times. On the other hand, SU and SC amount to lesser data traffic but each of them exceeds the Min DE 20 and 8 times, respectively. They also end up consuming more energy with SU consuming more than SC, which is primarily due to the fact that in SU, the controller is within the sensor component. This also means they exceed the Max EE limit. Our approach, on the other hand, consumes 1051.38 Joules, which is 944 Joules less than SC and 153 Joules less than SU. It also exceeds the energy limit just 18 times. Although It consumes around 300 Joules more than CO, we can clearly see that the data traffic is

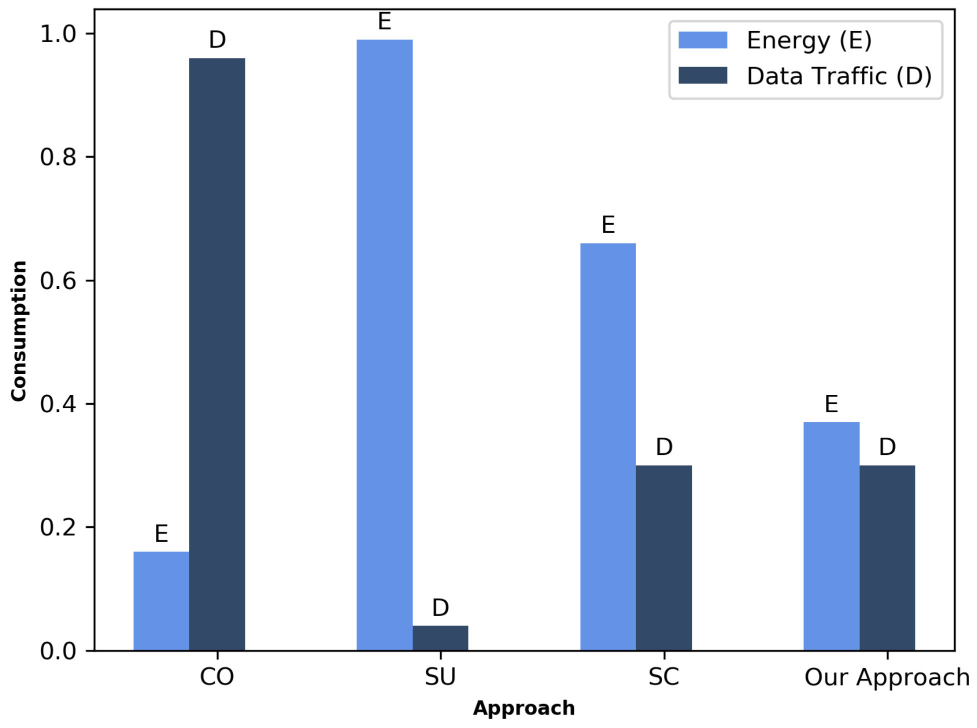
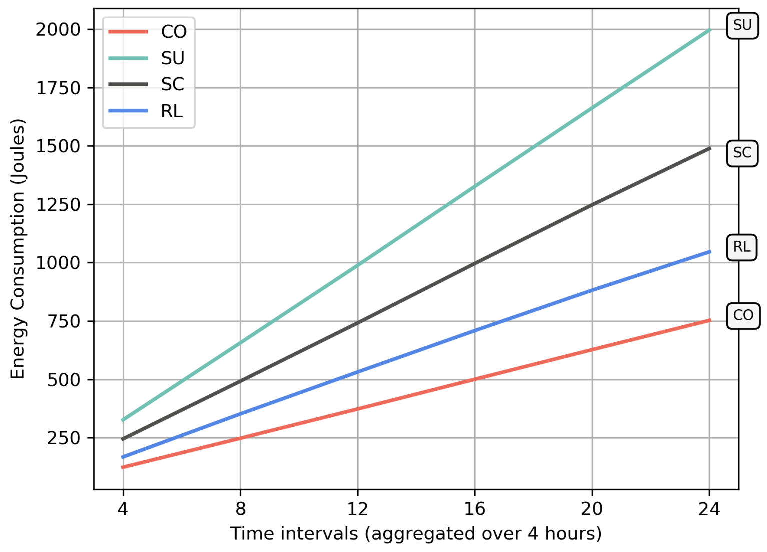


FIGURE 5.9: Bar plot of the normalized energy and data traffic consumption of different approaches

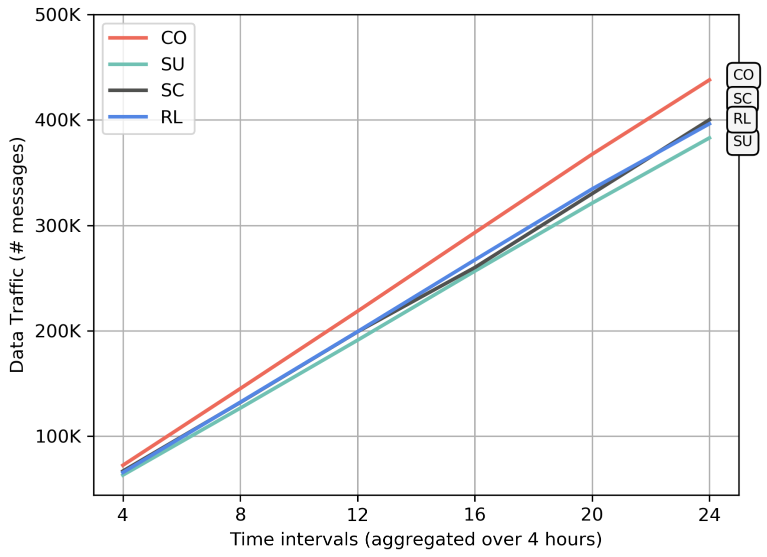
less than that of CO, and the Max DE is just 18 compared to 90 of CO. This value is a little higher than that of SU and SC, but we can see that the Min DE is much less than SU and SC. The normalized values of energy consumption and data traffic while using each of the approaches is shown in figure 5.9. We can clearly see that our approach is able to guarantee better energy efficiency without compromising on data traffic(Table 5.3). In fact, the ratio of energy saved to data traffic is almost 1 (0.66/0.7). This shows the overall effectiveness of the approach.

RQ2.3. How much does the adaptation quality improve with time?

Further, to evaluate how the approach improves as time progresses, we computed the total energy consumption of the sensors and total data traffic of the system while using the approach at the intervals of 4 hours to compute how the energy and data traffic has improved throughout the day. This was then compared with each of the individual patterns. The results are reported in Table 5.4 as well as in figures 5.10(A) and 5.10(B) in the form of cumulative energy and data traffic plots. For the first interval, the energy consumed using our approach (ER) is 167.62 Joules, and total data traffic (DR) is 65684. This data traffic value is 6789 less than that of CO (The pattern with highest data traffic), and energy is 159 Joules less than SU (the pattern with least data traffic).



(A) Cumulative plot of energy consumption



(B) Cumulative plot of data traffic

FIGURE 5.10: Cumulative plots of energy consumption and data traffic

This margin keeps improving and as time progresses we can see that the value of data traffic becomes much lesser than CO and the energy consumption becomes lesser from SU. This holds true with SC as well. The reason for this behavior can be attributed to the ability of the decision-maker to keep improving based on the feedback through rewards. The approach thus ensures that the system keeps learning and improving with time by reducing energy consumption while keeping the data traffic within the limits. For instance, at interval 8 we can observe that the value of data traffic (DR) is 132239 which is 12891 less than CO and the energy consumption (ER) is 302 Joules less than SU. This margin keeps improving and as time progresses we can see that the value of data traffic becomes much lesser than CO and the energy consumption becomes lesser from SU. This holds true with SC as well. The reason for this behavior can be attributed to the ability of the decision maker to keep improving based on the feedbacks through rewards. The approach thus ensures that the system keeps learning and improving with time by moving towards the goal state of reducing energy consumption while keeping the data traffic within the limits.

RQ2.4. How many adaptations are performed by the approach?

We evaluate the adaptation efficiency in terms of the number of adaptations by parsing the adaptation log files and extracting the information on the total number of adaptations performed for the total period of one day. We also computed the number of adaptations performed at an interval of four hours to compare how the number of adaptations has progressed with time. The results can be seen in table 5.4 (# of adaptations). The values indicate the number of adaptations performed in that corresponding 4-hour interval. For each interval, the average number of adaptation performed amounts to 13, with the total number of adaptations amounting to 79. On the other hand, every interval amounts to 240 minutes. The approach continuously monitors every 10-minute data. This implies that the approach can intelligently perform the adaptation only when required based on the forecast. This can be clearly attributed to LSTM's ability to accurately forecast the possible deviations beforehand and that of the Decision Maker component to generate the adaptation decision proactively, thus preventing the system from consuming more energy and data traffic with respect to the thresholds defined.

RQ2.5. How much time does the approach takes to perform adaptation?

To measure adaptation efficiency in terms of the time consumed per adaptation, we clocked the average time taken by the approach to generate the forecast, select a decision, and execute the adaptation. The results show that, on average, the approach takes 0.23 seconds for performing the whole process. The speed can be primarily attributed to an efficient data pipeline, which makes use of some enterprise-grade big data stack. The fact that Q-learning being a model-free technique also contribute to the speed of the approach. The majority of time is consumed by the adaptation executor, which averages around 0.13 seconds as it has to transfer the adaptation decision over the network. So, on thea whole, we can observe that the approach is able to perform adaptations in near real-time for the given study.

5.6 Discussion

In this section, we first provide details on the different lessons learned from the experiments and evaluations. Following this, we list down the possible threats to validity.

5.6.1 Lessons Learned

Accurate Forecasts of Different QoS Attributes. In the previous chapter, we saw how using the right machine learning techniques like LSTM can guarantee accurate QoS forecasts. In this chapter, we further extended the approach and applied it for forecasting data traffic as well. Our results show that high accuracy with a very less NRMSE of 0.8 and 0.9 (Table 5.2) for energy (multivariate dataset) and data traffic (univariate dataset), respectively can be obtained. They are thus ensuring that the adaptation is carried out only when necessary. Moreover, the results also indicate how the forecast accuracy (measured in this chapter using NRMSE) remained consistent across different datasets. The sudden mode changes cause an increase/decrease in the energy/data traffic values. The use of LSTM networks and periodic run of the build phase and model evaluations ensure that the model improves, and these variations are appropriately handled.

Combination Implies Better Results. Every architectural pattern has its strength and limitation. The results are visible in Table 5.3, where every pattern has some advantage for energy/data traffic and some disadvantage concerning either of these. Given the dynamic nature of the IoT system, the architecture needs to have the flexibility to intelligently switch between multiple patterns to guarantee the QoS. Our approach combines these patterns in such a way that based on the context, the pattern that offers the best energy efficiency satisfying the data traffic requirement is selected. This is further validated by the results in Table 5.3 and figure 5.9. These principles can also be extended to enable IoT devices to switch between edge, fog, and the cloud. However, this is left outside the scope of this thesis work.

Self-Adaptation to Self-Learning. Traditional approaches perform self-adaptation through *Making* and not *Achieving* thereby not improving with time [29]. This work shows that self-learning can be achieved through continuous self-adaptations using a combination of machine learning techniques. As described in Chapter 3, there have been works that exploit the use of RL techniques for performing adaptations [37, 114]. However as we have explained in Chapter 3, either they required clear knowledge of the environment or they were reactive in nature which increases the error in decisions. The result from Figure(s) 5.10 and Table 5.4) shows that as time progresses (intervals), our approach can save more energy compared to the three patterns, by still keeping the data traffic within limits. It demonstrates that with time, the systems can learn by better understanding the contexts and select better strategies, thereby improving the quality of adaptation. Further, it shows how with every adaptation, it can learn based on the obtained feedback (in the form of energy and data traffic forecasts obtained in every step) and how every learning enables the system to perform better adaptations. This is made possible by the *Decision Maker* algorithm (algorithm 2), which uses Q-learning to ensure that any wrong decision is penalized, thereby guiding the architecture to select proper decisions.

Near-Real-Time Adaptations. One of the characteristics of IoT systems is the amount of data produced and the speed at which this data is produced. With the help of enterprise-grade big data stack, we can see that our approach can analyze and perform adaptation in almost 0.23 seconds. These results indicate that the adaptation process by itself will not cause interruptions to the general functionality of the system. The use of technologies like Kafka ensures that scalability issues, latency, or data availability can be easily handled.

5.6.2 Threats to Validity

1. Internal Validity: Incorrect Forecasts/Decisions: To improve the accuracy, the build phase is executed (as mentioned in Section 5.3.2) in parallel at regular intervals. The incorrect decision issue is handled implicitly by our approach. Since the approach is proactive, the forecast in the next step will allow the algorithm to improve the decision. By keeping the reward value for incorrect decisions to be a high negative value, we ensure those wrong decisions are heavily penalized. To make the decision more accurate, the architecture can also be simulated upfront to get an initial Q-table matrix.

2. External Validity: Generalizability of the approach: Although our approach has been applied to a specific case study, the techniques used by our approach can be replicated in any case study/use-case as long as we have a way to generate the QoS data. The decision-maker uses the stakeholders' threshold limits, which can be easily modified depending on the case study and the corresponding rewards. Moreover, our approaches use generic machine-learning and techniques that do not use any specific properties or characteristics of the NdR case study. This is demonstrated later in Chapter 7 where we apply the techniques presented in this chapter to the domain of microservices.

To summarize, we demonstrated that our approach, by using a combination of proactive forecasts and RL, can enable systems to learn from every adaptation performed and vice versa. The approach therefore handled the challenge of *Systems adapting and not learning* mentioned in Section 1.2, thereby addressing *RQ2*. Such an approach improved the overall QoS of a given IoT system by handling trade-offs between different parameters, namely energy consumption and data traffic. It achieves this by providing accurate energy and data traffic forecasts and further achieves adaptation by selecting appropriate patterns using a model-free RL technique. We also demonstrated that the approach can perform effective as well as efficient adaptations. The learning is enabled by the feedbacks of the forecasts obtained in successive adaptation intervals. The approach is also made available as a tool [3].

However, we are using machine learning, in particular, Q-learning, for decision making. This implies that the decisions made, especially in the initial phase, can be erroneous due to Q-learning's exploration phase (refer Section 2.2.4). Such incorrect decisions lead to sub-optimal adaptations. Even though we handle it using proactive feedback, adaptation might still move the system to an undesirable state. To address the issue of providing qualitative guarantees to the decisions made by machine learning, we further extend the Plan activity of MAPE-K with an additional decision verifier component. It uses quantitative verification to provide guarantees to the decisions and further aids machine learning to converge towards optimal adaptations. This is further discussed in the next chapter. Further to address the issue of generalizability, in Chapter 7, we describe how the approach presented in this chapter is extended to perform service discovery in microservice-based architectures.

Chapter 6

Verification-aided Learning

In Chapter 5, we had described how a combination of Deep Learning (DL) and Reinforcement Learning (RL) techniques could are leveraged to enable the system to continuously learn and adapt, thereby allowing the system to converge towards better adaptation decisions. We further saw how it could be leveraged to improve an IoT system's energy efficiency developed for the NdR scenario, without affecting the overall performance. However, one of the issues that exist with the extensive use of these machine learning

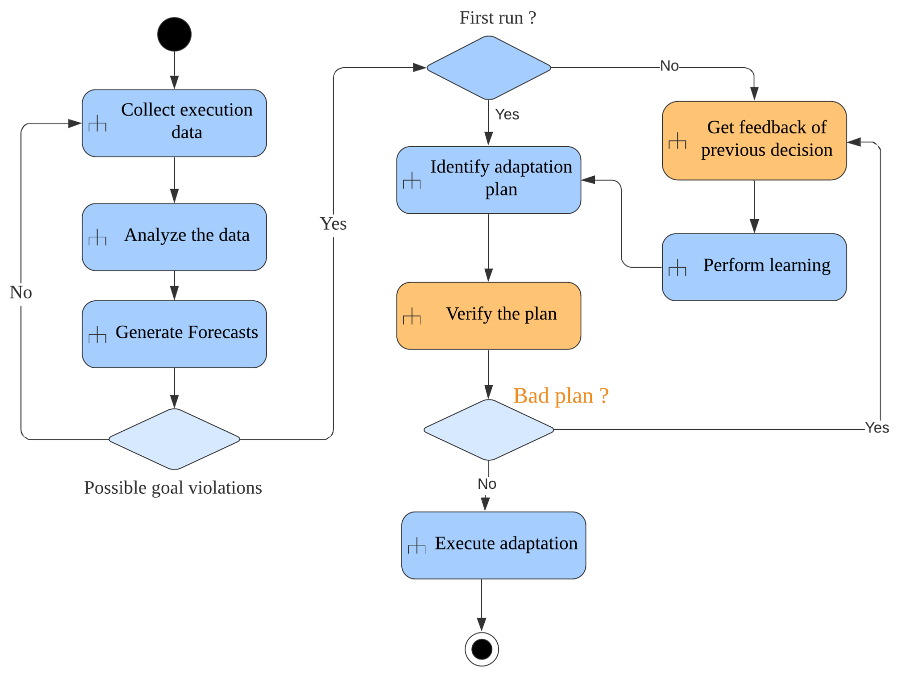


FIGURE 6.1: Verification-aided learning

(ML) techniques is that it suffers from learning bias induced from data and algorithms, which can sometimes result in bad predictions, leading to sub-optimal, or even infeasible adaptations [40] [41, 176]. For instance, as we already discussed in Chapter 5, we can use RL techniques to select the best pattern based on the execution scenario. But one of the issues with RL is that it has to make many mistakes (wrong decisions) before it can start selecting the right choices (refer Section 2.2.4). Although we handle this issue implicitly, as mentioned in (Section 5.6.2), such wrong selection of pattern might lead to sub-optimal adaptations, thereby affecting the overall QoS of the given system.

This is a problem where the use of Quantitative Verification (QV) techniques (and in particular, probabilistic model-checking [76]) can improve on the current situation, *complementing* the strengths of ML. Figure 6.1 shows the overall flow of such a process where quantitative verification can add as an aid and support to the machine learning decisions. As seen from the figure, there is a new activity, differently to learning-driven adaptation flow presented in Chapter 5 (Figure 5.1). This additional activity, *verify the plan*(highlighted in Figure 6.1), verifies the decision on receiving an adaptation plan. If the verification fails, it sents feedback that is further used by the overall approach to learn and generate a new adaptation plan. Instead, if the verification is successful, then the adaptation plan is executed. In this manner, given a decision and a formal description of the system's current configuration and its execution conditions, QV can provide feedback concerning the feasibility of the decisions selected by ML. This feedback, in the form of quantitative guarantees about expected QoS levels, is used to (i) prevent the execution of infeasible solutions selected by ML, and (ii) improve the quality of future decisions made by ML.

Although there are existing self-adaptation approaches that combine the use of ML and QV [120, 121], these use ML simply as a means to narrow down the decision space, with information flowing only from ML to QV, and having QV in charge of decision-making. In contrast, we propose a novel way of combining QV and ML. In this, QV is used to *verify* the decisions made by ML and *provide feedbacks* to ML, helping it to achieve *faster convergence towards optimal decisions* without narrowing down the solution space, opening up the possibility of finding better solutions than using either QV or ML in isolation.

Towards this direction, in this chapter, we extend the MAPE-K based approach presented in Chapter 5 to develop an approach for architecting self-adaptive IoT systems which use the learning ability of ML and the verification capabilities of QV to identify and enact optimal adaptations proactively. It thereby solves the *RQ3. How to guarantee the quality of the adaptation performed by such a machine learning process, and how can such guarantees, in turn, help the machine learning process?*

We further apply the approach to the IoT system developed for the NdR case study, where ML is tasked with selecting the best adaptation pattern for a given scenario. QV checks the feasibility of the adaptation decision, preventing the execution of unfeasible adaptations and providing feedback to RL, which helps to achieve faster convergence towards optimal decisions. Given a scenario and set of QoS constraints, the approach offers the following advantages:

1. It uses RL techniques to select an adaptation pattern and uses feedback to improve the decision making process continuously.

2. It verifies the feasibility of decisions made by the RL with respect to the QoS requirements using QV, in particular, probabilistic model checking.

3. Executes the adaptation if the decisions are feasible and otherwise requests a new decision from RL.

4. QV provides feedback to RL for improving the quality of future decisions. In this manner, RL obtains feedbacks from both QV as well as QoS forecasts to converge faster towards optimal adaptations.

6.1 Motivating Scenario

As a motivating scenario, let us consider the NdR case study, which we had presented in Section 2.5. The two main concerns we have in the NdR, as we have seen in Chapter 5, are related to ensuring that the overall system operates efficiently with respect to energy and overall data traffic requirements. We also saw how we could use a combination of ML techniques to ensure that the overall goals are satisfied. However, due to the bias and accuracy issues, the decision selected by the ML might not be optimal. For example, given a context that ML has predicted, there will be a sudden increase in data traffic. Let us consider that the three patterns, SU, SC, CO are available for performing the adaptation. Based on this, RL might select SU as the pattern to be used as it has mostly selected SU in such contexts in the past. However, this might eventually turn out to be a bad decision with the system consuming more traffic because some sensors had to operate in critical mode (which means there will be more communication between the sensors to ensure synchronization). This can then impact the reliability of the system. This is because RL or, in general, ML has no ways to guarantee with full confidence that SU is the best pattern for the current situation. Hence, we need QV to ensure if SU is the optimal pattern given the system execution context. Therefore given the energy and data traffic constraints, the system should have the ability to decide the best adaptation pattern and further provide a guarantee on the decision made. This can be formally defined as follows:

Given: (i) data traffic constraints D_{max} and D_{min}, i.e., the maximum and minimum data traffic levels acceptable for a given execution period of duration τ, and (ii) E_{max} being the maximum energy that can be consumed for that same τ-period, the goal of the system is maximizing the following utility function that captures the non-functional requirements of our scenario and enables us to quantify their satisfaction:

$$U_\tau = w_e \cdot E_\tau + w_d \cdot T_\tau, \text{ with}$$

$$T_\tau = \begin{cases} (D_{max} - d_\tau) \cdot p_{dv} & \text{if } d_\tau \geq D_{max} \\ d_\tau - D_{min} & \text{if } D_{max} > d_\tau > D_{min} \\ (d_\tau - D_{min}) \cdot p_{dv} & \text{if } d_\tau \leq D_{min} \end{cases}$$

$$E_\tau = \begin{cases} E_{max} - e_\tau & \text{if } e_\tau < E_{max} \\ (E_{max} - e_\tau) \cdot p_{ev} & \text{otherwise} \end{cases}$$

where, e_τ, d_τ represent the total energy and data traffic consumed by the system for the τ-period, and w_e, $w_d \in \mathbb{R}^+$ are weights that capture the priority of energy and data traffic savings, respectively. T_τ and E_τ are piece-wise functions that capture the data traffic and energy savings respectively where, p_{ev}, $p_{dv} \in \mathbb{R}^+$ represent penalties for the violations of energy and data traffic thresholds, respectively.

6.2 Approach Overview

In this section, we provide an overview of the approach by revisiting some of the concepts in the previous chapters and describing how it combines ML and QV techniques to converge towards optimal adaptations.

The approach builds on the MAPE-K pattern, as stated in the previous chapters, with some additions on each activities keeping more emphasis on the *Plan* and *Execute* activities. It instantiates its different components in the following way (Figure 6.2):

The *Monitor* activity regularly collects one additional types of data from the IoT system at run time, in addition to the QoS metrics (e.g., instantaneous traffic, energy consumption) namely, the data collected by sensors. These are continually sent to the *Analyze* activity and the *MLE*. The *MLE* is responsible for building *two types of ML models* using Long Short-Term Memory (LSTM) Networks: (i) QoS models for forecasting the expected energy consumption and data traffic up to the duration of the planning horizon, and (ii) forecast models predicting the expected behaviour of every sensor component up until the planning horizon. The *MLE* periodically updates these trained models in the *Knowledge Base*.

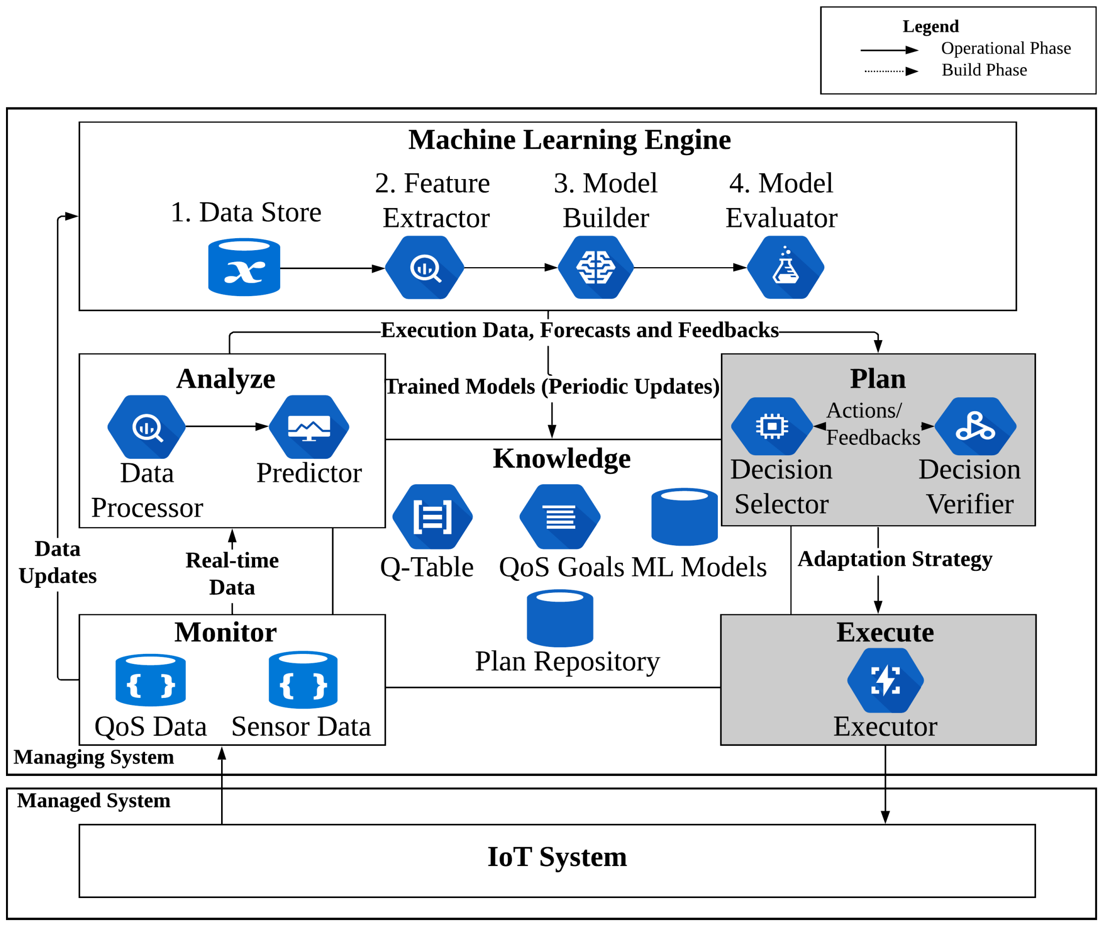


FIGURE 6.2: Approach Overview

The *Analyze* activity is responsible for identifying the need for adaptation based on the data obtained. It first processes the data as required by the *Predictor* component using the *Data Processor* component.

The *Predictor*[1] component uses the processed data to predict the expected QoS of the system and behaviour of sensors (operational modes) using forecast models from the *Knowledge Base*. These forecasts are used to identify the need for adaptation. If adaptation is needed, the forecasts are made available to the *Plan* activity. The *Plan* activity mainly consists of two components, (i) *Decision Selector* and (ii) *Decision Verifier*.

The *Decision Selector* selects the best architectural pattern (from those available in the *Knowledge Base*) that can be used to perform an adaptation based on the forecasts received. It performs this selection using RL. This selection is then fed to the *Decision Verifier* for further verification. This is to ensure that only decisions that are feasible with respect to QoS goals defined in the *Knowledge Base* are executed. In case of an infeasible decision, the decision verifier component sends negative feedback to the decision

[1]This component is different from the QoS predictor presented in previous chapter, Figure 5.3, as it also performs behavioral forecasts

selector and requests a new decision. If the decision produced by the decision selector is feasible, then the decision verifier sends positive feedback to the decision selector and sends the decision to the *Execute* activity. The *Decision Selector* collects feedback from two sources for continuous improvement: (i) the verification results obtained as the immediate feedback for the decision made, and (ii) for every decision that was sent to the *Execute* activity, it uses the QoS forecast received in the next iteration of decision making as an additional source for feedback.

The *Execute* activity is responsible for enacting the selected adaptation obtained from the *Plan* activity via effectors embedded at the system level that enable architectural change in the IoT system.

The *Knowledge Base* acts as central storage for different types of knowledge required by different layers for performing the adaptations. It stores four types of information: (i) *Q-Table*, which is a lookup table used by the Decision Selector to select the best pattern for adaptation, (ii) *QoS goals*, which capture the acceptable energy and data traffic thresholds as defined by stakeholders such as hardware and network engineers, (iii) *Model Repository*, which contains the updated ML models for forecasting the QoS and expected operational modes of sensor components. These models are further used by the *Predictor* component of the *Analyze* activity, and (iv) *Pattern Repository*, which contains the set of adaptation patterns available for performing the adaptation and their definitions.

Our instantiation of the *Monitor*, *Analyze*, and the decision making part of *Plan* activities, along with the *Machine Learning Engine* (MLE) have been presented in previous chapters. In this chapter, we focus further on the *Plan* and *Execute* activities (enclosed by grey boxes in Figure 6.2). In particular, we focus on how RL (*Decision Selector*) and formal verification (*Decision Verifier*) via model checking can complement each other to guide the system towards selecting architectural adaptations that optimize the guarantees of satisfying acceptable QoS levels.

6.3 Verification-Aided Machine Learning Approach

This section describes how the *Plan* activity uses RL to select the best adaptation pattern and further, how it uses QV, in particular, model checking as a guide to continually improve the decision making process using feedback obtained from verification. The decision making process, which involves six components, is shown in Figure 6.3. In the remainder of this section, we first introduce some preliminary definitions used in the approach, and then we explain each of the components and their interaction in detail.

6.3.1 Preliminaries

The overall goal of the approach is to ensure that the QoS requirements are satisfied throughout the execution of the system. These are formalized in QoS goals:

Definition 6.1 (QoS Goal)**.** We define a QoS goal as a pair (μ, u), where $\mu \in M$ is a unique label identifier for a QoS metric, and $u = (u_l, u_h) \in \mathbb{R}^2$ is a pair of threshold values.

Example 6.1. *For the scenario of the NdR case study (refer Section 2.5) mentioned in Section 6.1, we define two QoS goals:*

1. Energy consumption, $g_e = (energy, (e_l, e_h))$, states that the accrued energy consumed by the system should not exceed a maximum value e_h. In this goal, e_l represents the threshold below which the sensors save the maximum energy.

2. Data traffic, $g_d = (traffic, (d_l, d_h))$ states that the maximum total traffic allowed in the system should not exceed d_h, and should stay above d_l, which is the minimum traffic to be satisfied by the system to be able to maintain an acceptable service accuracy.

To achieve these QoS goals, our approach works by periodically generating an adaptation decision that considers predictions (forecasts) about the behaviour of the system over a time horizon of duration $H \in \mathbb{R}^+$. We assume a decision period of duration $\tau \in \mathbb{R}^+$, and consider H to be a multiple of τ. Hence, for a decision generated for time instant t: (i) the forecasts employed as model of the environment considered for the decision cover the period $[t, t + H]$, (ii) the decision made is executed (i.e., change of pattern, if the selected one is different from the current pattern active in the system), and (iii) the pattern selected is maintained as active for the time interval $[t, t + \tau]$. At that point $(t + \tau)$, a new decision is generated for the period $[t + \tau, t + 2 \cdot \tau]$ that considers forecasts for the period $[t + \tau, t + \tau + H]$. This sequence is continually repeated each τ-period.

Generating decisions employ two types of forecasts over the lookup horizon: (i) QoS forecasts, by leveraging the QoS data and (ii) behaviour forecasts, by leveraging the context data, which capture the evolution of the relevant system and environment variables.

Definition 6.2. A *QoS Forecast* a pair $(\mu, u) \in M \times \mathbb{R}^+$, where μ is a unique label identifier for a QoS metric, and u is its predicted value of the metric accrued over the duration of the lookup horizon.

Definition 6.3. A *Behaviour Forecast* is a sequence $\langle f_1, \ldots, f_k \rangle$ where each element $f_{i \in 1 \ldots k}$ is a tuple $(v, l, u) \in V \times \mathbb{R}^2 \times \mathcal{D}(v)$, where: v is a unique identifier for a system/environment variable, and l is a time interval during which v takes the value u.

$\mathcal{D}(v)$ denotes the domain of v. We assume that the time intervals of the elements in f_i fully cover up to the duration of the lookup horizon H.

Example 6.2. *Consider a horizon $H = 600$ seconds in our system. A sensor s may operate in a mode captured by variable $s.mode$, which takes values in the domain $\{normal, critical\}$. A behaviour forecast for the mode in which sensor is operating during the time interval $[t, t+600]$ like $\langle(s.mode, [0,100], normal), (s.mode, [101,600], critical)\rangle$ captures that sensor s is expected to operate in normal mode for the next 100 seconds, and in critical mode during the remainder of the time, up until the end of the horizon.*

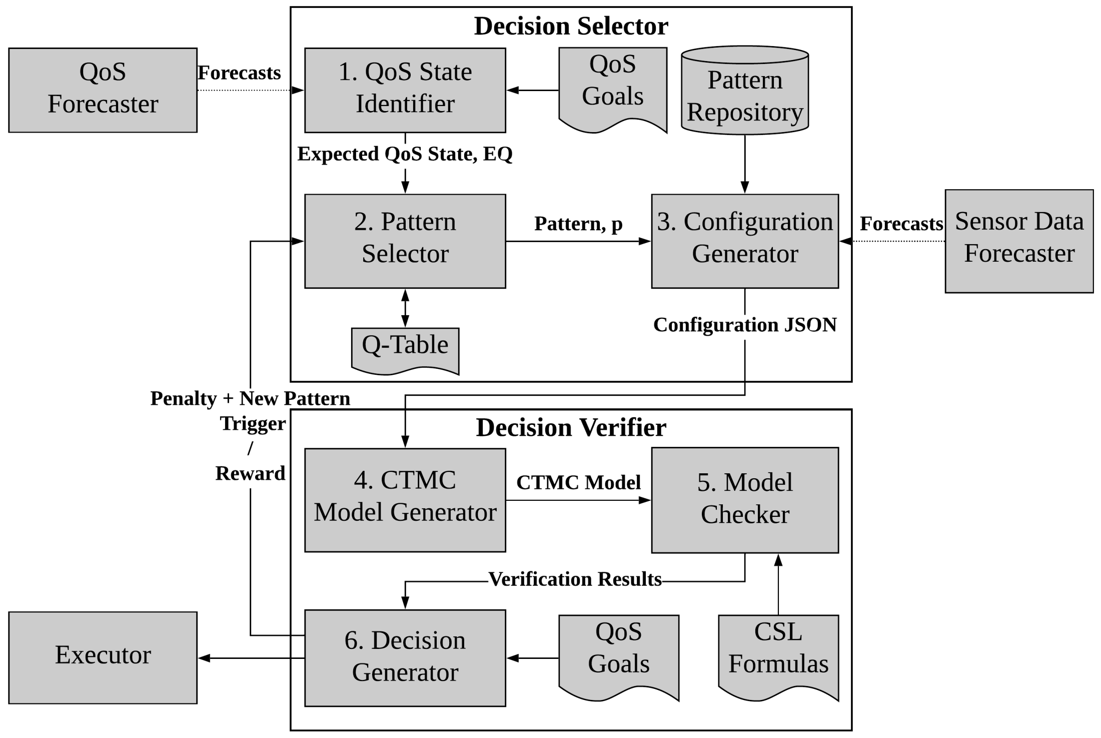


FIGURE 6.3: Detailed View of Decision Making Process

6.3.2 QoS State Identifier

Our approach assumes a discrete set of QoS categories for each dimension of concern (e.g., energy consumption, traffic).[2] The first part of decision making is identifying the expected QoS state of the system EQ for the next τ-period, i.e., the set of expected QoS categories for every dimension of concern (Component 1, Figure 6.3).

For each decision period elapsed, the *QoS State Identifier* receives the set of QoS forecasts QF for the next decision period from the *QoS Forecaster* of the *Analyze* activity. Then, QF is mapped to a set of categories EQ that identifies the QoS state of the system

[2]We denote the set of all QoS categories across dimensions by QC.

for the next decision period. The mapping between QF and EQ is obtained based on the values in QF, and how they meet the thresholds stated in QoS goals.

For instance, let the QoS goal for energy consumption be $(energy, (0.5, 2))$, meaning that the energy consumption of the system up until the time horizon should stay in the range 0.5-2 Joules. We can define a mapping function $qc : \mathbb{R}^2 \rightarrow \{low, medium, high\}$ as $\{[0, 0.5] \mapsto low, [0.5, 2] \mapsto medium, [2, \infty] \mapsto high\}$. To identify the QoS state of the system EQ, this mapping process is repeated across all quality dimensions of concern for every element of QF. Once identified, EQ is provided as input to the *Pattern Selector* (Component 2, Figure 6.3).

6.3.3 Pattern Selector

The role of the *Pattern Selector* is to select the best adaptation pattern that can be applied based on the expected QoS state of the system EQ. It uses, Q-learning [72] as described in Chapter 5, to decide on the adaptation pattern. However, differently from the algorithm 2 presented in Chapter 5, Pattern selector uses two-way feedback, both from the expected QoS forecasts as well as from the results of the verification process.

The state space of our Q-learning algorithm $S \subseteq 2^{QC} \times P$ is defined over the set of possible QoS states, and the set of available patterns $P = \{1 \ldots m\}$. The algorithm also assumes a set of actions, $A = \{a_1, a_2..a_m\}$ that correspond to reconfigurations to patterns in P, and a reward function $\rho : S \rightarrow \mathbb{Z}$ that maps states to an integer reward assigned for moving into a state via a pattern change action.

The algorithm makes use of a lookup Q-table matrix that can be encoded as a function $Q : S \times A \rightarrow \mathbb{R}$, that returns a real number (Q-Value) for any arbitrary state-action pair (s, a). This value gives an estimation of how valuable it is to select the action a from state s.

Pattern selection is presented in Algorithm 3. It takes as inputs the states, actions, expected QoS state and current pattern (S, A, EQ and p). Further inputs are reward function ρ, as well as parameters α and γ where, $0 \leq \alpha \leq 1$ represents the learning rate which captures the importance given to the learned observation at each step, and $0 \leq \gamma \leq 1$ represents the discount factor, which can be considered as the weight given to the next action. It additionally takes two inputs, a verification parameter, V and verification reward, r_m. V takes only Boolean values: 0 denotes the invocation of the algorithm for a new pattern by the *QoS State Identifier*, 1 indicates the trigger for a new pattern from the verification process. On the other hand, r_m represents the reward($r_m > 0$)/penalty ($r_m < 0$) obtained after the verification process from the

Decision Verifier. It goes without saying that, r_m is considered only when the value of V is 1. These parameter are used by the algorithm to decide the reward distribution.

Algorithm 3 Pattern Selection Algorithm

```
1: procedure PATTERN-SELECTOR($S$,$A$,$EQ$,$p$,$\rho$,$\alpha$, $\gamma$,$V$,$r_m$)    ▷ States, Actions, QoS state, active pattern, learning rate, discount rate, verification flag and verification reward
2:     $s^t \leftarrow (EQ, p)$
3:     $a^t \leftarrow a_p$
4:     if $V = 0$ then
5:         $r^t \leftarrow \rho(s^t)$
6:     else
7:         $r^t \leftarrow r_m$
8:     $(s^{t+1}, a) \leftarrow argmax_{(s,a)\in S\times A} Q(s, a)$
9:     $Q'(s^t, a^t) = (1-\alpha) \cdot Q(s^t, a^t) + \alpha \cdot (r^t + \gamma \cdot Q(s^{t+1}, a))$
10:    if $r_m > 0$ then    ▷ Successful verification
11:        return 1
12:    else
13:        return $a$
```

Before learning begins, Q (function that encodes Q-table) is initialized to an arbitrary fixed value. The Expected QoS state, EQ along with the current execution pattern p, becomes the current state s^t and the action corresponding to the pattern p (denoted by a_p) is assigned to the current action a^t (lines 2 and 3). At each time t the agent selects an action a^t, enters a new state s^{t+1}(that depends on the selected action – line 5), and observes a reward r^t based on V, If V is 1, then it denotes that the verification was either successful/failure and based on this a reward/penalty r_m is assigned to r^t (lines 4-7). Following this, Q is updated (line 5-6). The core of the algorithm, as explained in Chapter 5, is a simple value iteration update, using the weighted average of the old value and the new information (controlled by the α and γ parameters, respectively). The algorithm then checks the value of r_m. If it is greater than 0, it just returns a flag value indicating the update of the Q-table (lines 10-11). Instead, if the value of $r_m \leq 0$, then the algorithm returns the action a, which corresponds to the selection of the new pattern. This action maximizes the Q-value that can be obtained by performing the different actions available for the next state s^{t+1} (line 8), ensuring that for every time instant, the best pattern for adaptation according to Q is selected.

The pattern selected by the algorithm is sent to the *Model checker* for verification through the *Configuration Generator*.

6.3.4 Configuration Generator

The Configuration Generator is responsible for creating the configuration of the selected pattern at run time as required by the *CTMC Model Generator*. It uses the definitions of the patterns from the *Pattern Repository* along with the run-time execution data from the *Analyze* activity and the forecasts on the expected behaviour of sensors from the *Sensor Data Forecaster* to build the description of the different configurations.

The *Pattern Repository* contains the static information on the configuration of each adaptation pattern, which consists of a set of components involved C, and the set of connectors that exist between the components, $K \subseteq C \times C$.

Sensor components in C are annotated by properties such as the idle energy consumption and frequency of data transfers in different execution modes (for example, during critical situations such as emergency, sensors might communicate data more frequently, compared to a normal scenario).

Each connector $(c, c') \in K$ is annotated by properties such as the energy consumed for sending data from c to c', energy consumed by c for processing the data to be sent, and the energy consumed by c' for receiving the data from c.

Based on the pattern p selected by the *Pattern Selector*, the static configuration of the corresponding pattern is retrieved from the pattern repository and annotated with run-time information about sensor components, which includes the *Sensor Data Forecasts* obtained from the *Predictor* component of the *Analyze* activity. The *Predictor* uses the same principles of LSTM networks used to generate QoS forecasts (as explained in Chapters 4 and 5) for producing the forecasts of the expected sensor data. For example, with respect to the NdR scenario, these forecasts consist of the expected number of cars in the parking lot, the expected number of people entering a given venue, etc. for every minute of a given horizon. The forecast-annotated version of the configuration is then passed on to the CTMC Model Generator.

6.3.5 CTMC Model Generator

The CTMC model generator takes as input the configuration description obtained from the configuration generator, and produces a CTMC model (refer Section 2.3) analyzable via model checking. Concretely, the generator takes the specification of the set of components in the configuration, and for each one of them, it instantiates a process description using the different patterns shown in Figure 6.4. The mapping between component types in the class of IoT system we describe, and process types used in the CTMC models is shown in Table 6.1.

TABLE 6.1: Component-process type instantiation for patterns

Pattern / Comp.	**Sensor**	**Database**	**Controller**	**Display**
SU	Produce-Forward	Consume	- (in-sensor)	Consume
SC	Produce	Forward	Consume	Consume
CO	Produce	Consume	Forward	Consume

The *Produce/Produce-Forward* process type captures the behaviour of sensors, which can be in normal and critical mode (states N and C, respectively). Each one of the modes produces and sends sensor data at different rates (λ_n and λ_c) via action msg. Mode changes can be triggered by an exogenous event mode_chg (modeling, e.g., a variation in rate of cars entering a parking lot). In the SU pattern, this process type includes the additional states and transitions required to forward data received from other sensors (shown in dashed lines in Figure 6.4). In each one of the modes, the process can receive a message from any component it is connected to in the configuration (brackets denote multiple transitions), going into the states (N',C'), from which the message can be forwarded immediately (λ_i is a constant that denotes an instantaneous transition rate). Transitions that do not show a rate have a default value of 1, meaning that they are only triggered via synchronization on the events they are labeled with. Concerning reward structures, sending and receiving messages from node i accrues an energy cost of e_s^i and e_r^i units in reward structure energy, respectively, and 1 message for traffic in reward structure messages.

The *Consume* process type is used to capture data sink nodes in the system (e.g., displays, the database in the CO and SU patterns). It just receives messages from any node in the network it is connected to, consuming energy e_r^i.

Finally, the *Forward* process type captures the behaviour of intermediate nodes in the network, which can receive messages from multiple nodes, and forward them to other nodes (e.g., controller in the CO pattern, database in the SC pattern).

The overall CTMC model results from the standard CSP parallel composition of all the processes instantiated, which synchronize on shared labels generated from the connections included in the configuration description.

6.3.6 Model Checker

The model checking component takes as input the CTMC model produced by the *CTMC model generator*, and is able to quantify the expected amount of energy consumed, as well the overall number of messages exchanged in the system for the time frame of the lookup horizon H. Quantification of each one of these properties is achieved via

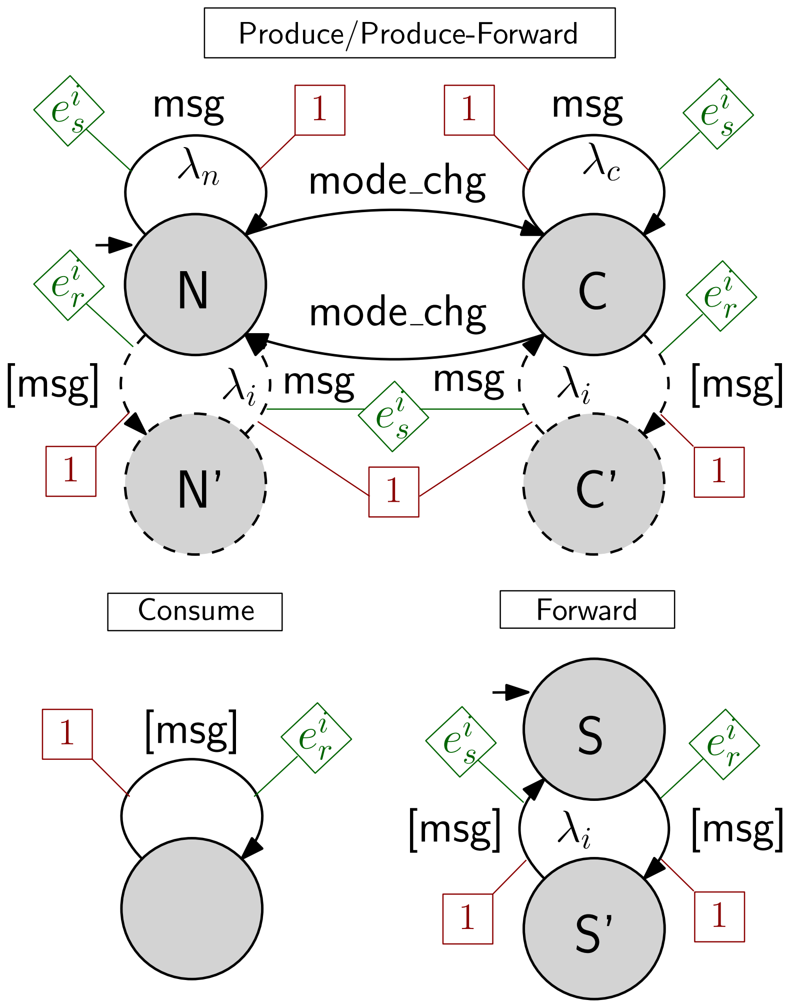


FIGURE 6.4: CTMC process types for IoT nodes

TABLE 6.2: CSL properties for model checking

Name	CSL Formula	Description
ϕ_e	$\mathsf{R}^{\mathsf{energy}}_{=?}[\mathsf{C}{<=}\mathsf{H}]$	**Efficiency**: overall energy units consumed during H time units.
ϕ_t	$\mathsf{R}^{\mathsf{messages}}_{=?}[\mathsf{C}{<=}\mathsf{H}]$	**Traffic**: overall number of messages exchanged during H time units.

model checking of the CSL properties shown in Table 6.2, for which the component uses PRISM's model checking engine [177].

6.3.7 Decision Generator

The *Decision Generator* is responsible for deciding the feasibility of the decision produced by the *Pattern Selector* based on the results of the analysis obtained from the *Model*

Checker. The model checker's output consists of the expected energy consumption and data traffic of the system while using the selected pattern. The decision generator verifies if these values are within the thresholds specified in *QoS goals*. If the expected energy consumption is less than the threshold, e_h as defined in QoS goal g_e and the expected data traffic is within the thresholds as defined in QoS goal g_d. The decision is considered as a feasible (valid) one, and positive feedback in the form of a reward, $r_m > 0$, is fed sent to the *Pattern Selector*. Otherwise, the decision is considered as infeasible (invalid), and negative feedback in the form of penalty $r_m < 0$ is sent back to the *Pattern Selector*, along with the request of for a different pattern.

Hence, for every decision made by the *Pattern Selector*, a verification step is performed by the *Model Checker*, improving the chances that only feasible adaptations are executed. Given an adaptation scenario, this combination provides multiple advantages: (i) it helps the system to select decisions based on past feedback obtained from the model checker as well as from the previous forecasts, (ii) it ensures that, even if RL generates a wrong decision, it does not affect the system execution due to the involvement of the model checker, and (iii) it ensures that the model checker does not have to perform exploration of a broader solution space because it just has to analyze system behaviour under the selected pattern.

6.4 Architecture and Implementation

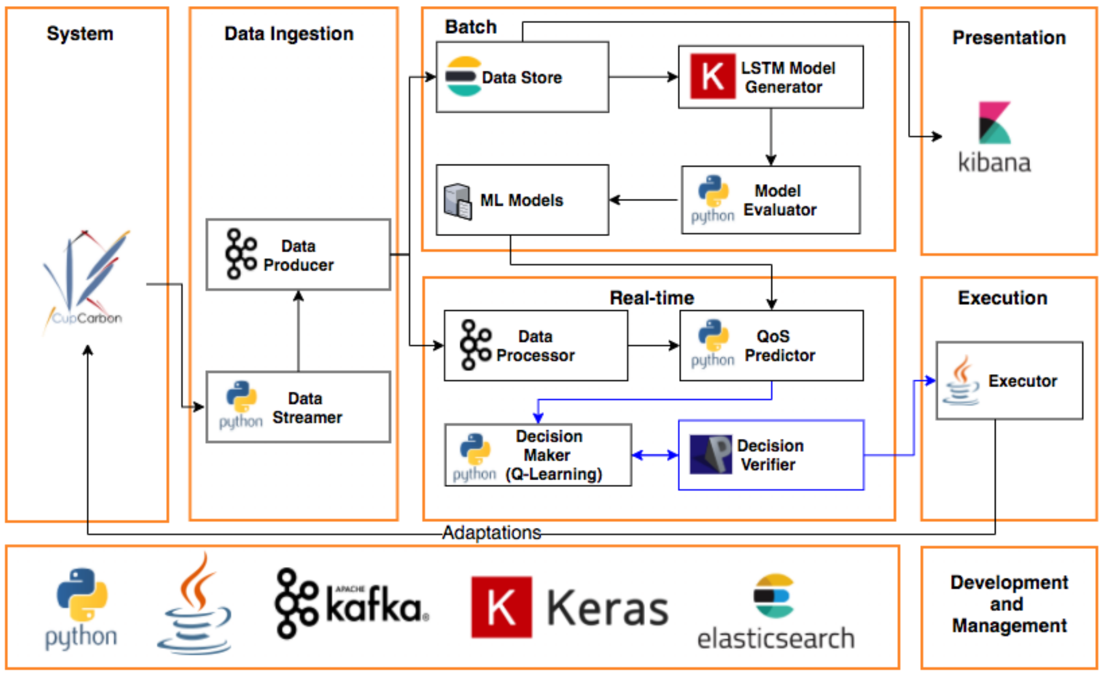


FIGURE 6.5: Implementation Pipeline View

Figure 6.5 shows the implementation view of our approach. It is very similar to the one presented in Chapter 4 with few modification and additions in the different layers (highlighted in blue in Figure 6.5). We consider the NdR scenario for the implementation. During the simulation using CupCarbon, in addition to the energy and data traffic logs, the real-time sensor data are also ingested into an Apache Kafka [158] broker. For generating the forecast models, we employed Python, along with the deep learning library Keras [162]. The *Pattern Selector* implementing Q-learning was written in Python. The *Model Checker* component in the *Decision Verifier* is written in Java and makes use of the PRISM model checker [177] API. The integration between the *Decision Maker* and the *Decision Verifier* was done using JPype [178]. Additionally, a web service implemented in Python using the Tornado framework [179] is used for communicating the pattern change to CupCarbon. The communication between CupCarbon and the different MAPE-K activities is powered by Apache Kafka [158]. For more details on the used technologies, we refer the reader to the Appendix A.

6.5 Experimentation and Evaluation

The objective of our evaluation is to assess the effectiveness and efficiency of the approach by answering:

RQ3.1. How effective is the approach with respect to satisfying overall energy and data traffic goals?

RQ3.2. How much does using model checking along with ML improve satisfaction of goals over the use of just ML?

RQ3.3. What is the efficiency of adaptations?

RQ3.4. What is the computation overhead of adaptation?

In the remainder of this section, we first describe our experimental setup, as well as the data and metrics used for the evaluation of the approach, following with a discussion of the evaluation questions informed by our results.

6.5.1 Experimental Setup

Our testbed was deployed on two VM instances in Google Cloud. The first one runs on an N1-Standard-4 CPU Intel Skylake Processor comprising 4 vCPU with 16 GB RAM. This instance was used for running the CupCarbon simulation and the producers for sending the QoS metrics and Sensor data to the Kafka broker. The second one runs on an N1-Standard-8 CPU with Intel Skylake Processor comprising 8 vCPU with 32 GB RAM. This was used for running the Kafka broker and the MAPE activities of our approach.

To simulate the real scenario of the case study with as much fidelity as possible, we created a script that generates data for each of the sensor components using intervals of 60 seconds with arrival rates based on a Poisson distribution for a period of 24 hours. The mean values of the distribution were selected based on observations from real NdR scenarios.

6.5.2 Evaluation Candidates

We evaluated the approach by performing the simulation of the case study using six different approaches for a period of 24 hours. Three of the approaches consisted of simulating the case study, fixing each of the patterns (SU, SC and CO). The other three approaches are as follows:

1. RL: Adaptation using just Q-learning as described in Chapter 5.

2. MC: Approach which performs adaptation just based on model checking, but without using Q-learning. For every decision period, it performs model checking to identify the best pattern by finding which pattern gives the maximum benefit for the thresholds specified.

3. RLMC: Our approach, which performs adaptation combining Q-learning and model checking.

Considering the operational constraints we have from the case study, we defined the set of QoS goals, $QG = \{(energy, (10.0, 6.0)), (traffic, (2500, 3000))\}$, with energy measured in Joules and traffic in # of messages exchanged. We consider a horizon $H = 600$ seconds and $\tau = 60$ seconds. We use a learning rate $\alpha = 0.02$ and discount factor $\gamma = 0.2$ for performing the Q-learning. The complete implementation along with the source code, datasets and ML models used for forecasts can be found here.[3]

6.5.3 Evaluation Metrics

To measure the effectiveness of the approach, we introduce four evaluation metrics: (i) Max and Min DV, which capture the number of violations of maximum (d_h) and minimum (d_l) data traffic thresholds as defined in g_d (c.f. Example I), (ii) EV, capturing

[3]https://github.com/karthikv1392/IoT_RLMC/

Approach	Energy (Joules)	Data Traffic (# messages)	Utility Score	# Max DV	# Min DV	# EV
CO	752.43	437885	186.89	90	0	0
SU	1995.85	382896	61.11	5	20	143
SC	1498.57	398393	158.11	14	8	104
RLMC	851.35	400239	355.24	17	6	2
MC	1246.43	407107	218.43	26	3	25
RL	1051.38	398294	272.98	18	7	8

Table 6.3: Energy Consumption and Data traffic comparison (#DV : Number of Data traffic Violations, #EV : Number of Energy Violations)

the number of violations of the maximum energy threshold (e_h) as defined in g_e, and (iii) Utility Score (U) as defined in Section 6.1, using normalized values for energy consumed e_τ and data traffic d_τ for every τ-period. We set $w_e = 2$, $w_d = 5$, $p_{ev} = 0.3$, and $p_{dv} = 0.5$. We assign a slightly higher weight to the traffic term (and also higher value to its penalty) because, although saving energy is a priority, we do not want to do it at the expense of a system that does not operate with the required accuracy.

6.5.4 Results

RQ3.1. *How effective is the approach with respect to satisfying overall energy and data traffic goals?*

To measure effectiveness, we calculate the total energy and data traffic consumed during simulation by each of the approaches. The aggregated results for our evaluation metrics are reported in Table 6.3. The table shows that CO consumes the least energy, but at the same time, it maximizes data traffic. This is due to the semi-decentralized nature of the pattern which results in an increased amount of exchanged messages, resulting from the presence of extra controller components in the architecture. SU is the pattern that presents the lowest traffic volume, although to a level that is detrimental to maintain service accuracy (presents more than double Min DV count, compared to other approaches). SU is also the least energy-efficient. In contrast, SC is more energy-efficient than SU, but presents a higher data traffic volume with a high count of Max DV. This is due to the fact that every decision has to be taken by the centralized controller and hence the sensors need to send information to the database at a faster rate compared to SU, where sensors are equipped with decision making abilities. MC consumes less energy than SU and SC, also with lower traffic than CO. It offers better utility compared to fixed patterns because it can choose what is determined to be more adequate for different decision episodes. However, RL offers much better Utility and consumes less energy with lower traffic compared to MC. This can be attributed to the ability of

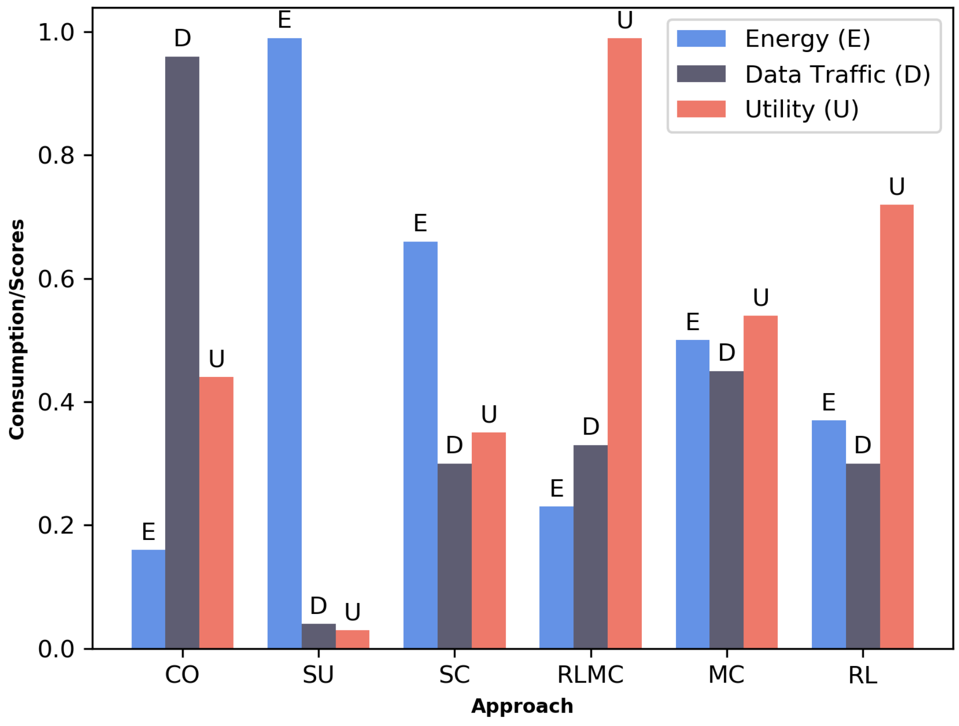


FIGURE 6.6: A bar plot of the different approaches and their respective normalized energy and traffic consumption along with overall utility scores

RL to learn from feedback over time. Furthermore, RLMC is the most energy-efficient, compared to RL and MC and only at a slightly higher traffic volume than RL. This yields an increment in utility of 39% and 63% with respect to RL and MC, respectively. This clearly shows the remarkable impact that QV has on RL for decision making. The normalized values of energy, traffic and utility are shown in Figure 6.6. RLMC scores the highest utility, being second to CO in energy efficiency. Although RL has lower data traffic than RLMC, the ratio of energy saved/traffic saved for RLMC ($0.78/0.62 = 1.26$) is higher than that of RL ($0.62/0.75 = 0.83$).

RQ3.2. *How much does using model checking along with ML improve satisfaction of goals over the use of just ML?*

To answer this question, we compare the cumulative utility score of all approaches (Figure 6.7). The plot shows how accrued utility starts diverging marginally during the initial stage, but then the gap between RL/RLMC and other approaches keeps on increasing. MC still offers better performance compared to each of the fixed patterns because it selects what it expects to give the best utility at the start of every decision period. However, it does not have a way to improve the decision in the next iteration based on the feedback of the past decision. In contrast, both RL and RLMC have the advantage of feedback which allows them to progressively improve their decisions. Unlike in RLMC, the feedback in RL is obtained only after execution of the selected adaptation, which might end up being sub-optimal. The effect of these sub-optimal decisions can be clearly observed in the graph, where RLMC offers initially an increment of just 1% in utility over RL, but as time progresses, this value increases up to 39% over a span of 24 hours.

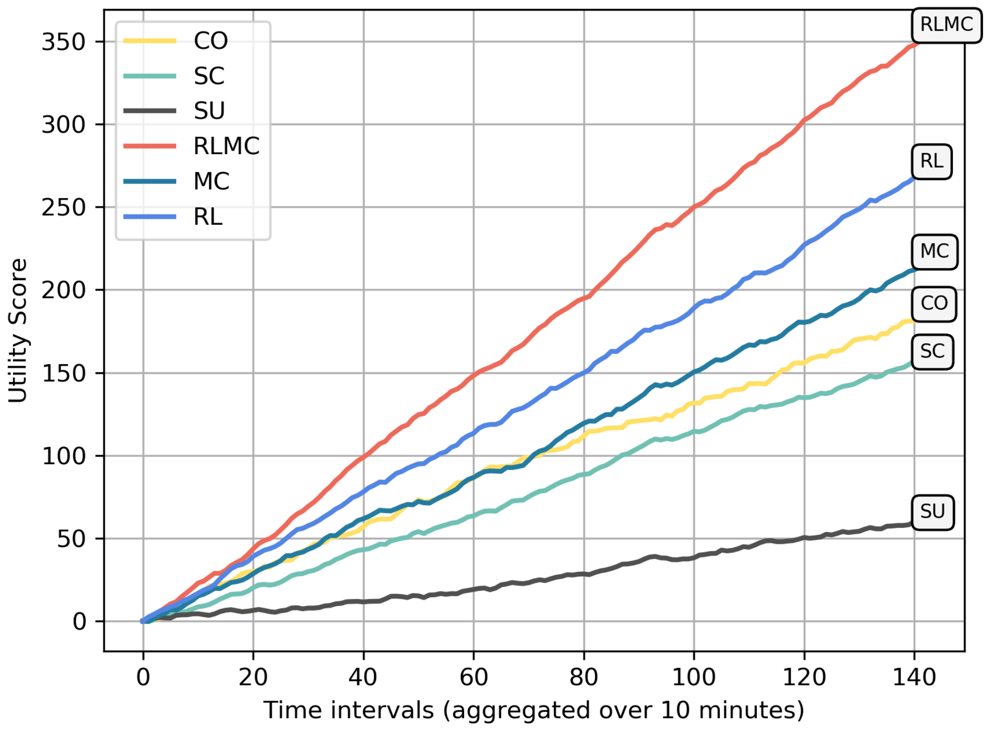


FIGURE 6.7: Cumulative Utility scores for each of the approaches

RQ3.3. *What is the efficiency of adaptations?*

We evaluate adaptation efficiency in terms of the number of corrections made by the model checker during each decision period. A scatter plot showing the number of corrections made per interval can be seen in Figure 6.8. The figure shows that in between intervals, there is an increase in the number of corrections, which goes as high as 6. At the start of simulation, correction count is high for the first two adaptation cycles (due to the time required for initial learning). Then, we can observe that the number of corrections made remains 0 until 160 minutes. This again increases and RLMC is able to continue without many corrections for some time until the next peak. The next peak is given when MC identifies that a decision at a given point is not feasible based on the expected context (behaviour/mode changes of sensors).

This behaviour illustrates the ability of RLMC to learn and improve from the results obtained by the model checker, as well as from the feedback obtained from ML forecasts. On average, the number of corrections amounts to 1, but there are instances where it is 0 and few instances where it is as high as 6 where RL is forced to generate decisions, incurring high penalties (negative rewards) and additional computation overhead. It is due to this effect that even with such a high increment in the number of corrections, RLMC is still able to continue operating for some time without need for new corrections. This indicates that RLMC is very efficient with respect to the number of corrections performed by the model checker.

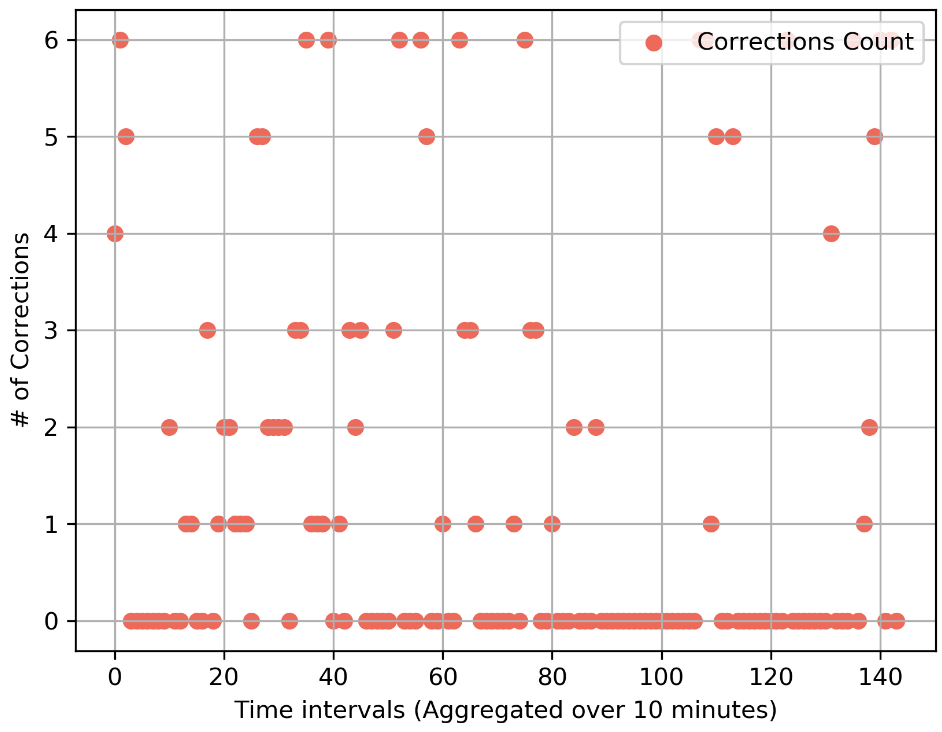


Figure 6.8: Number of corrections performed by MC per adaptation

RQ3.4. *What is the computation overhead of adaptation?*

To evaluate adaptation efficiency in terms of computation overhead, we clocked the time required to generate adaptation decisions. The results show that on average, RLMC takes approximately 2 seconds for generating an adaptation decision. From that time, the fraction employed by RL amounts to approximately 0.23 seconds because it consists of simple look up operations and state update. Each correction operation from model checker takes close to 1 second. Despite the overhead introduced by the model checker, the observed decision times are reasonable in the context of the class of IoT application described, showing feasibility of RLMC. Moreover, since the approach is proactive in nature, the system is not halted during the whole process.

6.6 Discussion

In this section, we provide details on the lessons learned from the evaluations and the threats to the validity of the evaluation.

6.6.1 Lessons Learned

ML and QV: A match for better decision making. Even though ML has shown effectiveness and efficiency in performing adaptions based on the data learned, it sometimes suffers from the problem of training/algorithmic bias [40], which can produce incorrect decisions. Our work demonstrates how using QV techniques can act as a guide for ML to select closer-to-optimal decisions. Our results indicate that QV can help improve the effectiveness of ML by reducing the error rate of producing infeasible decisions at least by 50%, increasing the overall utility by 39%. We believe that an effective combination of ML and QV for decision making can be a path forward for reliably architecting self-adaptive IoT and other types of adaptive CPS.

Effective self-learning through self-adaptation One of the main issues with reinforcement learning techniques such as Q-learning is that they take a lot of time to converge, leading to a lot of wrong decisions in the initial execution phase. As mentioned in Chapter 3, Section 3.3, there have been works that combine QV and ML, but all are reactive in nature, and the ones that use online learning does not make use of two-way feedback. Moreover, they take some adaptation cycles before a new model is generated. In this chapter, we have shown that (Figure 6.8) through the proactive nature of the approach, the use of QV, and by leveraging regular feedbacks through forecasts and verification, ML can be guided to auto-correct itself and further produce optimal decisions through faster convergence. When continued for a longer period, this process ensures that ML will gain a significant understanding of the best adaptation to be performed given a situation, thereby moving a step closer to self-learning architectures.

6.6.2 Threats to Validity

1. Construct Validity: It concerns the decisions made due to incorrect forecasts which could arise from the *Analyze activity.* The model might forecast high energy consumption, whereas the system would not have entered such a state, leading to lower utility scores. We understand this issue, and since the same prediction model and sensor data sets are used for the evaluation for all the three approaches, this does not over-weights or under-weights the overall utility score of any approach.

2. External Validity: It concerns the generalizability and scalability of our approach. Although our approach has been applied to a specific case study, it uses techniques that can be generalized to other classes of IoT systems with similar concerns (energy consumption, performance, availability). Moreover, we believe that our approach can be applied to more complex systems with a larger number of components with optimization of reinforcement learning and model checking components (e.g., using statistical model checking to improve scalability). Besides, the layered architecture used for implementation supports both horizontal and vertical scalability.

In conclusion, this chapter shows how ML and QV can be combined to architect self-adaptive IoT systems to optimize the guarantees of satisfying acceptable QoS levels with respect to energy consumption and network traffic. Further, we saw how this combination helps the overall approach converge faster towards optimal adaptations using the two-way feedback mechanisms, from the quantitative verification process and the forecasts of the succeeding adaptation intervals. Our evaluation shows that the approach exhibits a remarkable improvement (39% and 63%) over ML and QV's use in isolation. Although the use of QV adds computational overhead concerning using just ML, our results also show the feasibility of the approach, which can produce decisions within a reasonable timescale for IoT applications. We therefore tackle the challenge of *Quality assurances to learning* described in Section 1.2 and thereby address *RQ3*.

With this chapter, we conclude the core approach presented in this thesis. In the next chapter, we focus on demonstrating the generalizability of the techniques presented in the approach by applying it (especially the techniques presented in Chapters 4 and 5) to solve challenges related to service discovery in microservice-based architectures as well as self-adaptation challenges in microservice-based IoT systems.

Chapter 7

Data-driven Adaptation: The Case of Microservices

In previous chapters, we discussed how efficient and effective use of machine learning techniques enabled proactive adaptation of the architecture and further powered them to autonomously improve with every adaptation performed, supported with formal guarantees. In this chapter, we will see how such techniques, especially the ones presented in Chapters 4 and 5, can be applied to a more general class of systems, specifically to Microservice-based systems. Hence, through this chapter, we address *RQ4. How can the approach be generalized to other class of software systems?*

Microservice architectures (MSA) have become enormously popular since traditional monolithic architectures no longer meet the needs of scalability and the rapid development cycle of modern software systems (for more details on MSA, refer Section 2.4. The success of large companies (Netflix among them) in building and deploying services is also a strong motivation for other companies to consider making the change. The loosely coupled property of microservices allows the independence between each service, thus enabling the rapid, frequent, and reliable delivery of large, complex applications. This is evident from the fact that microservice-based architecture (MSA) is considered as one of the best possible solutions for architecting data-driven and event-driven systems like IoT [180]. However, like most transformational trends, architecting and implementing a microservice-based system poses its own *challenges:* Hundreds of microservices may be composed to form a complex architecture; thousands of instances of the same microservice can run on different servers; the number or locations of running instances could change very frequently [89, 90]. In addition, the set of service instances changes dynamically because of autoscaling, failures, and upgrades. Therefore, one of the challenges in a microservice architecture concerns how *services discover*, *connect*, and *interact* with each other. Consequently, elaborated *service discovery mechanisms* are required [91].

Moreover, *additional challenges* arise when microservices-based solutions are applied to IoT systems as the devices themselves are subjected to different uncertainties. These refer to the evaluation and maintenance of the Quality-of-Service (QoS) characteristics of systems (e.g., performance and reliability) due to the uncertainties faced by IoT devices because of resource constraints (e.g., battery level, network traffic) [22, 24].

In this chapter, we address the two specific challenges mentioned above by i) Applying the *combination of Machine Learning* (ML) techniques described in 5 to the service discovery process to perform context aware, service discovery; ii) Developing a novel *self-adaptive architecture for microservice-based IoT systems.*

7.1 Context Aware Service Discovery using Machine Learning

In this section, we focus on addressing the first challenge of performing effective service discovery in microservice based systems. Towards this, we use a combination of LSTM and Q-learning as described in Section 5.3.2 to preform machine learning-driven context-aware service discovery.

7.1.1 Service Discovery in Microservices: The Challenges

Service discovery mechanisms (refer Section 2.4.1) has continuously evolved during the last years (e.g., Consul, Etcd, Synapse, and ZooKeeper, etc.). A considerable effort has been reported to make the service discovery effective and efficient by improving the functional matching capability. This resulted in mechanisms that discovers all the available instances of a specific microservice very quickly while delegating QoS concerns to external load-balancing components (e.g., Elastic Load Balancing[1]). These solutions do not take explicitly into account the *context* and *QoSs*, which are transient and continuously change over time for several different reasons. For example, a service consumer/provider can change its context because of mobility/elasticity; a service provider can change its QoS profile according to day, time, etc. In this setting, our approach envisages a new service discovery mechanism that can deal with uncertainty and potential adverse effects attributed to frequent variability of the context and QoS profile of services.

[1]https://aws.amazon.com/elasticloadbalancing/

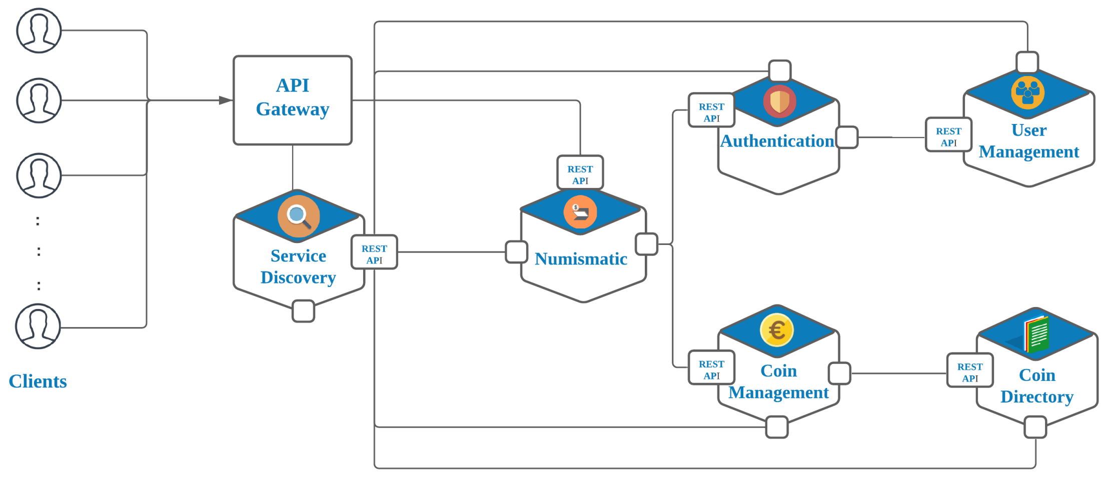


FIGURE 7.1: Architecture of the Prototype Application

7.1.2 Motivating Scenario: A Running Example

In this section, we elaborate on the motivation behind the approach through a prototype microservice-based application developed for managing coin collections of the users. The application provides different functionalities to the users by allowing them to: i) Register to the application by providing details such as name, location, etc; ii) Add information on the coins in hand (this information includes details such as name of the coin, country of use, etc; iii) Query information about the different types of coins available in the system; iv) Retrieve information on the different coins of a specific user or of the nearby users of a given location.

Figure 7.1 shows the high-level architecture of the coin coin collection application. As depicted in the figure, the prototype consists of five key microservices (excluding the Service Discovery):

1. *User Management* microservice is responsible for handling user management related operations such as adding new users, deleting users, managing user profiles, etc. It uses a database to store the user profile related information.

2. *Authentication* microservice provides functionalities to ensure that only authenticated users have permission to view/manage user-profiles and other user-related information. It has a simple database to store user credentials.

3. *Coin Management* microservice provides functionalities to manage coins such as adding/removing coins to/from the collection. It consists of a database for storing coin related information.

4. *Coin Directory* microservice is a simple directory management service that supports the coin management service in fetching additional information related to coin from external APIs.

5. *Numismatic* microservice accomplishes the key functionality of the application. It provides features such as retrieving user information, querying a specific user or nearby users' coin collection, and adding or removing coins from a user collection. It achieves these features by interacting with other microservices, as shown in Figure 7.1.

Each of the microservices is replicated as five microservices constituting an application consisting of 25 microservices. Further, each of them is geographically located at different locations and deployed in various types of machines with varying computing capabilities. External clients interact with the system through the API gateway, which further checks with the *Service Discovery* to fetch an instance of the desired microservice. Additionally, every microservice queries the service discovery before interacting with other microservices. For example, Numismatic service interacts with coin management microservice. This implies that every time numismatic service has to interact with coin management microservice, it first needs to identify the instance that needs to be invoked using the service discovery process.

However, one constraint that needs to be satisfied is that the clients should be served with the instance that offers the best QoS. For example, instance of a requested microservice that offers high throughput, reliability, least response time, etc. This can be challenging as the QoS of any instance of a given microservice depends on a number of parameters such as their geographic location, time of invocation, etc. So an ideal service discovery mechanism should have the ability to predict and select the instance that can guarantee the optimal QoS based on a given context.

7.1.3 System Model

In this section we introduce the terminology and notation used in the rest of the chapter, define the model of the system we are considering, and formally define the performance indexes we will use to measure the effectiveness of our approach.

We consider a set $\mathcal{S}$ of distributed services, hosted by different nodes in a networked system (e.g., edge, fog, or cloud architecture). We denote by s, s_i elements of the set $\mathcal{S}$. A Service $s \in \mathcal{S}$ is defined as a tuple (i, c, p, e), where:

- $s.i \in \mathcal{I}$ denotes the *interface* provided by the service, that is the functionality provided by s. We say that $s.i$ is the interface of s, and denote by i, i_j single elements of the set $\mathcal{I}$ [2].
- $s.c \in \mathcal{C}$ denotes the *context* of the service s. We say that $s.c$ is the context of s, and denote by c, c_j single elements of the set $\mathcal{C}$.
- $s.p \in \mathcal{P}$ denotes the *quality profile* of the service s. We say that $s.p$ is the quality delivered by s, and denote by p, p_j single elements of the set $\mathcal{P}$.
- $s.e \in \mathcal{E}$ denotes the *endpoint* of the service s. We say that $s.e$ is the endpoint of s, and denote by e, e_j single elements of the set $\mathcal{E}$.

Let $\mathcal{Q}$ be the set of lookup queries. A query $q \in \mathcal{Q}$ is a touple (i, c, p), where

- $q.i \in \mathcal{I}$ denotes the type of interface required by the service consumer.
- $q.c \in \mathcal{C}$ denotes the context of the service consumer.
- $q.p \in \mathcal{P}$ denotes the quality profile of interest for the service consumer.

Given a query $q \in \mathcal{Q}$, and a service set $S \in 2^{\mathcal{S}}$, the *Service Discovery* mechanism is defined according to the following functions, namely *match* and *select*:

- *match* : $\mathcal{Q} \times 2^{\mathcal{S}} \rightarrow 2^{\mathcal{S}}$ is a function that given a query $q \in \mathcal{Q}$ and a set of services $S \subseteq \mathcal{S}$, returns a set of services $\overline{S} \subseteq S$ such that $\overline{S} = \{s | s.i = q.i\}$.
- *select* : $\mathcal{Q} \times 2^{\mathcal{S}} \rightarrow \mathcal{S}$ is a function that given a query $q \in \mathcal{Q}$ and a set of services $S \subseteq \mathcal{S}$ returns a service $\hat{s} \in S$ such that $(\hat{s}.c \cong q.c) \wedge (\forall s \neq \hat{s} : \hat{s}.p \succeq s.p)$, where $\cong$ and $\succeq$ are defined according to some suitable criterion [130][134].

Therefore, the Service Discovery mechanism can be defined as a function that, given a query $q \in \mathcal{Q}$ and a set of services $SR \subseteq \mathcal{S}$ (Service Registry), returns a service instance $\hat{s}$, such that:

$$\hat{s} \leftarrow ServiceDiscovery(q, SR) \equiv select(q, match(q, SR))$$

Example 7.1. *Let $SR \subseteq \mathcal{S}$ be a service registry of the protype application depicted in Figure 7.1 storing the following services:*

$$s_1 = (\text{“}\mathtt{Numismatic}\text{”}, (\mathtt{loc}, \mathtt{EU}), ((\mathtt{responseTime}, 0.5), (\mathtt{throughput}, 100)), \mathtt{IP}_1)$$

[2]We use interchangeably the terms *interface* and *type* to denote the functionality of a service s

$$s_2 = (\text{“}\texttt{Numismatic}\text{”}, (\texttt{loc}, \texttt{EU}), ((\texttt{responseTime}, 0.3), (\texttt{throughput}, 300)), \texttt{IP}_2)$$

$$s_3 = (\text{“}\texttt{CoinManagement}\text{”}, (\texttt{loc}, \texttt{US}), ((\texttt{responseTime}, 0.2), (\texttt{throughput}, 500)), \texttt{IP}_3)$$

$$s_4 = (\text{“}\texttt{Numismatic}\text{”}, (\texttt{loc}, \texttt{US}), ((\texttt{responseTime}, 0.1), (\texttt{throughput}, 100)), \texttt{IP}_4)$$

Let $q \in \mathcal{Q} = (\text{“}\texttt{Numismatic}\text{”}, (\texttt{loc}, \texttt{FR}), \texttt{responseTime})$ *be a lookup query specifying that a given service consumer located in France is interested in a Numismatic Service providing best response time. Hence, applying the aforementioned functions, we obtain:*

$$\overline{S} = \{s_1, s_2, s_4\} \leftarrow match(q, SR)$$

$$\hat{s} = s_2 \leftarrow select(q, \overline{S})$$

where, $\cong$ *is defined according to geographical proximity (i.e., “close to”), and* $\succeq$ *is defined according to a less-is-better relationship (i.e. “*$\leq$*”). Then,* s_2 *is the Numismatic service providing the “best response time” among those services located “close” to France (i.e.,* s_1*, and* s_2*).*

In order to evaluate our approach, we define a performance index measuring the *QoS* delivered by all services. To this end, let $SB_t \subseteq \mathcal{S}$ be the set of services bound at a given time t. To measure the overall system performance, we define the *Average QoS* delivered by all services in SB_t:

$$\overline{QoS}(SB_t) = \frac{1}{|SB_t|} \sum_{s \in SB_t} s.p \tag{7.1}$$

7.1.4 ML Based Service Discovery

In this section, we describe how our approach uses the combination of ML techniques described in Chapter 5 to perform context-aware service discovery.

For each query, $q \in \mathcal{Q}$ received by the *Service Registry* component, $SR \subseteq \mathcal{S}$ from a service consumer s, the overall goal of the approach is to select the matching service provider $\hat{s} \in SR$ to maximize the QoS perceived by s with respect to the context of the service consumer, s as well as with that of $\hat{s}$.

Figure 7.2 shows the overall flow of the approach. On receiving a service query, the *Service Registry* component first identifies a matching set, $\overline{S}$ of instances based on the strategy defined in Section 7.1.3. It then uses a combination of ML techniques to select the best instance $\hat{s}$ from $\overline{S}$.

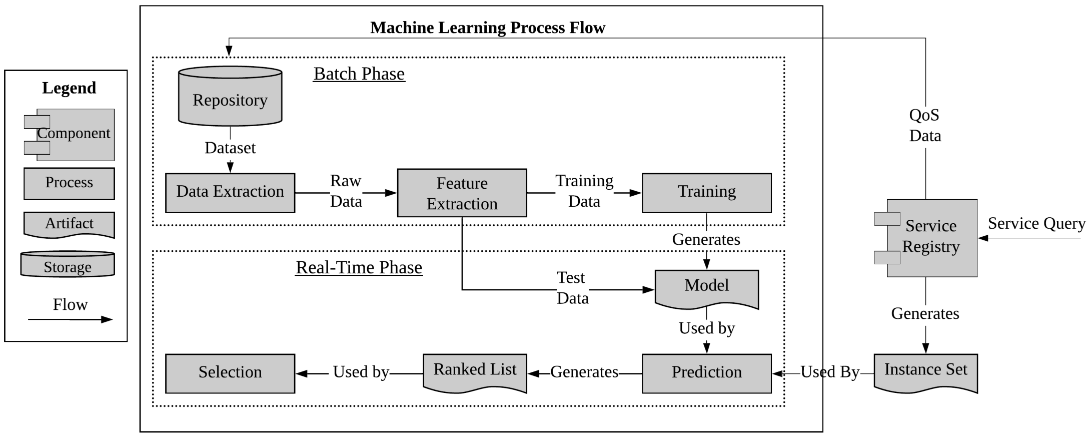


FIGURE 7.2: Machine Learning Process Flow

The first part of the *select* function is to estimate the expected QoS of every instance in $\overline{S}$. This is due to the fact that, based on the change in contexts such as time of the request, execution memory of instance, etc., there can be variations in the QoS offered by instances in $\overline{S}$ and neglecting this can lead to sub-optimal selections. Towards this, our approach uses deep neural networks that consider the historical QoS data offered by the instance along with their context data to *predict* the expected QoS for every instance in $\overline{S}$. The QoS forecasts alone are not sufficient for selection as the perceived QoS of $\hat{s}$ changes based on the context of s. In order to accommodate this, our approach further uses the RL technique to select the best instance $\hat{s}$ from $\overline{S}$.

The overall ML process of the approach primarily consists of two phases (as represented in the figure 7.2) (similar to the approach presented in 4) namely the *Batch Phase* and the *Real-Time Phase*. Training and building ML models for performing forecasts is a time-consuming process. For this reason, our approach involves periodic execution of *Batch Phase* where the historical *Quality Profile* information of each service $s_i \in SR$ as well as their context information are used for training and building models for forecasting the QoS. The *Real-Time Phase*, on the other hand, consists of the instance selection process that happens in real-time. In order to achieve this, it uses the latest ML models available from the batch phase to forecast the expected *Quality Profile* continuously of each service $s_i \in \overline{S}$. It further processes this forecast along with context of s using RL technique to *select*, $\hat{s}$ from $\overline{S}$

7.1.4.1 Data Extraction

The Quality Profiles (i.e., $s.p$) of all the service instances (e.g., response time, throughput, etc.) as well as the context information (i.e., $s.c$), which includes details such as instance memory, geographical location, etc. are continuously monitored and stored in a *Repository* component in the *Service Registry*. This forms the raw QoS/Context data. During the batch phase, the *Data Extraction process* retrieves the QoS/Context data from the *Repository*. This raw monitoring data contains the information of different QoS/Context attributes of every instance for different intervals of time. This data is sent to *Feature Extraction* process for further processing.

7.1.4.2 Feature Extraction

This process converts the raw monitoring data extracted during the data extraction process into a structured set of features as required by the ML technique.

This data has a temporal nature, and we use this to convert the problem of predicting QoS into a *time-series forecasting* problem. The QoS data with respect to time forms a *continuous time-series* [71]. For the ease of analysis, we first convert this into a *discrete time-series* by aggregating the data into uniform time intervals.

Let us assume to have m different service instances $s_1, \dots, s_m$ providing a given interface $i \in \mathcal{I}$, which have been running for n units of time. Each of these m instances has an associated d dimensional *Context Feature-set* $C_f \subseteq \mathcal{C}$ describing the context of each running instance s_j.

Definition 7.1 (Context Feature-set)**.** We define context feature as a set $C_f \subseteq \mathcal{C} = \{c_1, c_2, .., c_n\}$ where c_j represents a pair (l, v) such that l is a unique label identifier for a context attribute, and v denotes the value of the context attribute.

For example, $C_f = \{(\texttt{loc}, \texttt{IT}), (\texttt{day}, \texttt{Mon}), (\texttt{hour}, \texttt{10}), (\texttt{min}, \texttt{30})\}$ denotes that a given service instance $s \in S$ is located in Italy, day is Monday and the current time of invocation is 10:30.

Then for each instance, the observation at any instant of time t can be represented by a matrix $O \in V^{d \times n}$ where V denotes the domain of the observed features. The process of generating time-series results in the formation of a sequence of the form $O_1, O_2, O_3....O_t$. The problem of forecasting QoS values is then reduced to predicting the most likely k-length sequence in future given the previous j observations that include the current one:

$$O_{t+1}, ..., O_{t+k} = \underset{O_{t+1},..O_{t+k}}{argmax} \; P(O_{t+1}, ..., O_{t+k} | O_{t-j+1}, O_{t-j+2}, ..., O_t)$$

where P denotes the conditional probability of observing $O_{t+1}, ..., O_{t+k}$ given $O_{t-j+1}, O_{t-j+2}, ..., O_t$. Since we also consider the context data for forecasting the QoS level of each instance, the observation matrix, O can be considered as a multivariate time-series dataset and as the forecasting needs to be done for the next k steps, the problem of time series forecasting becomes a multivariate multi-step forecasting problem [71] (refer Seciton 4.2.2, Chapter 4).

Each column in O represents a feature vector, v. The process of feature scaling is applied to each column v such that $v_i \mapsto [0,1]\ \forall v \in O$. O is then divided into two data sets, training set, O_{train} and testing set, O_{test} in the ratio 7:3 respectively. The training set obtained is further sent to the *Training* Process.

7.1.4.3 Training

The training process uses the training set, O_{train} to create ML models for forecasting the expected QoS of every service instance, $s_j \in SR$ for a given time period known defined by the forecast horizon, H.

The approach makes use of LSTM for building the forecast models. In Chapter 4, we have described how LSTM networks (refer Section 4.2.2.2) can be used for forecasting QoS of the sensor components and why they are more effective when compared to traditional models. In this approach, we use the same techniques to train the LSTM network for forecasting the QoS of service instances. The training process results in the creation of a *Model*, which is further tested for accuracy using the test set, O_{test}. In the event of low accuracy, the approach performs retraining by tuning the neural network parameters. The tested models are further used by the *Prediction* process.

As explained in the previous chapters, the process of training is executed as a batch process at regular intervals to constantly update the models so as to avoid the problems of concept drift [113].

7.1.4.4 Prediction

Prediction is a real-time process responsible for forecasting QoS of the matching instances. For every query, q received by the service registry, the prediction process uses the trained LSTM models to generate the QoS forecasts for each $s_j \in \overline{S} \leftarrow match(q, SR)$. The definition of QoS forecasts (as defined in 6 is slightly modified as follows:

Definition 7.2 (QoS Forecasts)**.** We define the QoS forecasts as a set, $F = \{p_1, p_2, .., p_n\}$ where p_j represents the forecasted quality profile. Note that, $p_j = (a, v)$ where a identifies the quality attribute and $v \in \mathbb{R}$ denotes the forecasted quality value over the duration of the time horizon h.

For example, $H = (\texttt{10}, \texttt{sec})$ and $F = \{(\texttt{responseTime}, \texttt{0.2}), (\texttt{throughput}, \texttt{200})\}$ for a given instance s denotes that s is expected to have an average response time of 0.2 seconds and a throughput of 200 requests/second in the next 10 seconds.

These QoS forecasts for each of the service instances $s_j \in \overline{S}$ are then sent to the *Selection* process for further processing.

7.1.4.5 Selection

The role of selection process is to select the best instance $\hat{s}$, such that, $\hat{s} = s_2 \leftarrow select(q, \overline{S})$, based on the context of the service consumer as well as the forecasts of the expected QoS of each instance $s_j \in \overline{S}$. In order to achieve this, we use the Q-learning based algorithm similar to what used in Chapter 5, algorithm 2. In this approach, the state space of Q-learning is determined by two important attributes: (i) *QoS Categories*, and (ii) *Context Feature-set*.

Definition 7.3. (QoS Categories) We define QoS category, QC as a discrete set $\{qc_1, qc_2, ...qc_n\}$ where qc_j represents the expected category for the QoS metrics j, obtained by mapping the values of the QoS forecasts f_j to a unique label, $l \in \mathcal{L}$.

Algorithm 4 Instance Selection Algorithm

```
Require: :
 1: States $W = \{w_1, w_2, w_3, ...., w_n\}$
 2: Actions $A = \{0, 1, 2....m\}$                ▷ represents the selection of each $m$ instance
 3: Labels $L = \{l_1, l_2, l_3, ...l_p\}$                ▷ Threshold categories
 4: Forecasts $F = \{f_1, f_2, ..f_n\}$                ▷ Forecasts of QoS attributes
 5: Rewards $R = \{r_1, r_2, ..., r_n\}$                ▷ Reward for each of the state
 6: procedure DECISION-MAKER($W$,$A$,$L$,$F$,$R$,$\mathcal{C}$,$\alpha$,$\gamma$ )  ▷ Find the state of the system
    from the forecasts and context
 7:     $QC \leftarrow identify\_category(F, L)$                ▷ Get Category from Forecasts
 8:     $w \leftarrow (QC, c)$                ▷ combine category and context to form the state
 9:     $r \leftarrow R[w]$                ▷ Reward for attaining the state, $w$
10:     $(w', a') \leftarrow argmax_a Q(w, a)$
11:     $Q'(w, a) = (1 - \alpha) * Q(w, a) + \alpha * (r + \gamma * max(Q(w', a)))$
12:     $a \leftarrow a'$                ▷ The action to reach that state
13:     return $a$
```

For instance, let $F = \{(\texttt{responseTime}, \texttt{0.3}), (\texttt{throughput}, \texttt{100})\}$ represent the forecast vector and $L = \{\texttt{lowRt}, \texttt{highRt}, \texttt{lowTh}, \texttt{highTh}\}$ denote a set of labels. Then we can define a simple mapping function for the QoS attribute,
responsetime such that $[0, 0.2] \mapsto \texttt{lowRt}, [0.2, \infty] \mapsto \texttt{highRt}$. Similarly, we can define another mapping function for the QoS attribute, *throughput* such that $[0, 40] \mapsto \texttt{lowTh}, [40, \infty] \mapsto \texttt{highTh}$. We can then combine this to generate a QoS category set, QC for the given F as $QC = \{highRt, highTh\}$.

The state space, $W \subseteq QC \times \mathcal{C}$ is defined over the set of possible QoS categories and the set of all context features in $\mathcal{C}$.

Action space, A, consists of a set of actions $\{a_1, a_2, a_3, .., a_m\}$ such that $a_j \in A$ denotes the selection of instance s_j in $\overline{S}$. Rewards, R on the other hand is a set $\{r_1, r_2, ..r_n\}$ where $r_j \in \mathbb{Z}$ denotes the reward value for attaining a state, $w_j \in W$.

W and A together form a $n \times m$ Q-table matrix where n denotes the number of states and m denotes the number of actions. Each value corresponds in Q-table to a (w, a) pair and it's value denotes the benefit for selecting an instance s_j given the context of the service consumer and the expected QoS category of the instance. This property is leveraged by the selection process to select the best instance during the process of service discovery. The complete instance selection algorithm based on Q-learning is presented in algorithm 4. It is similar to algorithm 2 presented in Chapter 5. While the core algorithm remains the same, the key difference is that instead of just using the QoS states for identifying the state inside the Q-table, this algorithm also considers the context features. Moreover, the action space here is about switching between instances as opposed to patterns leading to an increased action space.

The algorithm first maps the forecasts received from the prediction process, F into a set of QoS Categories, QC based on the labels, L. The QC identified along with the context of the service consumer, $\mathcal{C}$ is combined to identify the current state, w inside the Q-table (lines 7-8). The algorithm then assigns a reward for attaining the state, w. Following this, the maximum value of state-action pair, (w', a') that can be reached from the current state is identified (lines 9-10). The Q-table is then updated using the Q-function (line 11) and the action (instance, $\hat{s}$) is returned (lines 12-13).

In this manner for every query, q, our algorithm selects the best instance $\hat{s}$ by selecting the action that maximizes the reward. By assigning negative rewards (penalties) to selection of instances that offers sub-optimal QoS, the approach ensures that any incorrect selection is penalized in the form of a high negative reward and this means as the time progresses, given a q, the algorithm continuously improves the selection process to guarantee optimal QoS by ignoring instances that can lead to high penalties.

7.1.5 Evaluation

In this section, we first describe our experimental setup, as well as the data and metrics used for the evaluation of our approach, followed by a discussion of the results generated by our approach. We used *response time* as the key QoS parameter for the evaluation of our approach. Hence, for each query, $q \in Q$ requested by the service consumer, s, the objective will be to select the instance, $\hat{s}$ that is expected to provide the best response time (best response time implies least response time).

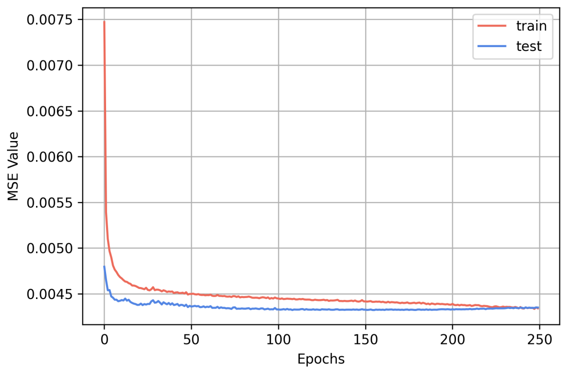
Experiment Specification. For experimentation and evaluation, we used the prototype application described in Section 7.1.2. The microservices were implemented using Java Spring Boot. The service discovery module was primarily implemented in Java while the ML part was implemented using Python (Keras framework with Tensorflow backend). The module also supports integration with technologies like Zookeeper, Netflix Eureka, etc[3]. For more details on the technologies used, we refer the readers to Appendix A.

Experimental Setup. Our system was deployed on two VM instances in Google Cloud. The first one was run on an N1-Standard-4 CPU Intel Haswell Processor comprising 4 vCPU and 17 GB RAM with US-Central-a as the geographical zone. The second one ran on an N2-Standard-4 Intel Skylake processor comprising 2 vCPU and 28GB RAM with Europe-West-a as the geographical zone. The microservice instances were distributed between the two VM instances. This was done to capture the different context dimensions that may arise from the type of CPU, number of cores, geographical zones, etc.

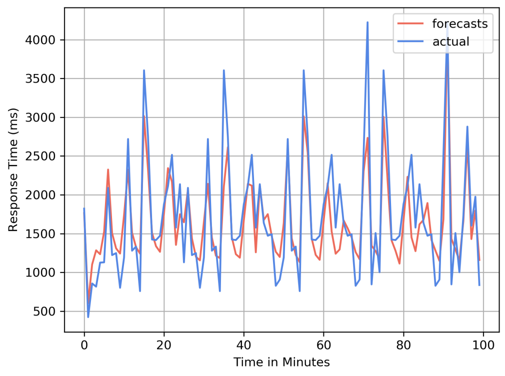
Evaluation Metrics. The objective of our evaluation is to assess the effectiveness and efficiency of the approach. The effectiveness of the approach is evaluated by i) measuring the *accuracy* of the response time forecasts produced; ii) calculating the *average response time* delivered by all service in SR, $\overline{QoS}(SB_t)$ as defined in Section 7.1.3; iii) computing the response time offered by the instance, $\hat{s}$ for every request made. Efficiency, on the other hand, is measured based on the average time taken to execute the matching and selection process.

Data Setup. We deployed the system (using standard service discovery mechanism) for a period of one week, and we developed 10 different clients to send requests to various microservices based on a Poisson distribution with different mean values based on the given day of the week. To simulate a different workload between instances of the same microservice, a delay has been added at each request that depends on the type of instance and the current time. This was done to emulate the real scenario where the incoming request rate can vary depending on the day and time. The response time of all the instances was continuously monitored and recorded. This was then used to create the LSTM based forecast model with a forecast horizon, $H = 5\ minutes$.

[3]The complete implementation along with the source code and dataset can be found at https://github.com/detomarco/machine-learning-driven-service-discovery



(A) Train test loss curve



(B) RMSE Plot

Figure 7.3: LSTM Response Time Forecasts

Evaluation Candidates. We evaluated the approach by deploying the system integrated with each of the five different strategies for a period of 72 hours. These strategies form the evaluation candidates:

1. static-greedy (sta_gre): Instances are ranked based on the static response time registered by the instances during service registration. Selection is performed using a greedy strategy (select instance with the least response time).

2. linereg-greedy (lin_gre): Prediction of instance response time is performed using linear regression [15] and selection is performed using a greedy strategy.

3. timeseries-greedy (tim_gre): Prediction of instance response time is performed using time-series and selection using greedy.

4. linereg-Q-Learn (lin_Qle): Prediction of instance response time is performed using linear regression and selection using Q-learning.

5. timeseries-Q-Learn (tim_Qle): Our approach, which performs prediction of response time using time-series and selection using Q-learning.

7.1.5.1 Approach Effectiveness

Forecast Accuracy. The first part of evaluating the approach effectiveness was to measure the accuracy of the response-time forecasts produced by the LSTM models. The dataset was divided into training and testing sets consisting of 8903 and 2206 samples, respectively. We build the LSTM model with a single hidden layer consisting of 60 neurons. This number was selected through experimentation. Mean Squared Error (MSE) loss function and the efficient Adam version of stochastic gradient descent were used for the optimization of the LSTM models. The model was fit in 250 epochs

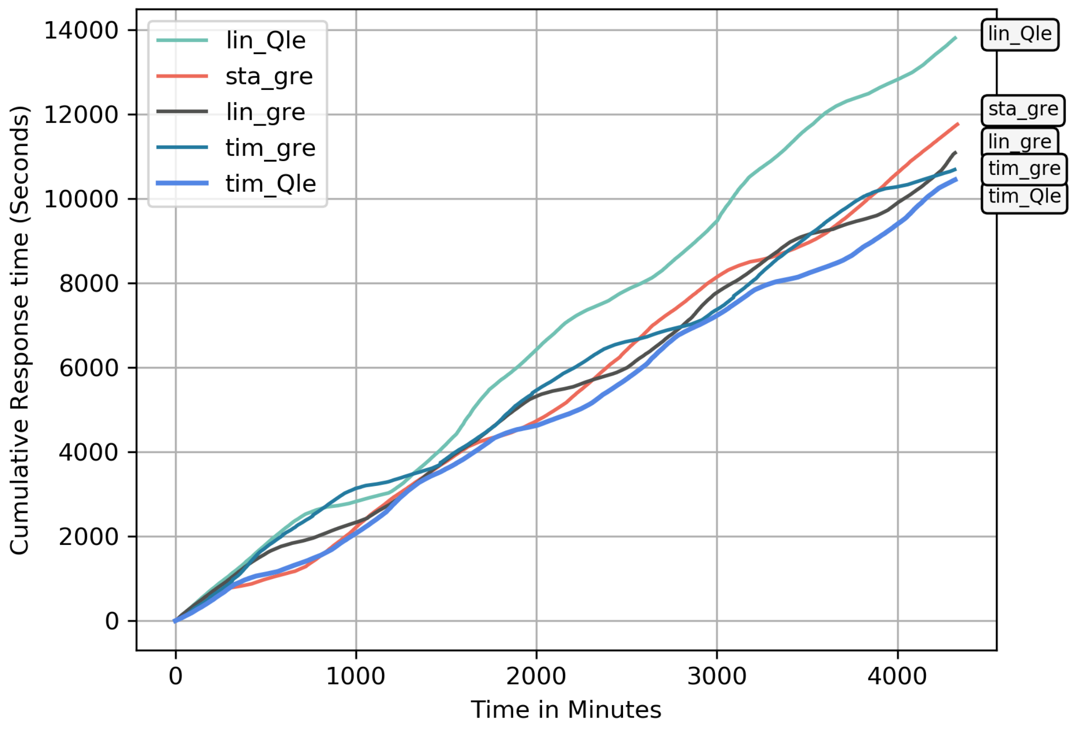


FIGURE 7.4: Cumulative $\overline{QoS}(SB_t)$ per minute - lower is better

(Figure 7.3a shows the training vs. testing loss curve). We used Root Mean Square Error (RMSE) for evaluating the accuracy of LSTM models on the testing set, where RMSE for a dataset, O_{test} with n samples is given by the formula:

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}\Big(p_i - y_i\Big)^2} \tag{7.2}$$

where p_i represents the predicted value and y_i represents the actual value.

The calculation resulted in a value of 406.73 ms, which means, on average, there is an error of 406.73 ms in prediction. Figure 7.3b shows the plot of actual versus forecasted values. We can see that the LSTM model's prediction is almost able to follow the curve of actual response time. This is due to the fact that LSTM, being a deep neural network, possesses the ability to identify any non-linear dependency that might exist between the different features such as the context attributes to generate accurate forecasts.

Average QoS Per Minute The second part of measuring the effectiveness was to calculate the metric, $\overline{QoS}(SB_t)$ as defined in Section 7.1.3. To accomplish this, we deployed our experiment by integrating the service discovery mechanism using each of the evaluation candidates for a period of 72 hours. The batch training phase was executed every 12 hours. The value of $\overline{QoS}(SB_t)$ was calculated for every minute, and Figure 7.4 shows the plot of the cumulative $\overline{QoS}(SB_t)$ while using each of the approaches. We can see that the cumulative $\overline{QoS}(SB_t)$ offered by the different approaches starts diverging marginally during the initial stages, but as time progresses, the gap between *tim_Qle* and other approaches slowly starts increasing. In particular, we can see that the gap between *tim_Qle* and the traditional *sta_gre* keeps increasing steadily after 1900 minutes, thereby resulting in the improvement of *tim_Qle* (10449 seconds) over *sta_gre* (11762 seconds) by 12% at the end of 72 hours. *lin_gre, tim_gre* on the other hand does perform better than *sta_gre* with *tim_gre* performing better than *lin_gre*. However, we can observe that the learning rate of these approaches is still less than *tim_Qle*, and they are often inconsistent. This is more evident, especially after 1900 minutes. This is due to Q-learning's ability to continuously improve with the help of feedback obtained for every selection performed. For Q-learning, the feedback for decision at a time, t obtained via forecast at the time, $t+1$. Hence poor forecast accuracy implies that Q-learning favors the selection of wrong instances, and due to this reason, *lin_Qle* performs the worst.

Average QoS Per Request To further evaluate the approach effectiveness in terms of the QoS offered by the instance, $\hat{s}$, we measured the response-time of $\hat{s}$ for every request made to $\hat{s}$. Figure 7.5 shows the box plot of the *response time* offered by $\hat{s}$ per request. The *tim_Qle* approach offers the least average response time of about 1236.23 ms which is $20\%, 19\%, 21\%$ and 16% better than the one offered by *sta_gre*, *lin_gre*, *lin_Qle* and *tim_gre* respectively. We can also observe that most of the values fall between the range of 500 ms and 1800 ms, which is much less compared to other approaches. Although, there are more outliers in the case of *tim_Qle* due to the initial phase of Q-learning where wrong selections are made, on average, the response time perceived by s when using by *tim_Qle* is significantly lower compared to the others.

7.1.5.2 Approach Efficiency

The efficiency of the approach was evaluated by measuring the time taken by the whole service discovery process when integrated with our approach. The results show that, on average, the approach takes 0.10 seconds for performing the whole process. The speed can be mainly attributed to the fact that Q-learning being a model-free technique, performs only a lookup operation in the Q-table. The majority of the time is taken by the prediction process as although it's a constant time process, the prediction needs to be done for service instances in each of the five services. LSTM training, which happens every 12 hours, takes around 125 minutes to complete, but this does not impact the real-time process as only the trained models are used for service discovery.

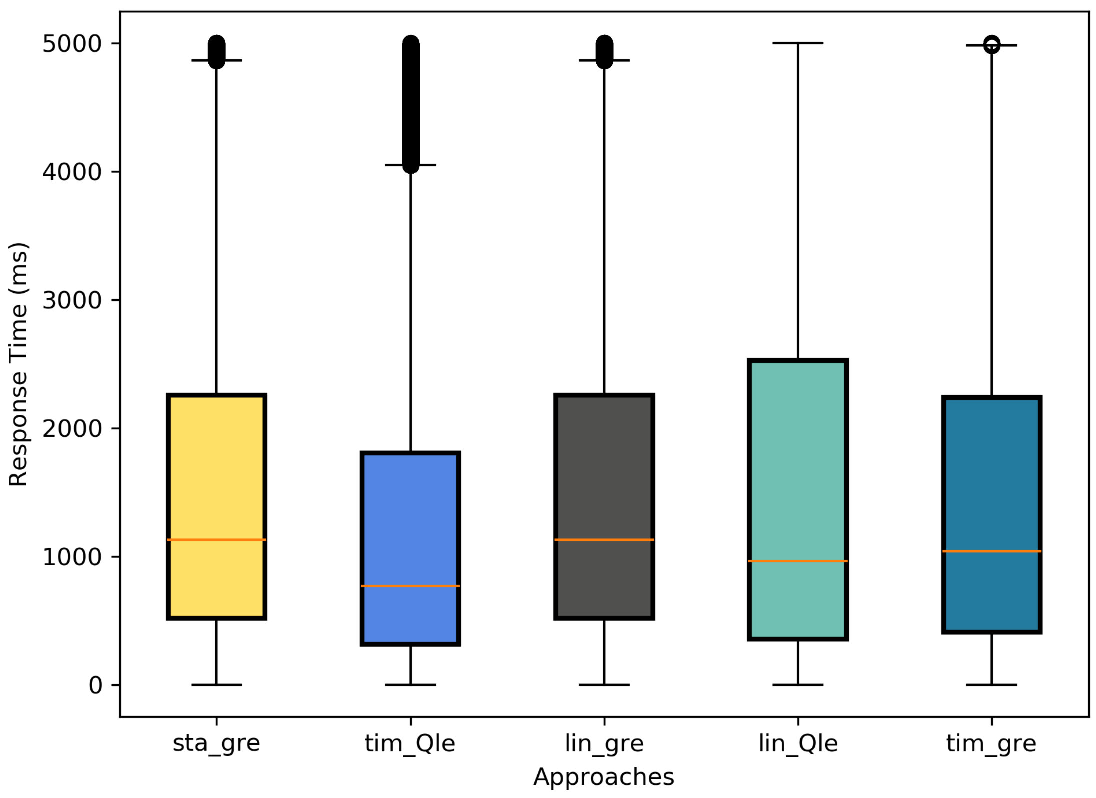


FIGURE 7.5: Response time of $\hat{s}$ per request

7.1.6 Threats to Validity

1. Construct Validity: It is related to the use of a controlled experimental setup and incorrect selections. Even though we performed *real-time execution* of the system, we simulated the context and quality change to emulate real-world scenarios as close as possible. In particular, the context setup such as the server locations, request simulations, dynamic delays were all incorporated to achieve this goal. To improve the selection accuracy, the build phase is executed at regular intervals. By keeping the reward for incorrect selections to a high negative value, wrong decisions can also be penalized.

2. External Validity: This concerns the generalizability and scalability of our approach. Although our approach has been applied to a system with 25 services, it uses techniques that can be generalized to more complex microservice systems. We have already seen this in the previous chapters as we have applied similar techniques to IoT systems. Moreover, our approach can be catered to large-scale systems by optimizing Q-learning (for instance, using Deep Q-learning [181]) to solve the effectiveness and efficiency issue that might arise from the increased state, action space.

In this section, we proposed a novel service discovery mechanism that takes into account the frequent variability of the consumers and providers contexts and the QoS profile of services. Therefore, the overall goal of the approach was to select the matching service provider $\hat{s}$ to maximize the QoS perceived by the service consumer s with respect to its context and with that of $\hat{s}$. For this purpose, we developed an approach by extending the approach presented in Chapter 5. It uses a combination of deep neural networks and reinforcement learning to select the best instance of $\hat{s}$. The approach has been evaluated in a controlled environment. Experimentation results are encouraging and show how such an approach is able to outperform traditional service discovery mechanisms.

In the next section, we shift our focus to microservice-based IoT systems and further elaborate on how the self-adaptation challenges in such a system can be handled using different techniques presented so far.

7.2 Data-driven Adaptation in MSA-IoT

In this section, we describe our novel self-adaptative architecture for MSA-IoT systems. The architecture considers the adaptation concerns that arise from different involved entities such as those from IoT devices, microservices, and the end-user due to changing user goals. It addresses this by leveraging the different types of data obtained from the different involved entities and further through a clear *separation of adaptation concerns*. The architecture is further explained in the remainder of this section.

7.2.1 MSA-IoT: The Challenges

As discussed at the beginning of this chapter, although MSA's are turning out to be the goto solution for architecting modern IoT systems, lots of new challenges arise when applying MSA for IoT. These challenges can be classified into three levels based on their origin. These include:

1. Device Level: IoT devices face different uncertainties due to the openness and continuously evolving environments such as device failures (e.g., hardware issues resulting in burn out of microcontroller), device damages (e.g., cameras can be damaged due to some human activity or by other external factors). Moreover, they are also subjected to various resource constraints arising from the availability of battery, network traffic, etc.

2. Microservice Level: Microservices themselves are subjected to various uncertainties due to challenges of resource management (e.g., monitoring VMs and containers which can fail/have resource constraints

3. Application Level: Apart from devices and microservices, the application itself is subjected to different uncertainties due to the change in usage contexts or changes in user goals.

Towards this, *Self-adaptation techniques* has emerged as one of the potential solutions due to their abilities to handle the uncertainties. We have already seen in the previous chapters how leveraging QoS data using machine learning techniques can result in effective proactive adaptations in IoT based systems. But, increasingly demanding users with dynamic behaviors, the uncertain nature of microservices (sudden service failures, memory outages, etc.) coupled with the *open* nature of the IoT context, call for *reactive adaptation* techniques as the uncertainties may not be possible to predict. Concerning the challenges mentioned above, an ideal MSA-IoT system should have self-adaptive capabilities to handle the uncertainties both reactively and proactively.

However, a recent study by Mendonça et al. [137] has shown that traditional self-adaptation techniques that make use of MAPE feedback loops or three layers models, in general, may not work in MSA. This is due to a *fundamental mismatch between the adaptation needs of microservice-based systems and the support offered by traditional self-adaptive frameworks and models.* Towards this direction, the approach in [137] proposes a service-mesh based approach for handling adaptation. That approach works at the microservice level only, while MSA-IoT systems expose adaptation needs that can arise even from the device level as well as the application level, and they might further impact the adaptation needs of the microservices.

This calls for an adaptation framework that encompasses the adaptation concerns arising from these different levels.

In this direction, in the remainder of this section, we describe a novel *self-adaptive architecture for microservice-based IoT systems.* In particular, the architecture handles proactive adaptation by using *machine learning* (ML) techniques based on what is proposed in the previous chapters, and reactive adaptation by exploiting *dynamic microservices composition.* Furthermore, it manages the adaptation at different levels and in a different manner: i) at device level, through the monitoring of QoS data of IoT devices, through a fog layer, ii) at service level, by continuously monitoring the QoS data of services and effectively leveraging the use of service mesh; iii) at application level, employing dynamic QoS-aware service composition, driven by users goals.

7.2.2 Motivating Scenario: MSA for NdR

In Section 2.5, we have described the high-level architecture designed for the NdR application. However, as stated in the description, we had only considered a part of the NdR application. In the larger context, the NdR application's requirement is to provide different functionalities that can improve the overall quality of the visiting experience. These functionalities include providing information on events, venues, parking lot, localization, etc. using various sensors, i.e., people counter, parking mats, beacons, cameras, QR code reader, etc. These sensors are deployed mostly in outdoor environments. Based on this scenario, we plan to develop an MSA-IoT application consisting of microservices for handling venue booking, venue management, localization, payment, etc. We decided to go with MSA due to the advantages offered (refer Section 2.4).

In this context, considering the heterogeneity and multitude of involved actors, such as IoT devices, microservices, and users, different kinds of uncertainties have to be managed. They can be classified into three levels, corresponding to the involved actors. For example:

i) **application level:** Suppose users want to use online booking for both the venue and desired transport mode. Embedding in the system behavior, all the possible combinations is cumbersome. Possible adaptation than can be to *dynamically* combine the venue booking and selected transport mode microservices (e.g., exhibition event and taxi) to accomplish the user goal.

ii) **microservice level:** Consider that the booking microservice suddenly gets lots of requests due to a popular event, and we gather context information from the camera about the flow of people to venues. This information, along with expected response time, can be used to perform proactive adaptation by adding new instance(s) of the booking microservice.

iii) **device level:** Consider that hand-held QR code reader has limited battery capacity. ML can be used to predict the battery level and dynamically adapt the data transfer frequency.

7.2.3 Self-adaptive Architecture for MSA-IoT

The overall architecture as depicted in Figure 7.6 consists of three layers, namely *Edge*, *Fog* and *Cloud*. While the fog layer handles the device level adaptations specific to devices in the edge layer, the cloud layer handles adaptations at the microservice and application levels.

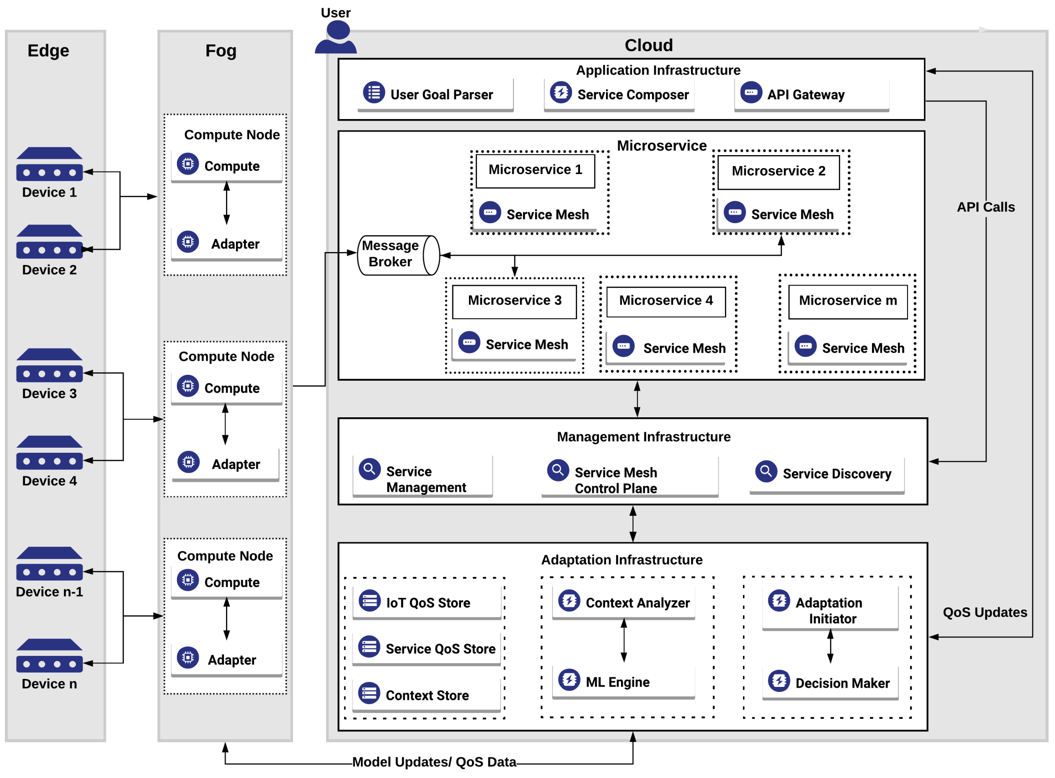


FIGURE 7.6: Self-Adaptive architecture for microservice based IoT Systems

7.2.3.1 Edge and Fog Layer

The *Edge* layer represents the set of IoT devices (sensors and actuators) in the system. Sensors send the data sensed to the *Fog layer* based on the frequency of data transfer. They also periodically send their QoS data including information such as battery level, memory consumption, etc. to *Fog Layer.*[4].

The *Fog Layer* is responsible for performing the lightweight computations on the sensed data and further perform architectural adaptations/re-configurations on the IoT devices if required based on the QoS data. It consists of multiple *Compute Node* consisting of a *Compute* component and an *Adapter* component. The former is responsible for performing preliminary computations on the sensed data such as data aggregation, cleaning, etc. The later is responsible for leveraging the QoS data obtained from the devices, by using ML models to perform *proactive* adaptations of the IoT devices. The adapter applies some pre-processing such as feature scaling, normalization, etc. on the real-time QoS data of the IoT devices. It then uses ML models to predict the expected QoS for a given time interval (Similar to the role of *Predictor* in the *Analyze* activity of our overall approach). These models are periodically received from the cloud layer. The Adapter then selects an adaptation plan, to be used if any QoS issue is forecasted, and

[4]In Figure 7.6 two IoT devices are shown to communicate to one compute unit in the Fog. This is shown for ease of understanding. In reality depending on the use case these numbers can change

communicates it to the compute component. The selected plan is then used to perform device-level adaptation, such as reduce the sensor data transfer frequency, modify the communication protocol, etc (eg. device level in Section 7.2.2). The *Fog Layer* further communicates the data received, which includes the QoS data as well as the sensed data, to the *Cloud Layer* through the *Message Broker*.

7.2.3.2 Cloud Layer

This layer performs heavyweight computations. It consists of four main layers. (Despite the ordering depicted in Figure 7.6, we describe, in the order, microservice, management, adaptation and application infrastructure layers, for comprehensiveness).

1) Microservice Layer: It consists of the set of microservices implementing the functionalities of a given IoT system. For monitoring the individual microservices and further use this for performing adaptation, we use the concept of *service-mesh/sidecar*, as suggested by Mendonça et al. [137, 182] (for more details on service mesh, refer Chapter 2, Section 2.4.2). The service-mesh provides ways to monitor various QoS parameters of each microservice, such as traffic, response time, etc., that can be obtained from the *Service Mesh Control Plane* in the *Management Infrastructure layer*.

2) Management Infrastructure Layer: It handles the discovery of microservices, provides information on their status, manages them and executes adaptation if needed. It consists of three main components: i) *Service Management*, it is responsible for executing the architectural adaptation of the microservices, e.g., increasing the memory of the microservice instance, automatically add an instance, etc. based on the adaptation decision provided by the lower layer. It is also responsible for providing information on the status of the microservices upon request to the ii) *Service Mesh Control Plane*, it regularly monitors the QoS level of every microservice. These QoS data are further sent to the *Adaptation Infrastructure layer* for processing; iii) *Service Discovery*, it routes the requests from the *API Gateway* to the respective instances of microservices. The discovery can be performed by leveraging the service instances' QoS data using the machine learning technique described in Section 7.1.

3) Adaptation Infrastructure Layer: This is a dedicated layer providing mechanisms for effectively supporting adaptations at different levels. It is responsible for collecting the context and QoS data from the *Fog* and *Management Infrastructure layers*. It leverages these data to generate learning models for QoS prediction and further decide on the best adaptation strategy based on the context. It consists of the following:

i) *IoT QoS Store*, ii) *Service QoS Store* and iii) *Context Store* stores information such as device-level QoS, service level QoS, and sensor data from Fog Layer, respectively.

iv) *ML Engine* is the key component as it is responsible for leveraging the data obtained to create forecast models that can predict the expected QoS of IoT devices and microservices. It is same as the *Machine Learning Engine* of the approach presented in the previous chapters. It mainly uses two data sources, IoT and Service QoS Stores. These data form time-series datasets consisting of the QoS values for different intervals of time. ML Engine applies pre-processing steps on these datasets, such as feature scaling, data aggregation, etc. These pre-processed data are then used for creating forecast models, by using LSTM networks [70] as defined in Chapter 4. This process is repeated periodically to ensure continuous update of the models thereby avoiding the possible issue of concept drift [113]. It results in the creation of two types of models: 1) for forecasting the QoS of IoT devices, which are communicated periodically to the Fog layer, and 2) for forecasting the QoS of microservices, which are sent to the *Context Analyzer*;

v) *Context Analyzer* is responsible for identifying the need for adaptation of microservices based on the QoS forecast models from the ML engine and the data from the context Data Store. It follows a two-step process. First, at every instant of time, it obtains the latest Service QoS data to forecast the expected Service QoS for a given interval of time, using the forecast model. This is then used to identify any possible QoS issues in any of the microservices. Second, it uses the context data to gather specific information (refer Section 7.2.2 (microservice level). It combines these data to identify the need for adaptation and triggers the decision-maker if an adaptation need arises.

vi) *Decision Maker* is responsible for identifying the best adaptation technique based on the information from the context analyzer. It uses a set of adaptation techniques that consists of adaptation options, such as dynamically scaling microservices, auto-rollback and restarting microservices, etc. These techniques can also be combined to form more complex *strategies*. In particular, to this end, we can exploit the combination of Q-learning and quantitative verification as defined in Chapter 6 or AI planning as defined by Bucchiarone [183]. For instance, in the (microservice level) scenario in Section 7.2.2, the *Context Analyzer* can trigger the adaptation based on the forecasted response time of the booking service and context data. Based on this, a new instance can be added. This decision is then communicated to the *Adaptation Initiator*.

vii) *Adaptation Initiator* acts as a bridge between the decision-maker and the *Management Infrastructure layer*, by forwarding the adaptation request to the higher layer.

4) Application Infrastructure Layer: Its purpose is to execute application-level adaptation based on their goals. In fact, some of the functionalities provided by the application are specified only as abstract goals (e.g., an application-level scenario in Section 7.2.2) that can be dynamically refined at runtime, through a composition of microservices whose execution allows users to achieve the goals. It consists of three main components:

i) *User Goal Parser:* it parses the abstract goals and translates them to the format as required by the *Service Composer*. Goals can be specified through various goal models available in the literature, for instance, the one used in [184]. ii) *Service Composer:* it uses the QoS data/forecasts available from the lower layer to decide on the best composition of microservices, driven by the abstract goals from the user goal parser. The composition can then be performed, for instance, by using an AI planning method as suggested by De Sanctis et al. [184]. The instance selection of such composed services can be performed using a combination of machine learning techniques as described in Section 7.1. The identified composition of microservices is sent back to the user application, which then uses the *API gateway* to invoke the respective microservices. For those scenarios where the service composition is not required, e.g., *login* operation, the user request is directly routed to the *API Gateway*. iii) *API Gateway:* it performs the routing of the requests from the user to the corresponding instances of the microservices through the service discovery component.

The approach can be realized by using a combination of various technologies. The message broker can be implemented using *Apache Kafka* [158]. For implementing the service mesh, technologies like *Istio* can be used [185]. The microservices can be deployed and managed by building them as *docker* containers [186]. Further for implementing the different data stores in the *Adaptation Infrastructure Layer*, *Elasticsearch* can be used [160]. The metrics collected from Istio regarding the QoS of the microservices, as well as the QoS of IoT devices, can be visualized using Grafana or Kibana [163]. Finally, as in the previous chapter, machine learning models can be developed using Keras [162].

In this section, We proposed a self-adaptive architecture for microservice-based IoT systems to handle the different types of self-adaptation challenges faced by these classes of systems. Further concrete realizations and validations are required to show the effectiveness and efficiency of the architecture. The approach uses the IoT system's data as the primary driver for adaptation and uses different machine learning techniques to ensure that the architecture improves over a while.

To summarize, the chapter's main focus was to tackle RQ4 and describe how the overall approach presented in this thesis can be generalized to more generic systems. Towards this, we applied the techniques presented in this thesis to solve two challenges in microservice-based software systems i) Context-aware service discovery using machine learning and ii) developing an architecture for enabling self-adaptation in MSA-IoT systems. As regards the former challenge, we proposed a novel context aware machine learning-driven service discovery for microservice based architecture. It was achieved by extending the core parts of the approach presented in Chapters 4 and 5. The later challenge was handled by developing a data-driven self-adaptive architecture for managing the different types of adaptation challenges in MSA-IoT systems.

Chapter 8

Conclusions and Future Work

"We can only see short distance ahead, but we can see plenty there that needs to be done" – Alan Turing

8.1 Conclusions

In this thesis, we have taken into account the different architecting challenges faced by modern software systems with a focus on IoT systems. We also elaborated on how these challenges lead to various types of uncertainties, affecting the overall quality of service (QoS). The field of self-adaptive systems has been focusing on addressing these challenges. However, we described the different challenges within the research domain for applying this technique to IoT systems or any modern software system in general. To address these challenges, we came up with an overall goal of developing a data-driven approach that, when applied to a software system with QoS constraints, allows it to automatically learn and improve the architecture by leveraging the different types of data generated using machine learning techniques. This overall goal was further broken down into four main research questions:

RQ1. How to perform effective and efficient proactive adaptation using machine learning techniques?

We addressed this question by developing a proactive machine learning-driven approach for architecting self-adaptive IoT systems by extending the traditional MAPE-K loop. The approach was used to improve the IoT system's energy efficiency by proactively forecasting the expected energy consumption using deep neural networks. We also performed evaluations to demonstrate how such techniques can accurately generate short-term and long-term forecasts and offer higher energy savings than their reactive counterparts.

Further, we also extensively evaluated the choice of adaptation and prediction intervals in a proactive setup.

RQ2. How can machine learning be used to continuously improve the adaptation process?

As an answer to this question, we extended the MAPE-K based proactive approach to support decision making using model-free reinforcement learning techniques. We showed how such an approach could perform adaptation by handling multiple QoS goals. The approach enabled the IoT system to optimize energy as well as data traffic consumption. We further demonstrated how a combination of machine learning techniques in a proactive adaptation setup enables the system to learn from the feedback of the succeeding forecasts and continuously improve the architecture.

RQ3. How to guarantee the quality of the adaptation performed by such a machine learning process and how can such guarantees in turn help the machine learning process?

We leveraged the use of quantitative verification techniques to address this question. To realize this, We integrated the proactive decision-making approach with quantitative verification, in particular probabilistic model checking. The approach enabled the IoT system to optimize energy and data traffic through proactive adaptations supported with qualitative guarantees. The approach provided quantitative guarantees to the machine learning process and enabled the learning to converge towards optimal adaptations. This was made possible by the two-way feedback mechanism from the verification process and succeeding forecasts after each adaptation.

RQ4. How can the approach be generalized to other class of software systems?

We first extended the approach of combining proactive forecasts with model-free reinforcement learning to perform service discovery in microservice-based architectures to address this question. The approach was applied to a prototype microservice application. The evaluation demonstrated how such an approach could perform better than state-of-the-art approaches. Further, we also extended some of the techniques used to develop a data-driven self-adaptive architecture for microservice-based IoT systems.

To summarize, Figure 8.1 represents the activity diagram of the overall approach presented in this thesis. The highlighted area represents the key contributions of the thesis. As it can be seen given an IoT system, our approach continuously monitors the QoS data as well as the context data. Based on the data, the approach: i) proactively forecasts the expected QoS as well as the expected context data using deep neural networks (Chapter 4); ii) selects the best adaptation strategy using reinforcement learning techniques (Chapter 5); iii) verifies the selection using quantitative verification techniques (Chapter 6); iv) uses the feedbacks of the verification process as well as the feedbacks of the

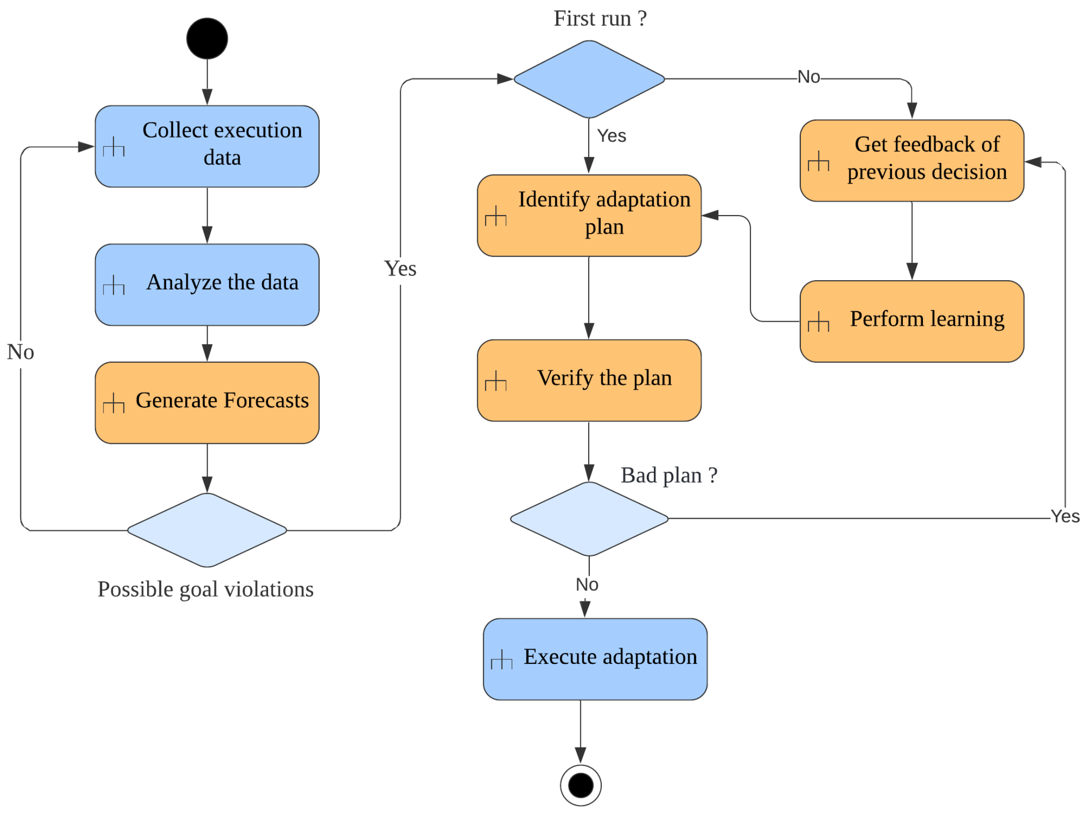


FIGURE 8.1: Overall Approach (Highlighted activities denote the thesis contributions)

forecast in successive adaptation intervals to continuously improve the decision-making process (Chapter 6). Applying the approach to the microservices domain resulted in a novel approach for performing machine learning-driven context-aware service discovery in microservice architectures (Chapter 7). Further, applying the approach to the domain of MSA-IoT resulted in the development of a novel data-driven self-adaptive architecture for MSA-IoT systems (Chapter 7).

Therefore, we advance the state-of-the-art by developing a data-driven approach for architecting self-adaptive systems, which in principle shifts focus from self-adaptive architectures to self-learning architectures. It achieves this using machine learning techniques to leverage the different types of data generated to ensure that the architecture learns from every adaptation and improves its architecture over time, thereby improving the overall QoS.

8.2 Future Work

We believe that what we have presented in this thesis is just a starting point and shall pave the way for more research in this direction to build self-learning software architectures just like self-driving cars. There are many more works that need to be done concerning different parts of the approach in the future. These include but not limited to:

Effective use of long-term forecasts: We have shown in Chapter 4, how we can use techniques like LSTM to generate long term forecasts to prediction horizon as long as 30-time steps. However, we also saw that the adaptation performed with such a long horizon might not be optimal. Towards, this we are working towards extending the three-layer approach proposed by Kramer et al. [52] with support for long-term forecasts and short-term forecasts. In this manner, the system will have the ability to adapt as per the short-term goals and long-term goals.

Evaluation with more deep learning techniques: The field of deep learning has been expanding very rapidly, and recently techniques like Convolution neural networks [187], LSTM with convolution layer [188], Resnet [189], etc. have been used for performing time-series forecasting. However still, most of them have been applied to univariate time-series data. Towards this we plan to expand our evaluations to perform comparisons (presented in Chapter 4), with these techniques.

Adaptations using fine-grained topologies: In Chapters 5 and 6, we were adapting the IoT system using three different architectural patterns. As future work, we plan to consider more fine-grained patterns instead of the more general, coarse-grained patterns that we are currently using. This will enable us to explore a much richer solution space, in which systems exhibit hybrid topologies with respect to the patterns considered (e.g., some IoT devices might prefer to perform computation in the edge, while others may prefer to use an intermediary controller node or a central controller), potentially leading to closer-to-optimal solutions with respect to the existing approach.

Scalability of the machine-learning aided verification: One of the immediate research avenues that we plan to explore with respect to the approach presented in Chapter 6, is scalability. This will entail considering more QoS parameters and patterns, extending and validating the adaptation approach by identifying more patterns and factoring into a learning-aided verification problem. This also involves taking into account the trade-offs between multiple QoS parameters.

Robustness of the approach: Our approach currently assumes that forecasts can be produced with acceptable accuracy. We plan on assessing the robustness of our approach with respect to uncertainties that affect the accuracy of the forecasts employed for decision making, studying how the quality of the decisions degrades with higher levels of uncertainty and devising extensions to mitigate the degradation of decision quality. Further, we also plan to extend the approach with reactive adaptation capabilities to ensure stability in the event of a sudden unforeseen situation.

Extending the verification-aided machine learning to more generic systems: In Chapter 6, we presented how the use of quantitative verification techniques along with machine learning, can enable the system to converge towards optimal adaptations. As our next step, we plan to extend this approach to generic systems like cloud based systems, microservices, etc. As a first step, we are in the process of applying learning-aided verification to the SWIM exemplar [190]. Initial results are promising, and we believe this would pave the way for more advanced research in this area.

Machine learning-driven service discovery for large scale systems: We plan to apply the machine learning-driven service discovery mechanism presented in Chapter 7 to microservice systems with a larger number of instances. This would mean that the instance selection algorithm based on reinforcement learning would have to handle larger action space complexity. Towards this, we plan to leverage the use of deep Q-learning [181]. Further, we also plan to explore the possibility of using transfer learning [191] to make the approach more robust across different classes of systems.

Realization of Data-driven Self-adaptive Architecture for MSA-IoT: The architecture presented in Chapter 7 (Section 7.2) was more a conceptual architecture and lacked concrete realization. We are currently in the process of realizing the architecture by applying into the microservice-based NdR case study presented in Chapter 7, Section 7.2.2. The approach's effectiveness shall be measured based on the accuracy of predictions made, energy saved for IoT devices, the degree of user goals achieved, the average response time of microservices, etc. The efficiency shall be measured based on the quality of the adaptations performed. The implementation shall also be extended to consider different challenges such as DevOps integration, testing, etc. as mentioned in [182].

Appendix A

Technology Stack

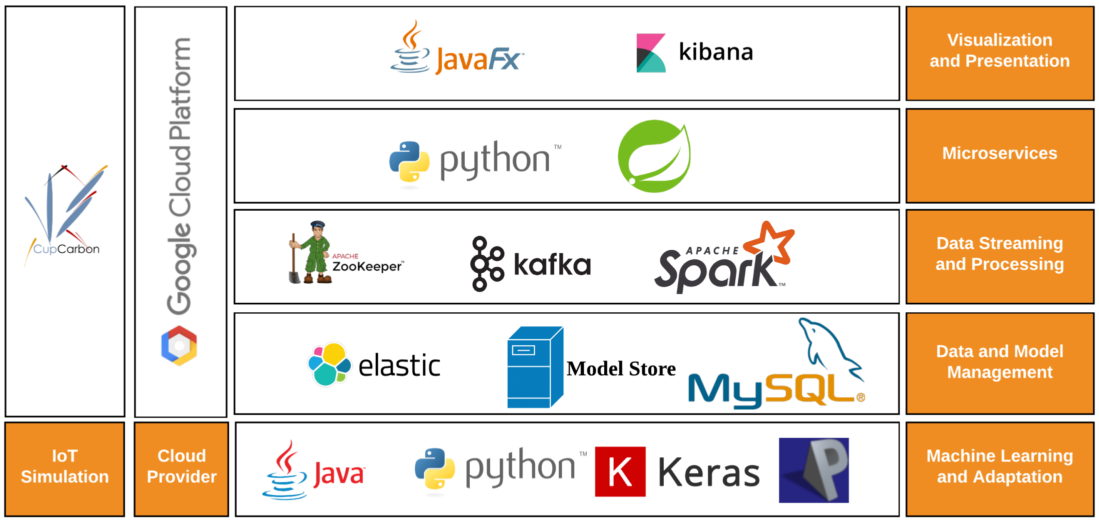
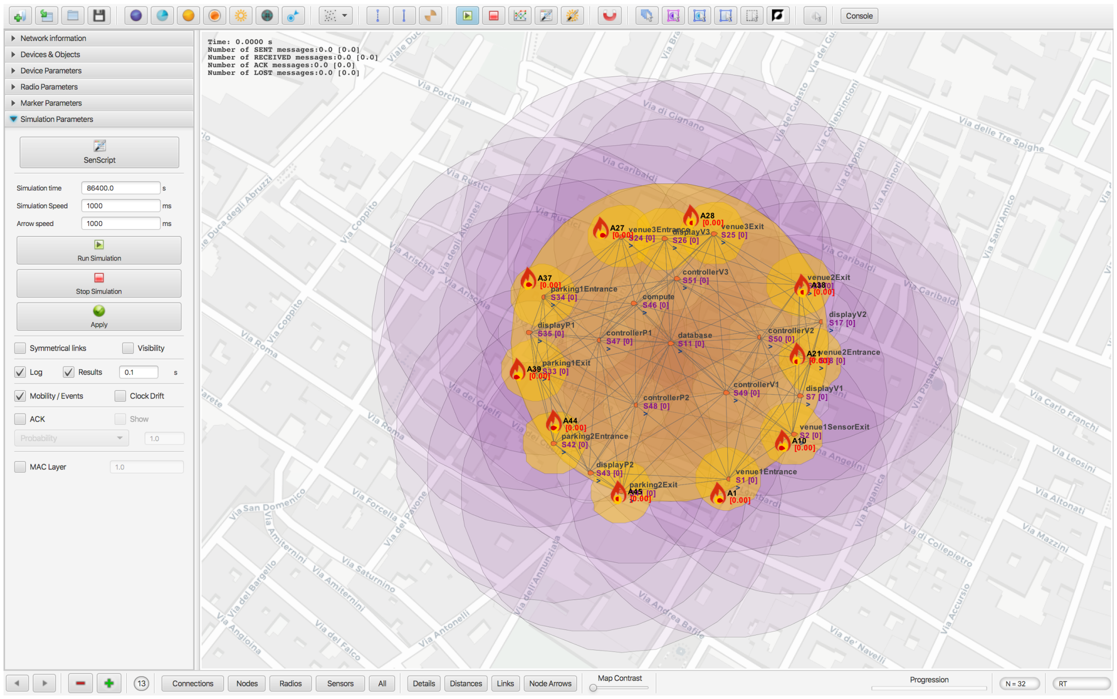


FIGURE A.1: Overall Technology Stack

This section provides the details of the various technologies used to realize the different parts of the approach presented throughout this thesis. Figure A.1 shows the technology stack diagram consisting of the different technologies used for the implementation of various parts of the approach presented in this thesis. The technologies used have been categorized into different parts based on their use.

IoT Simulation

We used *CupCarbon Simulator* to simulate the IoT system of NdR to evaluate our approaches further. CupCarbon is an open-source IoT and Wireless Sensor Networks (WSN) simulator [153] [154]. It is especially considered as an excellent tool for simulating the energy consumption of IoT systems [155] [156] [157]. The tool can be used for both educational and scientific purposes. It allows users to design IoT networks with sensors and other IoT components. The tool also provides an open street map, which allows

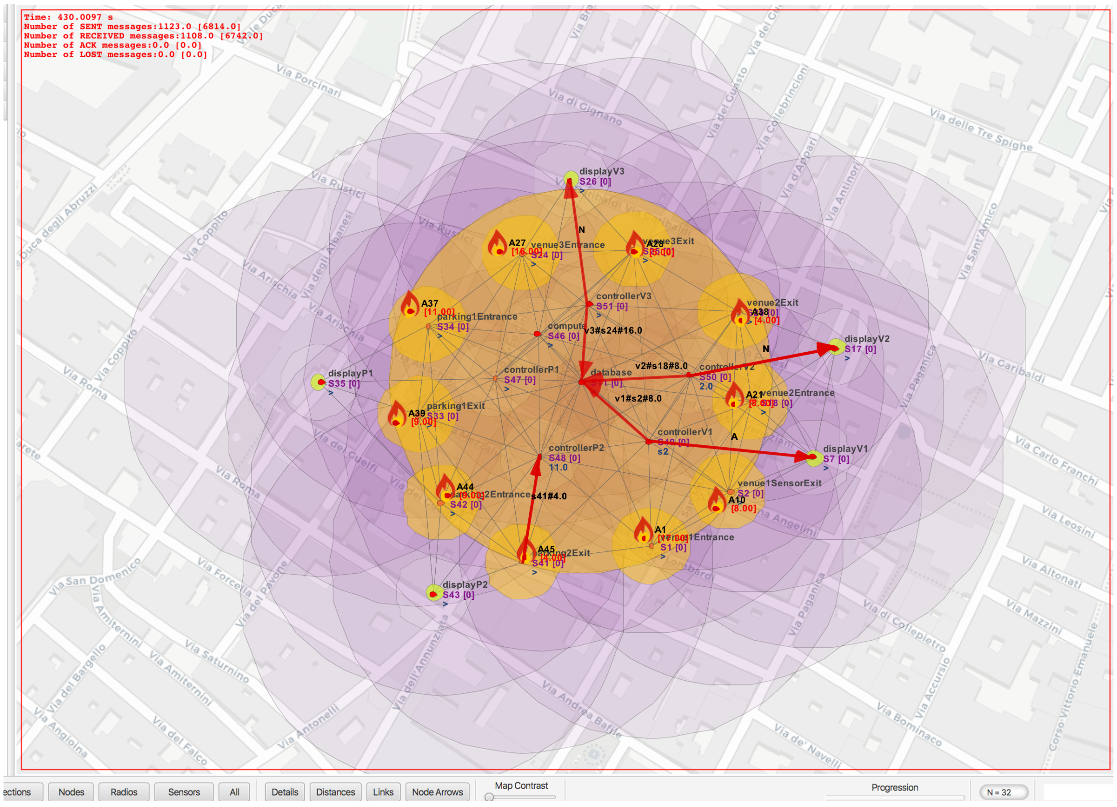


```
Time: 0.0000 s
Number of SENT messages:0.0 [0.0]
Number of RECEIVED messages:0.0 [0.0]
Number of ACK messages:0.0 [0.0]
Number of LOST messages:0.0 [0.0]
```

Simulation time: 86400.0 s
Simulation Speed: 1000 ms
Arrow speed: 1000 ms
[ ] Symmetrical links [ ] Visibility
[✓] Log [✓] Results 0.1 s
[✓] Mobility / Events [ ] Clock Drift
[ ] ACK [ ] Show
Probability 1.0
[ ] MAC Layer 1.0

Figure A.2: Screenshot of the NdR case implemented in CupCarbon

the designers to design, prototype the IoT system, and deploy the sensors directly on different locations on the map. This network can then be simulated for both educative and analytic purposes. It comes with a scripting language, *senscript*, which allows users to program and configure each of the sensor nodes individually. It allows users/designers to create a scenario, deploy the sensor network and simulate different parameters such as energy consumption, data traffic, battery level, etc. of the sensor components of a given IoT system. Figure A.2 shows the screenshot of the CupCarbon simulator with the NdR case. The image in the background depicts the map of L'Aquila. The sensor nodes are placed in locations, as in the case of L'Aquila. As seen in the image, the tool's left side panel provides different options to configure the simulation. These include: i) assigning each sensor with a script (based on senscript); ii) configuring their radio and communication channels (like WiFi, LoRa, etc.); iii) assigning energy consumption models (CupCarbon provides standard energy consumption models[153] for defining the energy consumption models for each sensor nodes); iv) defining initial battery level (in joules); v) defining the data distribution for each sensor nodes, it also provides a mechanism to upload custom data (we use this facility to upload the data for the sensors which were created using a Poisson distribution based on the real observations from the case study), and the list goes on. Figure A.3 shows the screenshot of the CupCarbon tool during the simulation of the NdR case. The directed red arrows indicate the flow of data from one sensor node to another, the arrow also contains a value indicating the data being sent. Further, CupCarbon allows generating code for hardware platforms such as Arduino/XBee with a single click. CupCarbon also stores the simulations' data in log



```
Time: 430.0097 s
Number of SENT messages:1123.0 [6814.0]
Number of RECEIVED messages:1108.0 [6742.0]
Number of ACK messages:0.0 [0.0]
Number of LOST messages:0.0 [0.0]
```

FIGURE A.3: Simulation view of NdR in CupCarbon

files, which may be used for further analysis. During the simulation, the logs generated are processed in real-time in our approach for providing adaptation capabilities.

One of the main parts of the approach presented in this thesis, in particular, the one shown in Chapter 5, is also available as a tool, Archlearner[1] [3]. The tool aids machine learning-driven proactive architectural adaptation. It uses the three self-adaptation patterns [56] to perform the adaptation. The tool leverages machine learning techniques implemented on an enterprise-grade big data stack. It enables the IoT system to i) automatically identify the need for adaptation at an early stage; ii) perform automated decision making for generating the best adaptation strategy; iii) gather the feedback of the selected decision for continuous improvement. It also provides a visual interface for i) defining the QoS thresholds; ii) visualizing the adaptation process; iii) specifying the forecasts and related adaptation configurations; iv) visualizing real-time QoS; v) adding/editing the machine learning related configurations.

Cloud Provider

Some parts of the approach, particularly those in Chapters 6 and 7 were implemented on *Google Cloud Platform*. Google Cloud also supported these researches under the

[1]https://mysat.gitlab.io/archlearner-web/

Google Cloud research credits program.[2]. Out of the many services offered by Google Cloud, we made use of the Infrastructure as a Services (Iaas) to launch our custom Virtual Machines (Compute Engine Service[3]). These were used to deploy the different components of the approach presented in Chapter 6. The virtual machines allowed us to easily manage the generated machine learning models and host web-services and CupCarbon simulator without any issues. Further, we launched two virtual machines in different geographical zones and different hardware and CPU configurations to simulate the contextual changes for the approach presented in Chapter 7.

Machine Learning and Adaptation

For developing the different machine learning models (LSTM models) in the approach, we made use of the *Keras* deep learning API with *Tensorflow* backend (Tensorflow version 1.15. However, the model building process can easily be extended to the current version of 2.0). Keras is one of the most popular and widely used deep learning APIs. Due to the native support for machine learning, Python was used for the implementation. Keras provides stable APIs for developing machine learning models based on most of the state-of-the-art deep learning algorithms. In particular, for developing the LSTM models, we made use of the *Sequential* model[4] library of the Keras API. It allows one to define and train LSTM models easily. For example, a one-line command, *model = Sequential()* allows us to define a sequential model (Indeed, the libraries need to be imported). Further *model.add(LSTM(294, input_shape=(10, 10]),return_sequences=False))* defines an LSTM model with with 294 LSTM units in the hidden layer and input shape of [10,10]. This implies a multi-variate input series with lag value, 10, and a number of features, 10. Further, we exploit the different features of Keras to develop other machine learning models. These were deployed in machines as a cronjob to perform the training periodically. After every training cycles, the models that are validated are versioned and stored in the Model store (see Figure A.1)

For executing the adaptations, we developed a Python script that sends the decision made (as a pattern to be used in the case of the approach presented in Chapter 5 and 6 and the sensor frequency for the approach presented in Chapter 4) to the CupCarbon simulator by writing the decision in a configuration JSON file. We also developed a web-service using the Python Tornado framework[5]. Further, we modified the source code of CupCarbon using Java to continuously read the configuration JSON file by continually sending a REST request to the web service. In this manner, any decision made is immediately used to reconfigure/adapt the architecture.

[2]https://edu.google.com/programs/credits/research/
[3]https://cloud.google.com
[4]https://keras.io/guides/sequential_model/
[5]https://www.tornadoweb.org/en/stable/

As stated in Chapter, 6, we used PRISM[6] model checker for implementing the verified to verify the decision produced by the q-learning algorithm. The biggest challenge, in this case, on the implementation side, was to integrate the machine learning process (implemented in Python) with the model-checking process in run-time and perform continuous verification. The model checker is written as a separate component in Java. To facilitate integration with Python, this was wrapped under a custom class importing Jpype[7] libraries to support the execution of the model checker via socket call. This was further deployed as an executable jar in a server. The decision-making process used a custom Jpype based python script to communicate with the model checker via socket call.

Data and Model Management

For the works mentioned in Chapter4, 5 and 6, *Elasticsearch* is used to store the QoS data as well as the sensor data (behavioral data, in case of Chapter 6). Elasticsearch is an open-source distributed REST-based search and analytics engine which is commonly used for performing log analytics, full-text search, security intelligence, etc. It is also widely known for its use as a data store for time-series data. Due to this property, we use it to store the different time-series data used by our approach. Elasticsearch uses the concept of *index* and *documents*. The index is akin to a table in a traditional database, and a document is similar to a row in a table. Each document is nothing but a JSON consisting of multiple fields. For storing the energy and data traffic consumed by the system, we created an index, "adaptive" with document types "energy" and "traffic." We created an index "sensor" with document type "data" for storing the sensor data. The id was set to auto-increment, and we kept adding the data received. This data was further used to visualize the near real-time energy, data traffic consumption, and sensor data (of different sensors to get occupancy in parking lots, venues, etc.). It is also used for further off-line training of the machine learning models to keep improving the models with more and more data during the build phase, as defined in Section 4.2.2. For storing and versioning the machine learning models, we used the traditional file-based storage as *Model store*, where the machine learning models were structured in different folders depending on their type. For instance, versions of energy forecast models were stored under the folder "energy" then traffic models under the folder "traffic", so on and so forth. In future versions, we intend to use *mlflow*[8] for handling the versioning and deployment of machine learning models. *MySQL*[9] was used as the database for the microservices in the prototype application used in Chapter 7. MySQL is one of the most popular open-source database platforms, and it provided all the required functionalities

[6]http://www.prismmodelchecker.org
[7]https://jpype.readthedocs.io/
[8]https://mlflow.org
[9]https://www.mysql.com

for managing the data of the prototype application. We use separate databases to align with the core principle of microservices (refer to Section 2.4). Since all the data used were in a relational form, there was no need to use an additional NoSQL database.

Data Streaming and Processing For streaming the QoS logs and the sensor data logs from CupCarbon to different components, we made use of *Apache Kafka*[10]. At the core, Apache Kafka is an open-source distributed event streaming platform. In other words, Kafka is a distributed publish-subscribe system that is scalable, robust, and fast by design. Kafka is one of the most widely used platforms for building real-time data pipelines to transfer data between heterogeneous components or for handling streaming data. It was first developed at LinkedIn to handle the issues associated with real-time notifications [159]. Like many publish-subscribe messaging systems, Kafka maintains feeds of messages in topics. Producers write data to topics, and consumers read from topics. Since Kafka is a distributed system, topics are partitioned and replicated across multiple nodes. It provides API for all major languages like Python, Java, Scala, etc. We used the python API, *kafka-python*[11] for the implementation. To facilitate real-time streaming of QoS and sensor data logs, we create three topics in Kafka, namely "energy", "traffic", and "sensor". Further, we write data producers using a simple Python script to stream data from CupCarbon to the corresponding Kafka topic, which is then pushed/pulled by a consumer. As stated in Chapters 4, 5 and 6, we used Kafka based consumer implemented using Python to consume the message ingested in the different Kafka topics. The consumer further aggregates the data as required for performing the forecasts. For example, suppose the lag, l, is 10 for energy forecasts. In that case, the consumer keeps aggregating the data in an array until it has reached 10 data points, and this is further sent to the machine learning components to make the prediction. Once sent, the data is flushed out (based on decision intervals) to handle the new data set.

For the tool, Archlearner [3], we made use of *Apache Spark* instead of *Apache Kafka* for processing and aggregating the data (the role of the consumer). Apache Spark is an open-source real-time distributed analytics and big data processing engine. It supports real-time processing of batch data via dedicated streaming API, known as Spark Streaming[12]. Spark, written in Scala, also provides API's to support different languages like Python, Java, etc. We used PySpark[13], (the Spark API for Python) to implement the real-time data processing module of the tool. We wrap this using a python module that executes spark in a streaming context. The spark streaming is set to consume the message from the topic "sensor", "energy", "traffic", etc. with a

[10]https://kafka.apache.org
[11]https://pypi.org/project/kafka-python/
[12]https://spark.apache.org/docs/latest/streaming-programming-guide.html
[13]https://spark.apache.org/docs/latest/api/python/index.html

batch interval based on the decision interval of adaptation as well as the lag value for performing the forecasts. It first applies a map operation with a split of the raw data into features required by the machine learning process. It then converts the data to time-series data, and this data is sent to Elasticsearch for storage. This batch is also appended to a pandas data frame. The data frame is aggregated for 1-minute intervals to generate the aggregated lag data at the decision period interval. This aggregated lag data is used for prediction.

To manage Kafka and Spark's different instances, we make use of *Apache Zookeeper*[14]. Zookeeper is an open-source centralized coordination service for enabling better coordination in a distributed computing environment. It is responsible for maintaining the configurations of the list of active nodes, provide naming service (which allows nodes to lookup active nodes in the cluster), better synchronization support, etc. In our implementation, the zookeeper is responsible for maintaining active Kafka and Spark nodes, which are then used by the Python modules to establish connections.

Microservices

The implementation of microservices for the approach presented in Chapter 7, was achieved using the *Tornado web framework*[15] (Python) and *Java Spring*[16]. Tornado is a web framework for developing Python-based web services. It provides an in-built web server and supports asynchronous non-blocking calls. Due to this reason, it can be easily used at scale to support a bulk amount of requests. Java Spring, on the other hand, is a framework for developing enterprise applications in Java. It provides a model view controller (MVC) framework for easily developing and marinating web applications. The microservices implementing the different functionalities of the case study mentioned in Section 7.1.2 has been developed using the Java Spring framework. Further, we replicated this to multiple instances by creating multiple copies of this web service deployed on different ports. Every microservice requiring data access were connected to their own MySQL database. The service discovery itself was implemented using Java Spring, and service for building and managing machine learning models for the QoS forecasts was implemented using the Python Tornado framework.

Visualization and Presentation

For the visualization and presentation, we made use of *JavaFX*[17] and *Kibana*[18]. JavaFX provides a platform for creating rich GUI applications based on Java. It is mainly used for creating desktop applications and provides extensive support for custom styling

[14]https: //zookeeper.apache.org
[15]https://www.tornadoweb.org/en/stable/
[16]https://spring.io
[17]https://openjfx.io
[18]https://www.elastic.co/kibana

of the user interfaces. We use JavaFX for developing the user interface (UI) of our tool Archlearner. The developed UI allows users to visualize the real-time energy and data traffic graphs and the different machine learning models' accuracy plots. The UI provides detailed views on how the adaptation is performed. It also allows stakeholders to define the respective configurations required for performing the adaptation and the QoS thresholds that the system has to maintain.

Kibana is an open-source visualization plugin that allows us to visualize and navigate the data on the elastic search cluster. It also provides dedicated support for visualizing time-series data. We use Kibana for creating multiple dashboards, based on the real-time QoS data from Elasticsearch. The Elasticsearch data is connected to Kibana by mapping the different Elasticsearch indexes, ”energy”, ”traffic”,”adaptive”, etc. to the Kibana cluster. It supports creating multiple dashboards, and this option is used to create near real-time dashboards for the energy consumption data of different components in the system as well as dashboard for real-time sensor data and data traffic information.

The working demo of the tool, Archlearner, can be found here[19]

[19] https://www.youtube.com/watch?v=BrRsQ6PYyVY

Bibliography

[1] Henry Muccini and Karthik Vaidhyanathan. A machine learning-driven approach for proactive decision making in adaptive architectures. In *2019 IEEE International Conference on Software Architecture Companion (ICSA-C)*, pages 242–245. IEEE, 2019.

[2] Henry Muccini and Karthik Vaidhyanathan. Pie-ml: A machine learning-driven proactive approach for architecting self-adaptive energy efficient iot systems. Technical report, TRCS: 002/2020, University of L'Aquila, Italy, 2020. URL https://tinyurl.com/y98weaat.

[3] Henry Muccini and Karthik Vaidhyanathan. Archlearner: leveraging machine-learning techniques for proactive architectural adaptation. In *Proceedings of the 13th European Conference on Software Architecture-Volume 2*, pages 38–41, 2019.

[4] Henry Muccini and Karthik Vaidhyanathan. Leveraging machine learning techniques for architecting self-adaptive iot systems. In *2020 IEEE International Conference on Smart Computing (SMARTCOMP)*, pages 65–72. IEEE, 2020.

[5] Javier Cámara, Henry Muccini, and Karthik Vaidhyanathan. Quantitative verification-aided machine learning: A tandem approach for architecting self-adaptive iot systems. In *2020 IEEE International Conference on Software Architecture (ICSA)*, pages 11–22. IEEE, 2020.

[6] Henry Muccini Mauro Caporuscio, Marco De Toma and Karthik Vaidhyanathan. A machine learning-driven approach to service discovery for microservice architectures. Technical report, TRCS: 003/2020, University of L'Aquila, Italy, 2020. URL https://tinyurl.com/y67krcn9.

[7] Martina De Sanctis, Henry Muccini, and Karthik Vaidhyanathan. Data-driven adaptation in microservice-based iot architectures. In *2020 IEEE International Conference on Software Architecture Companion (ICSA-C)*, pages 59–62. IEEE, 2020.

[8] Dewayne E Perry and Alexander L Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software engineering notes*, 17(4):40–52, 1992.

[9] Mary Shaw, David Garlan, et al. *Software architecture*, volume 101. prentice Hall Englewood Cliffs, 1996.

[10] David Garlan. Software architecture: a travelogue. In *Future of Software Engineering Proceedings*, pages 29–39. 2014.

[11] Henry Muccini, Patricia Lago, Karthik Vaidyanathan, Francesco Osborne, and Eltjo Poort. The history of software architecture-in the eye of the practitioner. *arXiv preprint arXiv:1806.04055*, 2018.

[12] McKinsey Analytics. The age of analytics: competing in a data-driven world, 2016.

[13] Ilias Gerostathopoulos, Marco Konersmann, Stephan Krusche, David I Mattos, Jan Bosch, Tomas Bures, Brian Fitzgerald, Michael Goedicke, Henry Muccini, Helena H Olsson, et al. Continuous data-driven software engineering-towards a research agenda: Report on the joint 5th international workshop on rapid continuous software engineering (rcose 2019) and 1st international works. *ACM SIGSOFT Software Engineering Notes*, 44(3):60–64, 2019.

[14] Forbes Bernard Marr. How much data do we create every day? the mind-blowing stats everyone should read, May 2018. URL https://www.forbes.com/sites/bernardmarr/2018/05/21/how-much-data-do-we-create-every-day-the-mind-blowing-stats-everyone-should-read/#6db34ea60ba9.

[15] Ethem Alpaydin. *Introduction to machine learning*. MIT press, 2009.

[16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

[17] Standford Business. Andrew ng: Why ai is the new electricity, March 2017. URL https://www.gsb.stanford.edu/insights/andrew-ng-why-ai-new-electricity.

[18] Gartner. Gartner predicts the future of ai technologies, October 2019. URL https://www.gartner.com/smarterwithgartner/gartner-predicts-the-future-of-ai-technologies/.

[19] Jan Bosch, Ivica Crnkovic, and Helena Holmström Olsson. Engineering ai systems: A research agenda. *arXiv preprint arXiv:2001.07522*, 2020.

[20] Zhiyuan Wan, Xin Xia, David Lo, and Gail C Murphy. How does machine learning change software development practices? *IEEE Transactions on Software Engineering*, 2019.

[21] Tingting Bi, Peng Liang, Antony Tang, and Chen Yang. A systematic mapping study on text analysis techniques in software architecture. *Journal of Systems and Software*, 144:533–558, 2018.

[22] Zainab H Ali, Hesham A Ali, and Mahmoud M Badawy. Internet of things (iot): Definitions, challenges and recent research directions. *International Journal of Computer Applications (0975–8887) Volume*, 2015.

[23] A. Taivalsaari and T. Mikkonen. A roadmap to the programmable world: Software challenges in the iot era. *IEEE Software*, 34(1):72–80, Jan 2017. ISSN 0740-7459. doi: 10.1109/MS.2017.26.

[24] Sarra Hammoudi, Zibouda Aliouat, and Saad Harous. Challenges and research directions for internet of things. *Telecommun. Syst.*, 67(2):367–385, February 2018. ISSN 1018-4864. doi: 10.1007/s11235-017-0343-y. URL https://doi.org/10.1007/s11235-017-0343-y.

[25] Jayavardhana Gubbi, Rajkumar Buyya, Slaven Marusic, and Marimuthu Palaniswami. Internet of things (iot): A vision, architectural elements, and future directions. *Future generation computer systems*, 29(7):1645–1660, 2013.

[26] Bob Familiar. Iot and microservices. In *Microservices, IoT, and Azure*, pages 133–163. Springer, 2015.

[27] Daniel Minoli and Benedict Occhiogrosso. Blockchain mechanisms for iot security. *Internet of Things*, 1:1–13, 2018.

[28] John A Stankovic. Research directions for the internet of things. *IEEE Internet of Things Journal*, 1(1):3–9, 2014.

[29] Mazeiar Salehie and Ladan Tahvildari. Self-adaptive software: Landscape and research challenges. *ACM Trans. Auton. Adapt. Syst.*, 4(2):14:1–14:42, May 2009. ISSN 1556-4665. doi: 10.1145/1516533.1516538. URL http://doi.acm.org/10.1145/1516533.1516538.

[30] Henry Muccini, Mohammad Sharaf, and Danny Weyns. Self-adaptation for cyber-physical systems: a systematic literature review. In *Proceedings of the 11th international symposium on software engineering for adaptive and self-managing systems*, pages 75–81. ACM, 2016.

[31] Christian Krupitzer, Felix Maximilian Roth, Sebastian VanSyckel, Gregor Schiele, and Christian Becker. A survey on engineering approaches for self-adaptive systems. *Pervasive and Mobile Computing*, 17:184–206, 2015.

[32] Danny Weyns. Software engineering of self-adaptive systems: an organised tour and future challenges. *Chapter in Handbook of Software Engineering*, 2017.

[33] Yuriy Brun, Giovanna Di Marzo Serugendo, Cristina Gacek, Holger Giese, Holger Kienle, Marin Litoiu, Hausi Müller, Mauro Pezzè, and Mary Shaw. Engineering self-adaptive systems through feedback loops. In *Software engineering for self-adaptive systems*, pages 48–70. Springer, 2009.

[34] Gabriel A Moreno, Javier Cámara, David Garlan, and Bradley Schmerl. Proactive self-adaptation under uncertainty: a probabilistic model checking approach. In *Proceedings of the 2015 10th joint meeting on foundations of software engineering*, pages 1–12, 2015.

[35] Konstantinos Angelopoulos, Alessandro V Papadopoulos, Vítor E Silva Souza, and John Mylopoulos. Model predictive control for software systems with cobra. In *2016 IEEE/ACM 11th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, pages 35–46. IEEE, 2016.

[36] Yakov Zalmanovich Tsypkin and Zivorad Jezdimir Nikolic. *Adaptation and learning in automatic systems*, volume 73. Academic press New York, 1971.

[37] Dongsun Kim and S. Park. Reinforcement learning-based dynamic adaptation planning method for architecture-based self-managed software. In *2009 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, pages 76–85, May 2009. doi: 10.1109/SEAMS.2009.5069076.

[38] Tao Chen and Rami Bahsoon. Self-adaptive and sensitivity-aware qos modeling for the cloud. In *Proceedings of the 8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pages 43–52. IEEE Press, 2013.

[39] Frank D. Macías-Escrivá, Rodolfo Haber, Raul del Toro, and Vicente Hernandez. Self-adaptive systems: A survey of current approaches, research challenges and applications. *Expert Systems with Applications*, 40(18):7267 – 7279, 2013. ISSN 0957-4174. doi: https://doi.org/10.1016/j.eswa.2013.07.033. URL http://www.sciencedirect.com/science/article/pii/S0957417413005125.

[40] Rachel Courtland. Bias detectives: the researchers striving to make algorithms fair. *Nature*, 558(7710):357–357, 2018.

[41] Ninareh Mehrabi, Fred Morstatter, Nripsuta Saxena, Kristina Lerman, and Aram Galstyan. A survey on bias and fairness in machine learning. *arXiv preprint arXiv:1908.09635*, 2019.

[42] The Newyork Times. Tesla autopilot system found probably at fault in 2018 crash, February 2020. URL https://www.nytimes.com/2020/02/25/business/tesla-autopilot-ntsb.htm.

[43] MIT Technology Review. Ai is sending people to jail—and getting it wrong, January 2019. URL https://www.technologyreview.com/2019/01/21/137783/algorithms-criminal-justice-ai/:.

[44] Mohsen Marjani, Fariza Nasaruddin, Abdullah Gani, Ahmad Karim, Ibrahim Abaker Targio Hashem, Aisha Siddiqa, and Ibrar Yaqoob. Big iot data analytics: architecture, opportunities, and open research challenges. *IEEE Access*, 5: 5247–5261, 2017.

[45] Thomas M. Mitchell. *Machine Learning*. McGraw-Hill, Inc., New York, NY, USA, 1 edition, 1997. ISBN 0070428077, 9780070428072.

[46] Henry Muccini and Karthik Vaidhyanathan. Towards self-learnable software architectures. *ERCIM News*, 2020(122), 2020. URL https://ercim-news.ercim.eu/en122/special/towards-self-learnable-software-architectures.

[47] Peter Naur and Brian Randell, editors. *Software Engineering: Report of a Conference Sponsored by the NATO Science Committee, Garmisch, Germany, 7-11 Oct. 1968, Brussels, Scientific Affairs Division, NATO*. 1969.

[48] Robert Laddaga. Active software. In *Proceedings of the First International Workshop on Self-adaptive Software*, IWSAS' 2000, pages 11–26, Berlin, Heidelberg, 2000. Springer-Verlag. ISBN 3-540-41655-2. URL http://dl.acm.org/citation.cfm?id=375094.375105.

[49] Betty H. Cheng, Rogério Lemos, Holger Giese, Paola Inverardi, Jeff Magee, Jesper Andersson, Basil Becker, Nelly Bencomo, Yuriy Brun, Bojan Cukic, Giovanna Marzo Serugendo, Schahram Dustdar, Anthony Finkelstein, Cristina Gacek, Kurt Geihs, Vincenzo Grassi, Gabor Karsai, Holger M. Kienle, Jeff Kramer, Marin Litoiu, Sam Malek, Raffaela Mirandola, Hausi A. Müller, Sooyong Park, Mary Shaw, Matthias Tichy, Massimo Tivoli, Danny Weyns, and Jon Whittle. Software engineering for self-adaptive systems. chapter Software Engineering for Self-Adaptive Systems: A Research Roadmap, pages 1–26. Springer-Verlag, Berlin, Heidelberg, 2009. ISBN 978-3-642-02160-2. doi: 10.1007/978-3-642-02161-9_1. URL http://dx.doi.org/10.1007/978-3-642-02161-9_1.

[50] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, Jan 2003. ISSN 0018-9162. doi: 10.1109/MC.2003.1160055.

[51] D. Garlan, S. . Cheng, A. . Huang, B. Schmerl, and P. Steenkiste. Rainbow: architecture-based self-adaptation with reusable infrastructure. *Computer*, 37(10): 46–54, Oct 2004. doi: 10.1109/MC.2004.175.

[52] J. Kramer and J. Magee. Self-managed systems: an architectural challenge. In *Future of Software Engineering (FOSE '07)*, pages 259–268, May 2007. doi: 10. 1109/FOSE.2007.19.

[53] R. Calinescu, L. Grunske, M. Kwiatkowska, R. Mirandola, and G. Tamburrelli. Dynamic qos management and optimization in service-based systems. *IEEE Transactions on Software Engineering*, 37(3):387–409, May 2011. ISSN 0098-5589. doi: 10.1109/TSE.2010.92.

[54] Antonio Filieri, Henry Hoffmann, and Martina Maggio. Automated design of self-adaptive software with control-theoretical formal guarantees. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 299–310, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2756-5. doi: 10. 1145/2568225.2568272. URL http://doi.acm.org/10.1145/2568225.2568272.

[55] An architectural blueprint for autonomic computing. Technical report, IBM, June 2005.

[56] Angelika Musil, Juergen Musil, Danny Weyns, Tomas Bures, Henry Muccini, and Mohammad Sharaf. Patterns for self-adaptation in cyber-physical systems. In *Multi-disciplinary engineering for cyber-physical production systems*, pages 331–368. Springer, 2017.

[57] P. Louridas and C. Ebert. Machine learning. *IEEE Software*, 33(5):110–115, Sept 2016. ISSN 0740-7459. doi: 10.1109/MS.2016.114.

[58] Sotiris B Kotsiantis. Supervised machine learning: A review of classification techniques. 2007.

[59] Yoshua Bengio, Aaron Courville, and Pascal Vincent. Representation learning: A review and new perspectives. *IEEE transactions on pattern analysis and machine intelligence*, 35(8):1798–1828, 2013.

[60] Xiaojin Zhu. Semi-supervised learning literature survey. *Computer Science, University of Wisconsin-Madison*, 2(3):4.

[61] Richard S Sutton and Andrew G Barto. *Introduction to reinforcement learning*, volume 135. MIT press Cambridge, 1998.

[62] Luciano da Fontoura Costa and Gonzalo Travieso. Fundamentals of neural networks: By laurene fausett. prentice-hall, 1994, pp. 461, isbn 0-13-334186-0, 1996.

[63] Robert Hecht-Nielsen. Theory of the backpropagation neural network. In *Neural networks for perception*, pages 65–93. Elsevier, 1992.

[64] Léon Bottou. Stochastic gradient learning in neural networks. *Proceedings of Neuro-Nımes*, 91(8):12, 1991.

[65] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[66] Douglas M Hawkins. The problem of overfitting. *Journal of chemical information and computer sciences*, 44(1):1–12, 2004.

[67] Alex Graves. Supervised sequence labelling. In *Supervised sequence labelling with recurrent neural networks*, pages 5–13. Springer, 2012.

[68] Yoshua Bengio, Patrice Simard, and Paolo Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks*, 5(2):157–166, 1994.

[69] Sepp Hochreiter, Yoshua Bengio, Paolo Frasconi, Jürgen Schmidhuber, et al. Gradient flow in recurrent nets: the difficulty of learning long-term dependencies, 2001.

[70] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[71] Chris Chatfield. *Time-series forecasting.* Chapman and Hall/CRC, 2000.

[72] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4): 279–292, 1992.

[73] Abhijit Gosavi. Reinforcement learning: A tutorial survey and recent advances. *INFORMS Journal on Computing*, 21(2):178–192, 2009.

[74] Dimitri P Bertsekas and John N Tsitsiklis. *Neuro-dynamic programming.* Athena Scientific, 1996.

[75] Eyal Even-Dar and Yishay Mansour. Learning rates for q-learning. *Journal of Machine Learning Research*, 5(Dec):1–25, 2003.

[76] Marta Kwiatkowska, Gethin Norman, and David Parker. Stochastic model checking. In *International School on Formal Methods for the Design of Computer, Communication and Software Systems*, pages 220–270. Springer, 2007.

[77] Marta Kwiatkowska. Quantitative verification: models techniques and tools. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 449–458, 2007.

[78] Marie-Aude Esteve, Joost-Pieter Katoen, Viet Yen Nguyen, Bart Postma, and Yuri Yushtein. Formal correctness, safety, dependability, and performance analysis of a satellite. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 1022–1031. IEEE, 2012.

[79] Husain Aljazzar, Manuel Fischer, Lars Grunske, Matthias Kuntz, Florian Leitner-Fischer, and Stefan Leue. Safety analysis of an airbag system using probabilistic fmea and probabilistic counterexamples. In *2009 Sixth International Conference on the Quantitative Evaluation of Systems*, pages 299–308. IEEE, 2009.

[80] Hans Hansson and Bengt Jonsson. A logic for reasoning about time and reliability. *Formal aspects of computing*, 6(5):512–535, 1994.

[81] William J Stewart. *Introduction to the numerical solution of Markov chains.* Princeton University Press, 1994.

[82] Adnan Aziz, Kumud Sanwal, Vigyan Singhal, and Robert Brayton. Verifying continuous time markov chains. In *International Conference on Computer Aided Verification*, pages 269–276. Springer, 1996.

[83] Christel Baier, Joost-Pieter Katoen, and Holger Hermanns. Approximative symbolic model checking of continuous-time markov chains. In *International Conference on Concurrency Theory*, pages 146–161. Springer, 1999.

[84] Marta Kwiatkowska. Advances in quantitative verification for ubiquitous computing. In *International Colloquium on Theoretical Aspects of Computing*, pages 42–58. Springer, 2013.

[85] James Lewis and Martin Fowler. Microservices, May 2014. URL https://martinfowler.com/articles/microservices.html.

[86] Nginx. Adopting microservices at netflix: Lessons for architectural design, Feb 2015. URL https://www.nginx.com/blog/microservices-at-netfli].

[87] Newstack. What led amazon to its own microservices architecture, October 2015. URL https://thenewstack.io/led-amazon-microservices-architecture/.

[88] Thomas Erl. *Service-oriented architecture.* Pearson Education Incorporated, 1900.

[89] Paolo Di Francesco, Patrizio Lago, and Ivano Malavolta. Research on architecting microservices: Trends, focus, and potential for industrial adoption. *IEEE*, 04 2017.

[90] Pooyan Jamshidi, Claus Pahl, Nabor C Mendonça, James Lewis, and Stefan Tilkov. Microservices: The journey so far and challenges ahead. *IEEE Software*, 35(3):24–35, 2018.

[91] Chris Richardson. Service discovery in a microservices architecture, May 2016. URL https://www.nginx.com/blog/service-discovery-in-a-microservices-architecture.

[92] Floyd Smith and Owen Garrett. Service mesh, April 2018. URL https://www.nginx.com/blog/what-is-a-service-mesh/.

[93] Jacopo Soldani, Damian Andrew Tamburri, and Willem-Jan Van Den Heuvel. The pains and gains of microservices: A systematic grey literature review. *Journal of Systems and Software*, 146:215–232, 2018.

[94] Francisco Javier Acosta Padilla. *Self-adaptation for Internet of things applications*. PhD thesis, 2016. URL http://www.theses.fr/2016REN1S094. Thèse de doctorat dirigée par Weis, Frédéric et Bourcier, Johann Informatique Rennes 1 2016.

[95] Federico Ciccozzi and Romina Spalazzese. Mde4iot: supporting the internet of things with model-driven engineering. In *International Symposium on Intelligent and Distributed Computing*, pages 67–76. Springer, 2016.

[96] Fahed Alkhabbas, Romina Spalazzese, and Paul Davidsson. Architecting emergent configurations in the internet of things. In *2017 IEEE International Conference on Software Architecture (ICSA)*, pages 221–224. IEEE, 2017.

[97] Fahed Alkhabbas, Romina Spalazzese, and Paul Davidsson. Eco-iot: An architectural approach for realizing emergent configurations in the internet of things. In *European Conference on Software Architecture*, pages 86–102. Springer, 2018.

[98] Mirko D'Angelo, Annalisa Napolitano, and Mauro Caporuscio. Cyphef: a model-driven engineering framework for self-adaptive cyber-physical systems. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, pages 101–104. ACM, 2018.

[99] M Usman Iftikhar and Danny Weyns. Activforms: Active formal models for self-adaptation. In *Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pages 125–134, 2014.

[100] M Usman Iftikhar and Danny Weyns. Activforms: A runtime environment for architecture-based adaptation with guarantees. In *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, pages 278–281. IEEE, 2017.

[101] Arthur Gatouillat, Youakim Badr, and Bertrand Massot. Qos-driven self-adaptation for critical iot-based systems. In *International Conference on Service-Oriented Computing*, pages 93–105. Springer, 2017.

[102] Danny Weyns, M Usman Iftikhar, Danny Hughes, and Nelson Matthys. Applying architecture-based adaptation to automate the management of internet-of-things. In *European Conference on Software Architecture*, pages 49–67. Springer, 2018.

[103] Mahyar T Moghaddam, Eric Rutten, Philippe Lalanda, and Guillaume Giraud. Ias: an iot architectural self-adaptation framework. In *European Conference on Software Architecture*, pages 333–351. Springer, 2020.

[104] Fahimen Alizadeh Moghaddam, Patricia Lago, and Iulia Cristina Ban. Self-adaptation approaches for energy efficiency: A systematic literature review. In *Proceedings of the 6th International Workshop on Green and Sustainable Software*, GREENS '18, pages 35–42, 2018. ISBN 978-1-4503-5732-6.

[105] Javier Mendonca Costa and Guowang Miao. Context-aware machine-to-machine communications. In *Computer Communications Workshops (INFOCOM WKSHPS), 2014 IEEE Conference on*, pages 730–735. IEEE, 2014.

[106] María V Moreno-Cano, José Santa, Miguel A Zamora-Izquierdo, and Antonio F Skarmeta. Future human-centric smart environments. In *Modeling and Processing for Next-Generation Big-Data Technologies*, pages 341–365. Springer, 2015.

[107] Simon Wunderlich, Juan A Cabrera, Frank HP Fitzek, and Martin Reisslein. Network coding in heterogeneous multicore iot nodes with dag scheduling of parallel matrix block operations. *IEEE Internet of Things Journal*, 4(4):917–933, 2017.

[108] Massimiliano Raciti. *Anomaly detection and its adaptation: Studies on cyber-physical systems*. PhD thesis, Linköping University Electronic Press, 2013.

[109] Ronny Seiger, Steffen Huber, Peter Heisig, and Uwe Aßmann. Toward a framework for self-adaptive workflows in cyber-physical systems. *Software & Systems Modeling*, pages 1–18, 2017.

[110] Ahmed Elkhodary, Naeem Esfahani, and Sam Malek. Fusion: a framework for engineering self-tuning self-adaptive software systems. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, pages 7–16, 2010.

[111] Naeem Esfahani, Ahmed Elkhodary, and Sam Malek. A learning-based framework for engineering feature-oriented self-adaptive software systems. *IEEE transactions on software engineering*, 39(11):1467–1493, 2013.

[112] Ivan Dario Paez Anaya, Viliam Simko, Johann Bourcier, Noël Plouzeau, and Jean-Marc Jézéquel. A prediction-driven adaptation approach for self-adaptive sensor networks. In *Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pages 145–154. ACM, 2014.

[113] Alexey Tsymbal. The problem of concept drift: definitions and related work. *Computer Science Department, Trinity College Dublin*, 106(2):58, 2004.

[114] Han Nguyen Ho and Eunseok Lee. Model-based reinforcement learning approach for planning in self-adaptive software system. In *Proceedings of the 9th International Conference on Ubiquitous Information Management and Communication*, page 103. ACM, 2015.

[115] Pawel Idziak and Siobhán Clarke. An analysis of decision-making techniques in dynamic, self-adaptive systems. In *Self-Adaptive and Self-Organizing Systems Workshops (SASOW), 2014 IEEE Eighth International Conference on*, pages 137–143. IEEE, 2014.

[116] Tao Chen and Rami Bahsoon. Self-adaptive and online qos modeling for cloud-based software services. *IEEE Transactions on Software Engineering*, 43(5):453–475, 2017.

[117] Nathalia Moraes do Nascimento and Carlos José Pereira de Lucena. Fiot: An agent-based framework for self-adaptive and self-organizing applications based on the internet of things. *Information Sciences*, 378:161–176, 2017.

[118] Piergiuseppe Mallozzi. Combining machine-learning with invariants assurance techniques for autonomous systems. In *Proceedings of the 39th International Conference on Software Engineering Companion*, pages 485–486. IEEE Press, 2017.

[119] Jeroen Van Der Donckt, Danny Weyns, M Usman Iftikhar, and Sarpreet Singh Buttar. Effective decision making in self-adaptive systems using cost-benefit analysis at runtime and online learning of adaptation spaces. In *International Conference on Evaluation of Novel Approaches to Software Engineering*, pages 373–403. Springer, 2018.

[120] Pooyan Jamshidi, Javier Cámara, Bradley Schmerl, Christian Käestner, and David Garlan. Machine learning meets quantitative planning: Enabling self-adaptation in autonomous robots. In *2019 IEEE/ACM 14th International Symposium on*

Software Engineering for Adaptive and Self-Managing Systems (SEAMS), pages 39–50. IEEE, 2019.

[121] Federico Quin, Thomas Bamelis, Singh Buttar Sarpreet, and Sam Michiels. Efficient analysis of large adaptation spaces in self-adaptive systems using machine learning. In *2019 IEEE/ACM 14th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, pages 1–12. IEEE, 2019.

[122] Jeroen Van Der Donckt, Danny Weyns, Federico Quin, Jonas Van Der Donckt, and Sam Michiels. Applying deep learning to reduce large adaptation spaces of self-adaptive systems with multiple types of goals. In *Proceedings of the IEEE/ACM 15th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pages 20–30, 2020.

[123] Gabriel A Moreno, Javier Cámara, David Garlan, and Bradley Schmerl. Flexible and efficient decision-making for proactive latency-aware self-adaptation. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 13(1):1–36, 2018.

[124] W3C. OWL-S: Semantic Markup for Web Services, 2004.

[125] ESSI WSMO working group. Web Services Modeling Ontology, 2007.

[126] Massimo Paolucci, Takahiro Kawamura, Terry Payne, and Katia Sycara. Semantic matching of web services capabilities. In *First International Semantic Web Conference*, 2002.

[127] Sonia Ben Mokhtar, Davy Preuveneers, Nikolaos Georgantas, Valérie Issarny, and Yolande Berbers. EASY: Efficient semantic service discovery in pervasive computing environments with QoS and context support. *J. Syst. Softw.*, 81:785–808, 2008.

[128] Zambonelli et al. Self-aware pervasive service ecosystems. *Procedia Computer Science*, 7(0):197 – 199, 2011.

[129] Graeme Stevenson, Juan Ye, Simon Dobson, Danilo Pianini, Sara Montagna, and Mirko Viroli. Combining self-organisation, context-awareness and semantic reasoning: the case of resource discovery in opportunistic networks. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, SAC '13, pages 1369–1376, 2013.

[130] Mauro Caporuscio, Vincenzo Grassi, Moreno Marzolla, and Raffaela Mirandola. GoPrime: a fully decentralized middleware for utility-aware service assembly.

IEEE Transactions on Software Engineering, 42(2):136–152, 2016. ISSN 0098-5589.

[131] Richi Nayak and Cindy Tong. Applications of data mining in web services. In Xiaofang Zhou, Stanley Su, Mike P. Papazoglou, Maria Elzbieta Orlowska, and Keith Jeffery, editors, *Web Information Systems – WISE 2004*, pages 199–205, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg. ISBN 978-3-540-30480-7.

[132] J. Andersson, A. Heberle, J. Kirchner, and W. Lowe. Service level achievements – distributed knowledge for optimal service selection. In *2011 IEEE Ninth European Conference on Web Services*, pages 125–132, 2011.

[133] Zeina Houmani, Daniel Balouek-Thomert, Eddy Caron, and Manish Parashar. Enhancing microservices architectures using data-driven service discovery and QoS guarantees. In *20th International Symposium on Cluster, Cloud and Internet Computing*, 2020.

[134] Mirko D'Angelo, Mauro Caporuscio, Vincenzo Grassi, and Raffaela Mirandola. Decentralized learning for self-adaptive qos-aware service assembly. *Future Generation Computer Systems*, 108:210 – 227, 2020. ISSN 0167-739X.

[135] Luca Florio and Elisabetta Di Nitto. Gru: An approach to introduce decentralized autonomic behavior in microservices architectures. In *2016 IEEE International Conference on Autonomic Computing (ICAC)*, pages 357–362. IEEE, 2016.

[136] Elisabetta Di Nitto, Luca Florio, and Damian A Tamburri. Autonomic decentralized microservices: The gru approach and its evaluation. In *Microservices*, pages 209–248. Springer, 2020.

[137] Nabor C. Mendonça, David Garlan, Bradley R. Schmerl, and Javier Cámara. Generality vs. reusability in architecture-based self-adaptation: the case for self-adaptive microservices. In *Proceedings of the 12th European Conference on Software Architecture: Companion Proceedings, ECSA 2018*, pages 18:1–18:6, 2018.

[138] Krasimir Baylov and Aleksandar Dimov. *Reference Architecture for Self-adaptive Microservice Systems*, pages 297–303. Springer International Publishing, Cham, 2018.

[139] Hamzeh Khazaei, Alireza Ghanbari, and Marin Litoiu. Adaptation as a service. In *CASCON*, pages 282–288, 2018.

[140] Carlos M. Aderaldo, Nabor C. Mendonça, Bradley Schmerl, and David Garlan. Kubow: An architecture-based self-adaptation service for cloud native applications. In *Proceedings of the 13th European Conference on Software Architecture - Volume 2*, ECSA '19, page 42–45, 2019.

[141] Basel Magableh and Muder Almiani. A self healing microservices architecture: A case study in docker swarm cluster. In *Advanced Information Networking and Applications*, pages 846–858. Springer International Publishing, 2020.

[142] Andreas Metzger, Osama Sammodi, and Klaus Pohl. Accurate proactive adaptation of service-oriented systems. In *Assurances for Self-Adaptive Systems*, pages 240–265. Springer, 2013.

[143] Anders SG Andrae and Tomas Edler. On global electricity usage of communication technology: trends to 2030. *Challenges*, 6(1):117–157, 2015.

[144] EU Union. Energy efficiency, March 2016. URL https://preview.tinyurl.com/yy87qdj7.

[145] Jason Brownlee. *Deep learning for time series forecasting: Predict the future with MLPs, CNNs and LSTMs in Python.* Machine Learning Mastery, 2018.

[146] Andrew Ng. Machine learning yearning. *URL: http://www. mlyearning. org/(96)*, 2017.

[147] Sima Siami-Namini, Neda Tavakoli, and Akbar Siami Namin. A comparison of arima and lstm in forecasting time series. In *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*, pages 1394–1401. IEEE, 2018.

[148] Sima Siami-Namini, Akbar Siami Namin, et al. Forecasting economics and financial time series: Arima vs. lstm. Technical report, 2018.

[149] Spyros Makridakis, Evangelos Spiliotis, and Vassilios Assimakopoulos. The m4 competition: 100,000 time series and 61 forecasting methods. *International Journal of Forecasting*, 36(1):54–74, 2020.

[150] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.

[151] Eduardo F Camacho and Carlos Bordons Alba. *Model predictive control.* Springer Science & Business Media, 2013.

[152] Mariam Kiran, Peter Murphy, Inder Monga, Jon Dugan, and Sartaj Singh Baveja. Lambda architecture for cost-effective batch and speed big data processing. In *2015 IEEE International Conference on Big Data (Big Data)*, pages 2785–2792. IEEE, 2015.

[153] Ahcène Bounceur. Cupcarbon: A new platform for designing and simulating smart-city and iot wireless sensor networks (sci-wsn). In *Proceedings of the International Conference on Internet of Things and Cloud Computing*, ICC '16, pages 1:1–1:1, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4063-2. doi: 10.1145/2896387.2900336. URL http://doi.acm.org/10.1145/2896387.2900336.

[154] CupCarbon. Cupcarbon simulator. URL http://www.cupcarbon.com.

[155] Maxim Chernyshev, Zubair Baig, Oladayo Bello, and Sherali Zeadally. Internet of things (iot): Research, simulators, and testbeds. *IEEE Internet of Things Journal*, 5(3):1637–1647, 2017.

[156] Cristina Lopez-Pavon, Sandra Sendra, Juan F Valenzuela-Valdés, et al. Evaluation of cupcarbon network simulator for wireless sensor networks. *Netw. Protoc. Algorithms*, 10(2):1–27, 2018.

[157] Wafa'a Kassab and Khalid A Darabkh. A–z survey of internet of things: Architectures, protocols, applications, recent advances, future directions and recommendations. *Journal of Network and Computer Applications*, page 102663, 2020.

[158] Apache Foundation. Apache kafka, May 2020. URL https://kafka.apache.org.

[159] Roshan Sumbaly, Jay Kreps, and Sam Shah. The big data ecosystem at linkedin. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 1125–1134. ACM, 2013.

[160] Elasticsearch. Elasticsearch service, July 2020. URL https://www.elastic.co.

[161] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: a system for large-scale machine learning. In *OSDI*, volume 16, pages 265–283, 2016.

[162] keras. Keras: Deep learning library, March 2020. URL https://keras.io.

[163] Kibana. Kibana, July 2020. URL https://www.elastic.co/products/kibana.

[164] Eva Ostertagova and Oskar Ostertag. Forecasting using simple exponential smoothing method. *Acta Electrotechnica et Informatica*, 12(3):62, 2012.

[165] Rob J Hyndman, Anne B Koehler, Ralph D Snyder, and Simone Grose. A state space framework for automatic forecasting using exponential smoothing methods. *International Journal of forecasting*, 18(3):439–454, 2002.

[166] Chris Chatfield. The holt-winters forecasting procedure. *Journal of the Royal Statistical Society: Series C (Applied Statistics)*, 27(3):264–279, 1978.

[167] Everette S Gardner Jr and Ed McKenzie. Note—seasonal exponential smoothing with damped trends. *Management Science*, 35(3):372–376, 1989.

[168] Timo Koskela, Mikko Lehtokangas, Jukka Saarinen, and Kimmo Kaski. Time series prediction with multilayer perceptron, fir and elman neural networks. In *Proceedings of the World Congress on Neural Networks*, pages 491–496. Citeseer, 1996.

[169] Forecast_x. Forecast_x: Toolkit with naive models for time series, March 2020. URL https://pypi.org/project/forecast-x/.

[170] statsmodels. Statistical models, March 2020. URL https://www.statsmodels.org/stable/index.html.

[171] Apache Foundation. Apache spark, May 2020. URL https://spark.apache.org/streaming/.

[172] Luigi Atzori, Antonio Iera, and Giacomo Morabito. The internet of things: A survey. *Computer networks*, 54(15):2787–2805, 2010.

[173] Jens Malmodin and Dag Lundén. The energy and carbon footprint of the global ict and e&m sectors 2010–2015. *Sustainability*, 10(9):3027, 2018.

[174] Michel Tokic and Haitham Bou Ammar. Teaching reinforcement learning using a physical robot.

[175] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521 (7553):436–444, 2015.

[176] Shahin Jabbari, Matthew Joseph, Michael Kearns, Jamie Morgenstern, and Aaron Roth. Fairness in reinforcement learning. In *International Conference on Machine Learning*, pages 1617–1626. PMLR, 2017.

[177] Marta Kwiatkowska, Gethin Norman, and David Parker. Prism 4.0: Verification of probabilistic real-time systems. In *International conference on computer aided verification*, pages 585–591. Springer, 2011.

[178] Luis Nell Steve Menard et al. Jpype, September 2020. URL https://jpype.readthedocs.io/en/latest/.

[179] The Tornado Authors. Tornado web framework, September 2020. URL https://www.tornadoweb.org/en/stable/.

[180] Microsoft. Azure iot reference architecture, ver. 2.1, 2018. https://aka.ms/iotrefarchitecture.

[181] Todd Hester, Matej Vecerik, Olivier Pietquin, Marc Lanctot, Tom Schaul, Bilal Piot, Dan Horgan, John Quan, Andrew Sendonaris, Ian Osband, et al. Deep q-learning from demonstrations. In *AAAI*, 2018.

[182] Nabor C Mendonça, Pooyan Jamshidi, David Garlan, and Claus Pahl. Developing self-adaptive microservice systems: Challenges and directions. *IEEE Software*, 2019.

[183] Antonio Bucchiarone, Martina De Sanctis, Annapaola Marconi, Marco Pistore, and Paolo Traverso. Incremental composition for adaptive by-design service based systems. In *IEEE International Conference on Web Services, ICWS*, pages 236–243, 2016.

[184] Martina De Sanctis, Romina Spalazzese, and Catia Trubiani. Qos-based formation of software architectures in the internet of things. In *Software Architecture - 13th European Conference, ECSA 2019*, pages 178–194, 2019.

[185] Istio Authors. Istio, August 2020. URL https://istio.io.

[186] Docker Inc. Docker, August 2020. URL https://www.docker.com.

[187] Ilias Gerostathopoulos, Tomas Bures, et al. A toolbox for realtime timeseries anomaly detection. In *2020 IEEE International Conference on Software Architecture Companion (ICSA-C)*, pages 278–281. IEEE, 2020.

[188] Ioannis E Livieris, Emmanuel Pintelas, and Panagiotis Pintelas. A cnn–lstm model for gold price time-series forecasting. *Neural Computing and Applications*, pages 1–10, 2020.

[189] D van Kuppevelt, C Meijer, F Huber, A van der Ploeg, S Georgievska, and VT van Hees. Mcfly: Automated deep learning on time series. *SoftwareX*, 12:100548, 2020.

[190] Gabriel A Moreno, Bradley Schmerl, and David Garlan. Swim: an exemplar for evaluation and comparison of self-adaptation approaches for web applications. In *2018 IEEE/ACM 13th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, pages 137–143. IEEE, 2018.

[191] Sinno Jialin Pan and Qiang Yang. A survey on transfer learning. *IEEE Transactions on knowledge and data engineering*, 22(10):1345–1359, 2009.